man pages section 3: Networking Library Functions

ORACLE®

Contents

Preface

Both novice users and those familar with the SunOS operating system can use online man pages to obtain information about the system and its features. A man page is intended to answer concisely the question "What does it do?" The man pages in general comprise a reference manual. They are not intended to be a tutorial.

Overview

The following contains a brief description of each man page section and the information it references:

- Section 1 describes, in alphabetical order, commands available with the operating system.

- Section 1M describes, in alphabetical order, commands that are used chiefly for system maintenance and administration purposes.

- Section 2 describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value.

- Section 3 describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2.

- Section 4 outlines the formats of various files. The C structure declarations for the file formats are given where applicable.

- Section 5 contains miscellaneous documentation such as character-set tables.

- Section 7 describes various special files that refer to specific hardware peripherals and device drivers. STREAMS software drivers, modules and the STREAMS-generic set of system calls are also described.

- Section 9E describes the DDI (Device Driver Interface)/DKI (Driver/Kernel Interface), DDI-only, and DKI-only entry-point routines a developer can include in a device driver.

- Section 9F describes the kernel functions available for use by device drivers.

- Section 9S describes the data structures used by drivers to share information between the driver and the kernel.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report,

there is no BUGS section. See the intro pages for more information and detail about each section, and man(1) for more information about man pages in general.

NAME

This section gives the names of the commands or functions documented, followed by a brief description of what they do.

SYNOPSIS

This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full path name is shown. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

[ ]   Brackets. The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.

. . .   Ellipses. Several values can be provided for the previous argument, or the previous argument can be specified multiple times, for example, "filename...".

| Separator. Only one of the arguments separated by this character can be specified at a time.

{ }   Braces. The options and/or arguments enclosed within braces are interdependent, such that everything enclosed must be treated as a unit.

PROTOCOL

This section occurs only in subsection 3R to indicate the protocol description file.

DESCRIPTION

This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES. Interactive commands, subcommands, requests, macros, and functions are described under USAGE.

IOCTL

This section appears on pages in Section 7 only. Only the device class that supplies appropriate parameters to the ioctl(2) system call is called ioctl and generates its own heading. ioctl calls for a specific device are listed alphabetically (on the man page for that specific device).

ioctl calls are used for a particular class of devices all of which have an io ending, such as mtio(7I).

OPTIONS
This section lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.

OPERANDS
This section lists the command operands and describes how they affect the actions of the command.

OUTPUT
This section describes the output – standard output, standard error, or output files – generated by the command.

RETURN VALUES
If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES.

ERRORS
On failure, most functions place an error code in the global variable errno indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.

USAGE
This section lists special rules, features, and commands that require in-depth explanations. The subsections listed here are used to explain built-in functionality:

Commands
Modifiers
Variables
Expressions
Input Grammar

EXAMPLES
This section provides examples of usage or of how to use a command or function. Wherever possible a complete

17

example including command-line entry and machine response is shown. Whenever an example is given, the prompt is shown as example%, or if the user must be superuser, example#. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS, and USAGE sections.

ENVIRONMENT VARIABLES   This section lists any environment variables that the command or function affects, followed by a brief description of the effect.

EXIT STATUS   This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion, and values other than zero for various error conditions.

FILES   This section lists all file names referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.

ATTRIBUTES   This section lists characteristics of commands, utilities, and device drivers by defining the attribute type and its corresponding value. See attributes(5) for more information.

SEE ALSO   This section lists references to other man pages, in-house documentation, and outside publications.

DIAGNOSTICS   This section lists diagnostic messages with a brief explanation of the condition causing the error.

WARNINGS   This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.

NOTES   This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.

BUGS   This section describes known bugs and, wherever possible, suggests workarounds.

Networking Library Functions

Name   accept – accept a connection on a socket

Synopsis   cc [*flag* ...] *file* ... -lsocket -lnsl [*library* ...]
#include <sys/socket.h>

int accept(int *s*, struct sockaddr **addr*, socklen_t **addrlen*);

Description   The argument *s* is a socket that has been created with socket(3SOCKET) and bound to an
address with bind(3SOCKET), and that is listening for connections after a call to
listen(3SOCKET). The accept() function extracts the first connection on the queue of
pending connections, creates a new socket with the properties of *s*, and allocates a new file
descriptor, *ns*, for the socket. If no pending connections are present on the queue and the
socket is not marked as non-blocking, accept() blocks the caller until a connection is present.
If the socket is marked as non-blocking and no pending connections are present on the queue,
accept() returns an error as described below. The accept() function uses the netconfig(4)
file to determine the STREAMS device file name associated with *s*. This is the device on which
the connect indication will be accepted. The accepted socket, *ns*, is used to read and write data
to and from the socket that connected to *ns*. It is not used to accept more connections. The
original socket (*s*) remains open for accepting further connections.

The argument *addr* is a result parameter that is filled in with the address of the connecting
entity as it is known to the communications layer. The exact format of the *addr* parameter is
determined by the domain in which the communication occurs.

The argument *addrlen* is a value-result parameter. Initially, it contains the amount of space
pointed to by *addr*; on return it contains the length in bytes of the address returned.

The accept() function is used with connection-based socket types, currently with
SOCK_STREAM.

It is possible to select(3C) or poll(2) a socket for the purpose of an accept() by selecting or
polling it for a read. However, this will only indicate when a connect indication is pending; it is
still necessary to call accept().

Return Values   The accept() function returns −1 on error. If it succeeds, it returns a non-negative integer
that is a descriptor for the accepted socket.

Errors   accept() will fail if:

EBADF	The descriptor is invalid.
ECONNABORTED	The remote side aborted the connection before the accept() operation completed.
EFAULT	The *addr* parameter or the *addrlen* parameter is invalid.
EINTR	The accept() attempt was interrupted by the delivery of a signal.
EMFILE	The per-process descriptor table is full.

ENODEV	The protocol family and type corresponding to *s* could not be found in the `netconfig` file.
ENOMEM	There was insufficient user memory available to complete the operation.
ENOSR	There were insufficient STREAMS resources available to complete the operation.
ENOTSOCK	The descriptor does not reference a socket.
EOPNOTSUPP	The referenced socket is not of type SOCK_STREAM.
EPROTO	A protocol error has occurred; for example, the STREAMS protocol stack has not been initialized or the connection has already been released.
EWOULDBLOCK	The socket is marked as non-blocking and no connections are present to be accepted.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also poll(2), bind(3SOCKET), connect(3SOCKET), listen(3SOCKET), select(3C), socket.h(3HEAD), socket(3SOCKET), netconfig(4), attributes(5)

Name accept – accept a new connection on a socket

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <sys/socket.h>

int accept(int *socket*, struct sockaddr *restrict *address*,
    socklen_t *restrict *address_len*);

Description The accept() function extracts the first connection on the queue of pending connections, creates a new socket with the same socket type protocol and address family as the specified socket, and allocates a new file descriptor for that socket.

The function takes the following arguments:

socket   Specifies a socket that was created with socket(3XNET), has been bound to an address with bind(3XNET), and has issued a successful call to listen(3XNET).

address   Either a null pointer, or a pointer to a sockaddr structure where the address of the connecting socket will be returned.

address_len   Points to a socklen_t which on input specifies the length of the supplied sockaddr structure, and on output specifies the length of the stored address.

If *address* is not a null pointer, the address of the peer for the accepted connection is stored in the sockaddr structure pointed to by *address*, and the length of this address is stored in the object pointed to by *address_len*.

If the actual length of the address is greater than the length of the supplied sockaddr structure, the stored address will be truncated.

If the protocol permits connections by unbound clients, and the peer is not bound, then the value stored in the object pointed to by *address* is unspecified.

If the listen queue is empty of connection requests and O_NONBLOCK is not set on the file descriptor for the socket, accept() will block until a connection is present. If the listen(3XNET) queue is empty of connection requests and O_NONBLOCK is set on the file descriptor for the socket, accept() will fail and set errno to EAGAIN or EWOULDBLOCK.

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

Usage When a connection is available, select(3C) will indicate that the file descriptor for the socket is ready for reading.

Return Values Upon successful completion, accept() returns the nonnegative file descriptor of the accepted socket. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The accept() function will fail if:

EAGAIN	
EWOULDBLOCK	O_NONBLOCK is set for the socket file descriptor and no connections are present to be accepted.
EBADF	The *socket* argument is not a valid file descriptor.
ECONNABORTED	A connection has been aborted.
EFAULT	The *address* or *address_len* parameter can not be accessed or written.
EINTR	The accept() function was interrupted by a signal that was caught before a valid connection arrived.
EINVAL	The *socket* is not accepting connections.
EMFILE	OPEN_MAX file descriptors are currently open in the calling process.
ENFILE	The maximum number of file descriptors in the system are already open.
ENOTSOCK	The *socket* argument does not refer to a socket.
EOPNOTSUPP	The socket type of the specified socket does not support accepting connections.

The accept() function may fail if:

ENOBUFS	No buffer space is available.
ENOMEM	There was insufficient memory available to complete the operation.
ENOSR	There was insufficient STREAMS resources available to complete the operation.
EPROTO	A protocol error has occurred; for example, the STREAMS protocol stack has not been initialized.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also bind(3XNET), connect(3XNET), listen(3XNET), socket(3XNET), attributes(5), standards(5)

Name ber_decode, ber_alloc_t, ber_free, ber_bvdup, ber_init, ber_flatten, ber_get_next, ber_skip_tag, ber_peek_tag, ber_scanf, ber_get_int, ber_get_stringa, ber_get_stringal, ber_get_stringb, ber_get_null, ber_get_boolean, ber_get_bitstring, ber_first_element, ber_next_element, ber_bvfree, ber_bvecfree – Basic Encoding Rules library decoding functions

Synopsis cc [*flag*...] *file*... -lldap [*library*...]
```
#include <lber.h>
```

```
BerElement *ber_alloc_t(int options);
```

```
struct berval *ber_bvdup(const struct berval *bv);
```

```
void ber_free(BerElement *ber, int freebuf);
```

```
BerElement *ber_init(const struct berval *bv);
```

```
int ber_flatten(BerElement *ber, struct berval **bvPtr);
```

```
ber_tag_t ber_get_next(Sockbuf *sb, ber_len_t *len, BerElement *ber);
```

```
ber_tag_t ber_skip_tag(BerElement *ber, ber_len_t *len);
```

```
ber_tag_t ber_peek_tag(BerElement *ber, ber_len_t *len);
```

```
ber_tag_t ber_get_int(BerElement *ber, ber_int_t *num);
```

```
ber_tag_t ber_get_stringb(BerElement *ber, char *buf,
    ber_len_t *len);
```

```
ber_tag_t ber_get_stringa(BerElement *ber, char **buf);
```

```
ber_tag_t ber_get_stringal(BerElement *ber, struct berval **bv);
```

```
ber_tag_t ber_get_null(BerElement *ber);
```

```
ber_tag_t ber_get_boolean(BerElement *ber, int *boolval);
```

```
ber_tag_t ber_get_bitstringa(BerElement *ber, char **buf,
    ber_len_t *len);
```

```
ber_tag_t ber_first_element(BerElement *ber, ber_len_t *len,
    char **last);
```

```
ber_tag_t ber_next_element(BerElement *ber, ber_len_t *len,
    char *last);
```

```
ber_tag_t ber_scanf(BerElement *ber, const char *fmt [, arg...]);
```

```
void ber_bvfree(struct berval *bv);
```

```
void ber_bvecfree(struct berval **bvec);
```

Description These functions provide a subfunction interface to a simplified implementation of the Basic Encoding Rules of ASN.1. The version of BER these functions support is the one defined for the LDAP protocol. The encoding rules are the same as BER, except that only definite form lengths are used, and bitstrings and octet strings are always encoded in primitive form. In

addition, these lightweight BER functions restrict tags and class to fit in a single octet (this means the actual tag must be less than 31). When a "tag" is specified in the descriptions below, it refers to the tag, class, and primitive or constructed bit in the first octet of the encoding. This man page describes the decoding functions in the lber library. See ber_encode(3LDAP) for details on the corresponding encoding functions.

Normally, the only functions that need be called by an application are ber_get_next() to get the next BER element and ber_scanf() to do the actual decoding. In some cases, ber_peek_tag() may also need to be called in normal usage. The other functions are provided for those applications that need more control than ber_scanf() provides. In general, these functions return the tag of the element decoded, or −1 if an error occurred.

The ber_get_next() function is used to read the next BER element from the given Sockbuf, *sb*. A Sockbuf consists of the descriptor (usually socket, but a file descriptor works just as well) from which to read, and a BerElement structure used to maintain a buffer. On the first call, the *sb_ber* struct should be zeroed. It strips off and returns the leading tag byte, strips off and returns the length of the entire element in *len*, and sets up *ber* for subsequent calls to ber_scanf(), and all to decode the element.

The ber_peek_tag() function returns the tag of the next element to be parsed in the BerElement argument. The length of this element is stored in the *lenPtr* argument. LBER_DEFAULT is returned if there is no further data to be read. The decoding position within the *ber* argument is unchanged by this call; that is, the fact that ber_peek_tag() has been called does not affect future use of *ber*.

The ber_skip_tag() function is similar to ber_peek_tag(), except that the state pointer in the BerElement argument is advanced past the first tag and length, and is pointed to the value part of the next element. This function should only be used with constructed types and situations when a BER encoding is used as the value of an OCTET STRING. The length of the value is stored in *lenPtr*.

The ber_scanf() function is used to decode a BER element in much the same way that scanf(3C) works. It reads from *ber*, a pointer to a BerElement such as returned by ber_get_next(), interprets the bytes according to the format string fmt, and stores the results in its additional arguments. The format string contains conversion specifications which are used to direct the interpretation of the BER element. The format string can contain the following characters.

a   Octet string. A char ** should be supplied. Memory is allocated, filled with the contents of the octet string, null-terminated, and returned in the parameter.

s   Octet string. A char * buffer should be supplied, followed by a pointer to an integer initialized to the size of the buffer. Upon return, the null-terminated octet string is put into the buffer, and the integer is set to the actual size of the octet string.

O   Octet string. A struct ber_val ** should be supplied, which upon return points to a memory allocated struct berval containing the octet string and its length. ber_bvfree() can be called to free the allocated memory.

b   Boolean. A pointer to an integer should be supplied.

i   Integer. A pointer to an integer should be supplied.

B   Bitstring. A char ** should be supplied which will point to the memory allocated bits, followed by an unsigned long *, which will point to the length (in bits) of the bitstring returned.

n   Null. No parameter is required. The element is simply skipped if it is recognized.

v   Sequence of octet strings. A char *** should be supplied, which upon return points to a memory allocated null-terminated array of char *'s containing the octet strings. NULL is returned if the sequence is empty.

V   Sequence of octet strings with lengths. A struct berval *** should be supplied, which upon return points to a memory allocated, null-terminated array of struct berval *'s containing the octet strings and their lengths. NULL is returned if the sequence is empty. ber_bvecfree() can be called to free the allocated memory.

x   Skip element. The next element is skipped.

{   Begin sequence. No parameter is required. The initial sequence tag and length are skipped.

}   End sequence. No parameter is required and no action is taken.

[Begin set. No parameter is required. The initial set tag and length are skipped.

]   End set. No parameter is required and no action is taken.

The ber_get_int() function tries to interpret the next element as an integer, returning the result in *num*. The tag of whatever it finds is returned on success, −1 on failure.

The ber_get_stringb() function is used to read an octet string into a pre-allocated buffer. The *len* parameter should be initialized to the size of the buffer, and will contain the length of the octet string read upon return. The buffer should be big enough to take the octet string value plus a terminating NULL byte.

The ber_get_stringa() function is used to allocate memory space into which an octet string is read.

The ber_get_stringal() function is used to allocate memory space into which an octet string and its length are read. It takes a struct berval **, and returns the result in this parameter.

The ber_get_null() function is used to read a NULL element. It returns the tag of the element it skips over.

The ber_get_boolean() function is used to read a boolean value. It is called the same way that ber_get_int() is called.

The ber_get_bitstringa() function is used to read a bitstring value. It takes a char ** which will hold the allocated memory bits, followed by an unsigned long *, which will point to the length (in bits) of the bitstring returned.

The ber_first_element() function is used to return the tag and length of the first element in a set or sequence. It also returns in *last* a magic cookie parameter that should be passed to subsequent calls to ber_next_element(), which returns similar information.

The ber_alloc_t() function constructs and returns BerElement. A null pointer is returned on error. The options field contains a bitwise-OR of options which are to be used when generating the encoding of this BerElement. One option is defined and must always be supplied:

```
#define LBER_USE_DER 0x01
```

When this option is present, lengths will always be encoded in the minimum number of octets. Note that this option does not cause values of sets and sequences to be rearranged in tag and byte order, so these functions are not suitable for generating DER output as defined in X.509 and X.680

The ber_init function constructs a BerElement and returns a new BerElement containing a copy of the data in the *bv* argument. The ber_init function returns the null pointer on error.

The ber_free() function frees a BerElement which is returned from the API calls ber_alloc_t() or ber_init(). Each BerElement must be freed by the caller. The second argument *freebuf* should always be set to 1 to ensure that the internal buffer used by the BER functions is freed as well as the BerElement container itself.

The ber_bvdup() function returns a copy of a *berval*. The *bv_val* field in the returned *berval* points to a different area of memory as the *bv_val* field in the argument *berval*. The null pointer is returned on error (that is, is out of memory).

The ber_flatten() function allocates a struct berval whose contents are BER encoding taken from the *ber* argument. The *bvPtr* pointer points to the returned *berval*, which must be freed using ber_bvfree(). This function returns 0 on success and −1 on error.

Examples EXAMPLE 1 Assume the variable *ber* contains a lightweight BER encoding of the following ASN.1 object:

```
AlmostASearchRequest := SEQUENCE {
    baseObject      DistinguishedName,
    scope           ENUMERATED {
        baseObject      (0),
        singleLevel     (1),
        wholeSubtree    (2)
    },
    derefAliases    ENUMERATED {
        neverDerefaliases   (0),
```

EXAMPLE 1 Assume the variable *ber* contains a lightweight BER encoding of the following ASN.1 object: *(Continued)*

```
        derefInSearching    (1),
        derefFindingBaseObj (2),
        alwaysDerefAliases  (3N)
    },
    sizelimit       INTEGER (0 .. 65535),
    timelimit       INTEGER (0 .. 65535),
    attrsOnly       BOOLEAN,
    attributes      SEQUENCE OF AttributeType
}
```

EXAMPLE 2 The element can be decoded using ber_scanf() as follows.

```
int    scope, ali, size, time, attrsonly;
char   *dn, **attrs;
if ( ber_scanf( ber, "{aiiiib{v}}", &dn, &scope, &ali,
    &size, &time, &attrsonly, &attrs ) == −1 )
        /* error */
else
        /* success */
```

Errors   If an error occurs during decoding, generally these functions return −1.

Notes   The return values for all of these functions are declared in the <lber.h> header. Some functions may allocate memory which must be freed by the calling application.

Attributes   See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also   ber_encode(3LDAP), attributes(5)

Yeong, W., Howes, T., and Hardcastle-Kille, S., "Lightweight Directory Access Protocol", OSI-DS-26, April 1992.

Information Processing - Open Systems Interconnection - Model and Notation - Service Definition - Specification of Basic Encoding Rules for Abstract Syntax Notation One, International Organization for Standardization, International Standard 8825.

Name   ber_encode, ber_alloc, ber_printf, ber_put_int, ber_put_ostring, ber_put_string, ber_put_null, ber_put_boolean, ber_put_bitstring, ber_start_seq, ber_start_set, ber_put_seq, ber_put_set – simplified Basic Encoding Rules library encoding functions

Synopsis   cc[*flag*...] *file*... -lldap[*library*...]
#include <lber.h>

BerElement *ber_alloc();

ber_printf(BerElement *ber, char **fmt[, arg...]);

ber_put_int(BerElement *ber, long num, char tag);

ber_put_ostring(BerElement *ber, char **str, unsigned long len,
    char tag);

ber_put_string(BerElement *ber, char **str, char tag);

ber_put_null(BerElement *ber, char tag);

ber_put_boolean(BerElement *ber, int bool, char tag);

ber_put_bitstring(BerElement *ber, char *str, int blen, char tag);

ber_start_seq(BerElement *ber, char tag);

ber_start_set(BerElement *ber, char tag);

ber_put_seq(BerElement *ber);

ber_put_set(BerElement *ber);

Description   These functions provide a subfunction interface to a simplified implementation of the Basic Encoding Rules of ASN.1. The version of BER these functions support is the one defined for the LDAP protocol. The encoding rules are the same as BER, except that only definite form lengths are used, and bitstrings and octet strings are always encoded in primitive form. In addition, these lightweight BER functions restrict tags and class to fit in a single octet (this means the actual tag must be less than 31). When a "tag" is specified in the descriptions below, it refers to the tag, class, and primitive or constructed bit in the first octet of the encoding. This man page describes the encoding functions in the lber library. See ber_decode(3LDAP) for details on the corresponding decoding functions.

Normally, the only functions that need be called by an application are ber_alloc(), to allocate a BER element, and ber_printf() to do the actual encoding. The other functions are provided for those applications that need more control than ber_printf() provides. In general, these functions return the length of the element encoded, or −1 if an error occurred.

The ber_alloc() function is used to allocate a new BER element.

The ber_printf() function is used to encode a BER element in much the same way that sprintf(3S) works. One important difference, though, is that some state information is kept with the *ber* parameter so that multiple calls can be made to ber_printf() to append things to the end of the BER element. Ber_printf() writes to *ber*, a pointer to a BerElement such as

returned by `ber_alloc()`. It interprets and formats its arguments according to the format string `fmt`. The format string can contain the following characters:

b   Boolean. An integer parameter should be supplied. A boolean element is output.

B   Bitstring. A char * pointer to the start of the bitstring is supplied, followed by the number of bits in the bitstring. A bitstring element is output.

i   Integer. An integer parameter should be supplied. An integer element is output.

n   Null. No parameter is required. A null element is output.

o   Octet string. A char * is supplied, followed by the length of the string pointed to. An octet string element is output.

O   Octet string. A struct berval * is supplied. An octet string element is output.

s   Octet string. A null-terminated string is supplied. An octet string element is output, not including the trailing null octet.

t   Tag. An int specifying the tag to give the next element is provided. This works across calls.

v   Several octet strings. A null-terminated array of char * is supplied. Note that a construct like '{v}' is required to get an actual sequence of octet strings.

{   Begin sequence. No parameter is required.

}   End sequence. No parameter is required.

[Begin set. No parameter is required.

]   End set. No parameter is required.

The `ber_put_int()` function writes the integer element *num* to the BER element *ber*.

The `ber_put_boolean()` function writes the boolean value given by *bool* to the BER element.

The `ber_put_bitstring()` function writes *blen* bits starting at *str* as a bitstring value to the given BER element. Note that *blen* is the length in *bits* of the bitstring.

The `ber_put_ostring()` function writes *len* bytes starting at *str* to the BER element as an octet string.

The `ber_put_string()` function writes the null-terminated string (minus the terminating ") to the BER element as an octet string.

The `ber_put_null()` function writes a NULL element to the BER element.

The `ber_start_seq()` function is used to start a sequence in the BER element. The `ber_start_set()` function works similarly. The end of the sequence or set is marked by the nearest matching call to `ber_put_seq()` or `ber_put_set()`, respectively.

The ber_first_element() function is used to return the tag and length of the first element in a set or sequence. It also returns in *cookie* a magic cookie parameter that should be passed to subsequent calls to ber_next_element(), which returns similar information.

Examples EXAMPLE 1 Assuming the following variable declarations, and that the variables have been assigned appropriately, an BER encoding of the following ASN.1 object:

```
AlmostASearchRequest := SEQUENCE {
    baseObject      DistinguishedName,
    scope           ENUMERATED {
        baseObject     (0),
        singleLevel    (1),
        wholeSubtree   (2)
    },
    derefAliases    ENUMERATED {
        neverDerefaliases   (0),
        derefInSearching    (1),
        derefFindingBaseObj (2),
        alwaysDerefAliases  (3N)
    },
    sizelimit       INTEGER (0 .. 65535),
    timelimit       INTEGER (0 .. 65535),
    attrsOnly       BOOLEAN,
    attributes      SEQUENCE OF AttributeType
}
```

can be achieved like so:

```
int     scope, ali, size, time, attrsonly;
char    *dn, **attrs;

/* ... fill in values ... */
if ( (ber = ber_alloc( )) == NULLBER )
/* error */

if ( ber_printf( ber, "{siiiib{v}}", dn, scope, ali,
    size, time, attrsonly, attrs ) == −1 )
        /* error */
else
        /* success */
```

Return Values If an error occurs during encoding, ber_alloc() returns NULL; other functions generally return −1.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ber_decode(3LDAP), attributes(5)

Yeong, W., Howes, T., and Hardcastle-Kille, S., "Lightweight Directory Access Protocol", OSI-DS-26, April 1992.

Information Processing - Open Systems Interconnection - Model and Notation - Service Definition - Specification of Basic Encoding Rules for Abstract Syntax Notation One, International Organization for Standardization, International Standard 8825.

Notes The return values for all of these functions are declared in <lber.h>.

Name bind – bind a name to a socket

Synopsis cc [*flag* ...] *file* ... -lsocket -lnsl [*library* ...]
#include <sys/socket.h>

int bind(int *s*, const struct sockaddr **name*, socklen_t *namelen*);

Description The bind() function assigns a name to an unnamed socket. When a socket is created with socket(3SOCKET), it exists in a name space (address family) but has no name assigned. The bind() function requests that the name pointed to by *name* be assigned to the socket.

Return Values Upon successful completion 0 is returned. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The bind() function will fail if:

EACCES	The requested address is protected, and {PRIV_NET_PRIVADDR} is not asserted in the effective set of the current process.
EADDRINUSE	The specified address is already in use.
EADDRNOTAVAIL	The specified address is not available on the local machine.
EBADF	*s* is not a valid descriptor.
EINVAL	*namelen* is not the size of a valid address for the specified address family.
	The socket is already bound to an address.
	Socket options are inconsistent with port attributes.
ENOSR	There were insufficient STREAMS resources for the operation to complete.
ENOTSOCK	*s* is a descriptor for a file, not a socket.

The following errors are specific to binding names in the UNIX domain:

EACCES	Search permission is denied for a component of the path prefix of the pathname in *name*.
EIO	An I/O error occurred while making the directory entry or allocating the inode.
EISDIR	A null pathname was specified.
ELOOP	Too many symbolic links were encountered in translating the pathname in *name*.
ENOENT	A component of the path prefix of the pathname in *name* does not exist.
ENOTDIR	A component of the path prefix of the pathname in *name* is not a directory.
EROFS	The inode would reside on a read-only file system.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also unlink(2), socket(3SOCKET), attributes(5), privileges(5), socket.h(3HEAD)

Notes 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 by using unlink(2).

The rules used in name binding vary between communication domains.

Name   bind – bind a name to a socket

Synopsis   cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <sys/socket.h>

int bind(int *socket*, const struct sockaddr **address*,
    socklen_t *address_len*);

Description   The bind() function assigns an *address* to an unnamed socket. Sockets created with
socket(3XNET) function are initially unnamed. They are identified only by their address
family.

The function takes the following arguments:

socket   Specifies the file descriptor of the socket to be bound.

address   Points to a sockaddr structure containing the address to be bound to the
                socket. The length and format of the address depend on the address family of
                the socket.

address_len   Specifies the length of the sockaddr structure pointed to by the *address*
                argument.

The socket in use may require the process to have appropriate privileges to use the bind()
function.

Usage   An application program can retrieve the assigned socket name with the
getsockname(3XNET) function.

Return Values   Upon successful completion, bind() returns 0. Otherwise, −1 is returned and errno is set to
indicate the error.

Errors   The bind() function will fail if:

EADDRINUSE   The specified address is already in use.

EADDRNOTAVAIL   The specified address is not available from the local machine.

EAFNOSUPPORT   The specified address is not a valid address for the address family of the
                  specified socket.

EBADF   The *socket* argument is not a valid file descriptor.

EFAULT   The *address* argument can not be accessed.

EINVAL   The socket is already bound to an address, and the protocol does not
                  support binding to a new address; or the socket has been shut down.

ENOTSOCK   The *socket* argument does not refer to a socket.

EOPNOTSUPP   The socket type of the specified socket does not support binding to an
                  address.

If the address family of the socket is AF_UNIX, then bind() will fail if:

EACCES   A component of the path prefix denies search permission, or the requested
                 name requires writing in a directory with a mode that denies write
                 permission.

EDESTADDRREQ
EISDIR   The *address* argument is a null pointer.

EIO   An I/O error occurred.

ELOOP   Too many symbolic links were encountered in translating the pathname in
                 address.

ENAMETOOLONG   A component of a pathname exceeded NAME_MAX characters, or an entire
                 pathname exceeded PATH_MAX characters.

ENOENT   A component of the pathname does not name an existing file or the
                 pathname is an empty string.

ENOTDIR   A component of the path prefix of the pathname in *address* is not a
                 directory.

EROFS   The name would reside on a read-only filesystem.

The bind() function may fail if:

EACCES   The specified address is protected, and {PRIV_NET_PRIVADOR} is not
                 asserted in the effective set of the current process.

EINVAL   The *address_len* argument is not a valid length for the address family.

EISCONN   The socket is already connected.

ENAMETOOLONG   Pathname resolution of a symbolic link produced an intermediate result
                 whose length exceeds PATH_MAX.

ENOBUFS   Insufficient resources were available to complete the call.

ENOSR   There were insufficient STREAMS resources for the operation to complete.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also   connect(3XNET), getsockname(3XNET), listen(3XNET), socket(3XNET), attributes(5), privileges(5), standards(5)

Name   byteorder, htonl, htonll, htons, ntohl, ntohll, ntohs – convert values between host and network byte order

Synopsis   cc [*flag*...] *file*... -lsocket -lnsl [*library*...]
```
#include <sys/types.h>
#include <netinet/in.h>
#include <inttypes.h>

uint32_t htonl(uint32_t hostlong);

uint64_t htonll(uint64_t hostlonglong);

uint16_t htons(uint16_t hostshort);

uint32_t ntohl(uint32_t netlong);

uint64_t ntohll(uint64_t hostlonglong);

uint16_t ntohs(uint16_t netshort);
```

Description   These functions convert 16-bit, 32-bit, and 64–bit quantities between network byte order and host byte order. On some architectures these routines are defined as NULL macros in the include file <netinet/in.h>. On other architectures, the routines are functional when the host byte order is different from network byte order.

These functions are most often used in conjunction with Internet addresses and ports as returned by gethostent() and getservent(). See gethostbyname(3NSL) and getservbyname(3SOCKET).

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   gethostbyname(3NSL), getservbyname(3SOCKET), inet.h(3HEAD), attributes(5)

Name cldap_close – dispose of connectionless LDAP pointer

Synopsis cc[*flag*...] *file*... -lldap[*library*...]
```
#include <lber.h>
#include <ldap.h>

void cldap_close(LDAP *ld);
```

Description The cldap_close() function disposes of memory allocated by cldap_open(3LDAP). It should be called when all CLDAP communication is complete.

Parameters *ld*   The LDAP pointer returned by a previous call to cldap_open(3LDAP).

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap(3LDAP), cldap_open(3LDAP), cldap_search_s(3LDAP), cldap_setretryinfo(3LDAP)

Name cldap_open – LDAP connectionless communication preparation

Synopsis cc[*flag*...] *file*... -lldap[*library*...]
#include <lber.h>
#include <ldap.h>

LDAP *cldap_open(char *host*, int *port*);

Parameters *host*   The name of the host on which the LDAP server is running.

port   The port number to connect.

Description The cldap_open() function is called to prepare for connectionless LDAP communication
(over udp(7P)). It allocates an LDAP structure which is passed to future search requests.

If the default IANA-assigned port of 389 is desired, LDAP_PORT should be specified for *port*.
host can contain a space-separated list of hosts or addresses to try. cldap_open() returns a
pointer to an LDAP structure, which should be passed to subsequent calls to
cldap_search_s(3LDAP), cldap_setretryinfo(3LDAP), and cldap_close(3LDAP).
Certain fields in the LDAP structure can be set to indicate size limit, time limit, and how
aliases are handled during operations. See ldap_open(3LDAP) and <ldap.h> for more details.

Errors If an error occurs, cldap_open() will return NULL and errno will be set appropriately.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap(3LDAP) cldap_search_s(3LDAP), cldap_setretryinfo(3LDAP),
cldap_close(3LDAP), attributes(5), udp(7P)

Name cldap_search_s – connectionless LDAP search

Synopsis cc[*flag...*] *file...* -lldap[*library...*]
```
#include <lber.h>
#include <ldap.h>

int cldap_search_s(LDAP *ld, char *base, int scope, char *filter,
    char *attrs, int attrsonly, LDAPMessage **res, char *logdn);
```

Description The cldap_search_s() function performs an LDAP search using the Connectionless LDAP
(CLDAP) protocol.

cldap_search_s() has parameters and behavior identical to that of ldap_search_s(3LDAP),
except for the addition of the *logdn* parameter. *logdn* should contain a distinguished name to
be used only for logging purposed by the LDAP server. It should be in the text format
described by *RFC 1779, A String Representation of Distinguished Names*.

Retransmission
Algorithm

cldap_search_s() operates using the CLDAP protocol over udp(7P). Since UDP is a
non-reliable protocol, a retry mechanism is used to increase reliability. The
cldap_setretryinfo(3LDAP) function can be used to set two retry parameters: *tries*, a count
of the number of times to send a search request and *timeout*, an initial timeout that determines
how long to wait for a response before re-trying. *timeout* is specified seconds. These values are
stored in the ld_cldaptries and ld_cldaptimeout members of the ld LDAP structure, and
the default values set in ldap_open(3LDAP) are 4 and 3 respectively. The retransmission
algorithm used is:

Step 1   Set the current timeout to ld_cldaptimeout seconds, and the current LDAP server
address to the first LDAP server found during the ldap_open(3LDAP) call.

Step 2   Send the search request to the current LDAP server address.

Step 3   Set the wait timeout to the current timeout divided by the number of server
addresses found during ldap_open(3LDAP) or to one second, whichever is larger.
Wait at most that long for a response; if a response is received, STOP. Note that the
wait timeout is always rounded down to the next lowest second.

Step 4   Repeat steps 2 and 3 for each LDAP server address.

Step 5   Set the current timeout to twice its previous value and repeat Steps 2 through 5 a
maximum of *tries* times.

Examples Assume that the default values for *tries* and *timeout* of 4 tries and 3 seconds are used. Further,
assume that a space-separated list of two hosts, each with one address, was passed to
cldap_open(3LDAP). The pattern of requests sent will be (stopping as soon as a response is
received):

```
 Time        Search Request Sent To:
  +0          Host A try 1
  +1 (0+3/2)      Host B try 1
  +2 (1+3/2)      Host A try 2
```

```
+5  (2+6/2)       Host B try 2
+8  (5+6/2)       Host A try 3
+14 (8+12/2)    Host B try 3
+20 (14+12/2)   Host A try 4
+32 (20+24/2)   Host B try 4
+44 (20+24/2)   (give up - no response)
```

Errors cldap_search_s() returns LDAP_SUCCESS if a search was successful and the appropriate LDAP error code otherwise. See ldap_error(3LDAP) for more information.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap(3LDAP), ldap_error(3LDAP), ldap_search_s(3LDAP), cldap_open(3LDAP), cldap_setretryinfo(3LDAP), cldap_close(3LDAP), attributes(5), udp(7P)

Name cldap_setretryinfo – set connectionless LDAP request retransmission parameters

Synopsis cc[*flag*...] *file*... -lldap[*library*...]
#include <lber.h>
#include <ldap.h>

void cldap_setretryinfo(LDAP *ld, int *tries*, int *timeout*);

Parameters *ld* LDAP pointer returned from a previous call to cldap_open(3LDAP).

tries Maximum number of times to send a request.

timeout Initial time, in seconds, to wait before re-sending a request.

Description The cldap_setretryinfo() function is used to set the CLDAP request retransmission
behavior for future cldap_search_s(3LDAP) calls. The default values (set by
cldap_open(3LDAP)) are 4 tries and 3 seconds between tries. See cldap_search_s(3LDAP)
for a complete description of the retransmission algorithm used.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap(3LDAP), cldap_open(3LDAP), cldap_search_s(3LDAP), cldap_close(3LDAP),
attributes(5)

Name   connect – connect a socket

Synopsis   cc [*flag* ...] *file* ... -lsocket -lnsl [*library* ...]
#include <sys/socket.h>

int connect(int *s*, const struct sockaddr **name*, socklen_t *namelen*);

Description   The parameter *s* is a socket. If it is of type SOCK_DGRAM, connect() specifies the peer with
which the socket is to be associated. This address is the address to which datagrams are to be
sent if a receiver is not explicitly designated. This address is the only address from which
datagrams are to be received. If the socket *s* is of type SOCK_STREAM, connect() attempts to
make a connection to another socket. This behavior can be modified by the
SO_PASSIVE_CONNECT socket option provided by setsockopt(3SOCKET). The other socket is
specified by *name*, which is an address in the communication space of the socket. Each
communication space interprets the *name* parameter in its own way. If *s* is not bound, then *s*
will be bound to an address selected by the underlying transport provider. Generally, stream
sockets can successfully connect() only once. Datagram sockets can use connect() multiple
times to change their association. Datagram sockets can dissolve the association by
connecting to a null address.

Return Values   If the connection or binding succeeds, 0 is returned. Otherwise, −1 is returned, errno is set to
indicate the error, and state of the socket is unspecified. Applications should close the file
descriptor and create a new socket before attempting to reconnect.

Errors   The call fails if:

EACCES	Search permission is denied for a component of the path prefix of the pathname in *name*.
EADDRINUSE	The address is already in use.
EADDRNOTAVAIL	The specified address is not available.
EAFNOSUPPORT	Addresses in the specified address family cannot be used with this socket.
EALREADY	The socket is non-blocking, and a previous connection attempt has not yet been completed.
EBADF	*s* is not a valid descriptor.
ECONNREFUSED	The attempt to connect was forcefully rejected.
EINPROGRESS	The socket is non-blocking, and the connection cannot be completed immediately. You can use select(3C) to complete the connection by selecting the socket for writing.
EINTR	The connection attempt was interrupted before any data arrived by the delivery of a signal. The connection, however, will be established asynchronously.
EINVAL	*namelen* is not the size of a valid address for the specified address family.

EIO	An I/O error occurred while reading from or writing to the file system.
EISCONN	The socket is already connected.
ELOOP	Too many symbolic links were encountered in translating the pathname in *name*.
ENETUNREACH	The network is not reachable from this host.
EHOSTUNREACH	The remote host is not reachable from this host.
ENOENT	A component of the path prefix of the pathname in *name* does not exist.
ENOENT	The socket referred to by the pathname in *name* does not exist.
ENOSR	There were insufficient STREAMS resources available to complete the operation.
ENXIO	The server exited before the connection was complete.
ETIMEDOUT	Connection establishment timed out without establishing a connection.
EWOULDBLOCK	The socket is marked as non-blocking, and the requested operation would block.

The following errors are specific to connecting names in the UNIX domain. These errors might not apply in future versions of the UNIX IPC domain.

ENOTDIR	A component of the path prefix of the pathname in *name* is not a directory.
ENOTSOCK	*s* is not a socket.
ENOTSOCK	*name* is not a socket.
EPROTOTYPE	The file that is referred to by *name* is a socket of a type other than type *s*. For example, *s* is a SOCK_DGRAM socket, while *name* refers to a SOCK_STREAM socket.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   close(2), accept(3SOCKET), getsockname(3SOCKET), select(3C), setsockopt(3SOCKET), socket(3SOCKET), socket.h(3HEAD), attributes(5)

Name   connect – connect a socket

Synopsis   cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <sys/socket.h>

int connect(int *socket*, const struct sockaddr **address*,
    socklen_t *address_len*);

Description   The connect() function requests a connection to be made on a socket. The function takes the following arguments:

socket   Specifies the file descriptor associated with the socket.

address   Points to a sockaddr structure containing the peer address. The length and format of the address depend on the address family of the socket.

address_len   Specifies the length of the sockaddr structure pointed to by the *address* argument.

If the socket has not already been bound to a local address, connect() will bind it to an address which, unless the socket's address family is AF_UNIX, is an unused local address.

If the initiating socket is not connection-mode, then connect() sets the socket's peer address, but no connection is made. For SOCK_DGRAM sockets, the peer address identifies where all datagrams are sent on subsequent send(3XNET) calls, and limits the remote sender for subsequent recv(3XNET) calls. If *address* is a null address for the protocol, the socket's peer address will be reset.

If the initiating socket is connection-mode, then connect() attempts to establish a connection to the address specified by the *address* argument.

If the connection cannot be established immediately and O_NONBLOCK is not set for the file descriptor for the socket, connect() will block for up to an unspecified timeout interval until the connection is established. If the timeout interval expires before the connection is established, connect() will fail and the connection attempt will be aborted. If connect() is interrupted by a signal that is caught while blocked waiting to establish a connection, connect() will fail and set errno to EINTR, but the connection request will not be aborted, and the connection will be established asynchronously.

If the connection cannot be established immediately and O_NONBLOCK is set for the file descriptor for the socket, connect() will fail and set errno to EINPROGRESS, but the connection request will not be aborted, and the connection will be established asynchronously. Subsequent calls to connect() for the same socket, before the connection is established, will fail and set errno to EALREADY.

When the connection has been established asynchronously, select(3C) and poll(2) will indicate that the file descriptor for the socket is ready for writing.

The socket in use may require the process to have appropriate privileges to use the connect() function.

Usage If connect() fails, the state of the socket is unspecified. Portable applications should close the file descriptor and create a new socket before attempting to reconnect.

Return Values Upon successful completion, connect() returns 0. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The connect() function will fail if:

EADDRNOTAVAIL	The specified address is not available from the local machine.
EAFNOSUPPORT	The specified address is not a valid address for the address family of the specified socket.
EALREADY	A connection request is already in progress for the specified socket.
EBADF	The *socket* argument is not a valid file descriptor.
ECONNREFUSED	The target address was not listening for connections or refused the connection request.
EFAULT	The address parameter can not be accessed.
EINPROGRESS	O_NONBLOCK is set for the file descriptor for the socket and the connection cannot be immediately established; the connection will be established asynchronously.
EINTR	The attempt to establish a connection was interrupted by delivery of a signal that was caught; the connection will be established asynchronously.
EISCONN	The specified socket is connection-mode and is already connected.
ENETUNREACH	No route to the network is present.
ENOTSOCK	The *socket* argument does not refer to a socket.
EPROTOTYPE	The specified address has a different type than the socket bound to the specified peer address.
ETIMEDOUT	The attempt to connect timed out before a connection was made.

If the address family of the socket is AF_UNIX, then connect() will fail if:

EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating the pathname in *address*.
ENAMETOOLONG	A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX characters.

ENOENT   A component of the pathname does not name an existing file or the pathname is an empty string.

ENOTDIR   A component of the path prefix of the pathname in *address* is not a directory.

The connect() function may fail if:

EACCES   Search permission is denied for a component of the path prefix; or write access to the named socket is denied.

EADDRINUSE   Attempt to establish a connection that uses addresses that are already in use.

ECONNRESET   Remote host reset the connection request.

EHOSTUNREACH   The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).

EINVAL   The *address_len* argument is not a valid length for the address family; or invalid address family in sockaddr structure.

ENAMETOOLONG   Pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.

ENETDOWN   The local interface used to reach the destination is down.

ENOBUFS   No buffer space is available.

ENOSR   There were insufficient STREAMS resources available to complete the operation.

EOPNOTSUPP   The socket is listening and can not be connected.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also close(2), poll(2), accept(3XNET), bind(3XNET), getsockname(3XNET), select(3C), send(3XNET), shutdown(3XNET), socket(3XNET), attributes(5), standards(5)

Name   dial, undial – establish an outgoing terminal line connection

Synopsis   cc [*flag*...] *file*... -lnsl [*library*...]
`#include <dial.h>`

`int dial(CALL `*call*`);`

`void undial(int `*fd*`);`

Description   The `dial()` function returns a file-descriptor for a terminal line open for read/write. The argument to `dial()` is a CALL structure (defined in the header <dial.h>).

When finished with the terminal line, the calling program must invoke `undial()` to release the semaphore that has been set during the allocation of the terminal device.

CALL is defined in the header <dial.h> and has the following members:

```
struct termio *attr;     /* pointer to termio attribute struct */
int           baud;      /* transmission data rate */
int           speed;     /* 212A modem: low=300, high=1200 */
char          *line;     /* device name for out-going line */
char          *telno;    /* pointer to tel-no digits string */
int           modem;     /* specify modem control for direct lines */
char          *device;   /* unused */
int           dev_len;   /* unused */
```

The CALL element speed is intended only for use with an outgoing dialed call, in which case its value should be the desired transmission baud rate. The CALL element baud is no longer used.

If the desired terminal line is a direct line, a string pointer to its device-name should be placed in the line element in the CALL structure. Legal values for such terminal device names are kept in the Devices file. In this case, the value of the baud element should be set to -1. This value will cause dial to determine the correct value from the <Devices> file.

The telno element is for a pointer to a character string representing the telephone number to be dialed. Such numbers may consist only of these characters:

0-9	dial 0-9
*	dail *
#	dail #
=	wait for secondary dial tone
-	delay for approximately 4 seconds

The CALL element modem is used to specify modem control for direct lines. This element should be non-zero if modem control is required. The CALL element attr is a pointer to a termio structure, as defined in the header <termio.h>. A NULL value for this pointer element may be

passed to the dial function, but if such a structure is included, the elements specified in it will be set for the outgoing terminal line before the connection is established. This setting is often important for certain attributes such as parity and baud-rate.

The CALL elements device and dev_len are no longer used. They are retained in the CALL structure for compatibility reasons.

Return Values On failure, a negative value indicating the reason for the failure will be returned. Mnemonics for these negative indices as listed here are defined in the header <dial.h>.

```
INTRPT  −1       /* interrupt occurred */
D_HUNG  −2       /* dialer hung (no return from write) */
NO_ANS  −3       /* no answer within 10 seconds */
ILL_BD  −4       /* illegal baud-rate */
A_PROB  −5       /* acu problem (open( ) failure) */
L_PROB  −6       /* line problem (open( ) failure) */
NO_Ldv  −7       /* can't open Devices file */
DV_NT_A −8       /* requested device not available */
DV_NT_K −9       /* requested device not known */
NO_BD_A −10      /* no device available at requested baud */
NO_BD_K −11      /* no device known at requested baud */
DV_NT_E −12      /* requested speed does not match */
BAD_SYS −13      /* system not in Systems file*/
```

Files /etc/uucp/Devices

/etc/uucp/Systems

/var/spool/uucp/LCK..*tty-device*

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also uucp(1C), alarm(2), read(2), write(2), attributes(5), termio(7I)

Notes Including the header <dial.h> automatically includes the header <termio.h>. An alarm(2) system call for 3600 seconds is made (and caught) within the dial module for the purpose of "touching" the LCK.. file and constitutes the device allocation semaphore for the terminal device. Otherwise, uucp(1C) may simply delete the LCK.. entry on its 90-minute clean-up rounds. The alarm may go off while the user program is in a read(2) or write(2) function, causing an apparent error return. If the user program expects to be around for an hour or more, error returns from read()s should be checked for (errno==EINTR), and the read() possibly reissued.

This interface is unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

Name dlpi_arptype – convert a DLPI MAC type to an ARP hardware type

Synopsis cc [*flag*...] *file*... -ldlpi [*library*...]
#include <libdlpi.h>
uint_t dlpi_arptype(uint_t *dlpitype*);

Description The dlpi_arptype() function converts a DLPI MAC type to an ARP hardware type defined in
<netinet/arp.h>

Return Values Upon success, the corresponding ARP hardware type is returned. Otherwise, zero is returned.

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libdlpi(3LIB), attributes(5)

Name dlpi_bind – bind DLPI handle

Synopsis cc [*flag* ...] *file* ... -ldlpi [*library* ...]
#include <libdlpi.h>

int dlpi_bind(dlpi_handle_t *dh*, uint_t *sap*, uint_t **boundsap*);

Description The dlpi_bind() function attempts to bind the DLPI handle *dh* to the SAP *sap*. The handle must be in the DL_UNBOUND DLPI state and will transition to the DL_IDLE DLPI state upon success. Some DLPI MAC types can bind to a different SAP than the SAP requested, in which case *boundsap* returns the actual bound SAP. If *boundsap* is set to NULL, dlpi_bind() fails if the bound SAP does not match the requested SAP. If the caller does not care which SAP is chosen, DLPI_ANY_SAP can be specified for *sap*. This is primarily useful in conjunction with dlpi_promiscon() and DL_PROMISC_SAP to receive traffic from all SAPs. If DLPI_ANY_SAP is specified, any transmitted messages must explicitly specify a SAP using dlpi_send(3DLPI).

Upon success, the caller can use dlpi_recv(3DLPI) to receive data matching the bound SAP that is sent to the DLPI link associated with *dh*. In addition, the caller can use dlpi_send(3DLPI) to send data over the bound SAP address associated with DLPI handle *dh*. The physical address of the bound handle can be retrieved with dlpi_info(3DLPI).

Return Values Upon success, DLPI_SUCCESS is returned. If DL_SYSERR is returned, errno contains the specific UNIX system error value. Otherwise, a DLPI error value defined in <sys/dlpi.h> or an error value listed in the following section is returned.

Errors DLPI_EBADMSG   Bad DLPI message

DLPI_EINHANDLE   Invalid DLPI handle

DLPI_ETIMEDOUT   DLPI operation timed out

DLPI_EUNAVAILSAP   Unavailable DLPI SAP

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also dlpi_info(3DLPI), dlpi_recv(3DLPI), dlpi_send(3DLPI), dlpi_unbind(3DLPI), libdlpi(3LIB), attributes(5)

Name dlpi_close – close DLPI link

Synopsis cc [*flag* ...] *file* ... -ldlpi [*library* ...]
#include <libdlpi.h>

void *dlpi_close*(dlpi_handle_t *dh*);

Description The dlpi_close() function closes the open DLPI link instance associated with *dh* and
destroys *dh* after closing the DLPI link instance.

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also dlpi_open(3DLPI), libdlpi(3LIB), attributes(5)

Name dlpi_disabnotify – disable DLPI notification

Synopsis cc [*flag...*] *file...* -ldlpi [*library...*]
#include <libdlpi.h>

int dlpi_disabnotify(dlpi_handle_t *dh*, dlpi_notifyid_t *id*,
    void ***argp);

Description The dlpi_disabnotify() function disables the notification registration associated with
identifier *id*. If *argp* is not NULL, the argument *arg* that was passed to
dlpi_enabnotify(3DLPI) during registration is also returned. This operation can be
performed in any DLPI state of a handle.

Closing the DLPI handle *dh* will also remove all associated callback functions.

Return Values Upon success, DLPI_SUCCESS is returned. If DL_SYSERR is returned, errno contains the specific
UNIX system error value. Otherwise, a DLPI error value defined in <sys/dlpi.h> or an error
value listed in the following section is returned.

Errors

DLPI_EINHANDLE	A DLPI handle is invalid.
DLPI_EINVAL	An argument is invalid.
DLPI_ENOTEIDINVAL	The DLPI notification ID is invalid.
DLPI_FAILURE	The DLPI operation failed.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also dlpi_enabnotify(3DLPI), libdlpi(3LIB), attributes(5)

Name dlpi_enabmulti, dlpi_disabmulti – enable or disable DLPI multicast messages for an address

Synopsis cc [*flag*...] *file*... -ldlpi [*library*...]
#include <libdlpi.h>

int dlpi_enabmulti(dlpi_handle_t *dh*, const void **addrp*,
    size_t *addrlen*);

int dlpi_disabmulti(dlpi_handle_t *dh*, const void **addrp*,
    size_t *addrlen*);

Description The dlpi_enabmulti() function enables reception of messages destined to the multicast address pointed to by *addrp* on the DLPI link instance associated with DLPI handle *dh*. The DLPI link instance will pass up only those messages destined for enabled multicast addresses. This operation can be performed in any DLPI state of a handle.

The dlpi_disabmulti() function disables a specified multicast address pointed to by *addrp* on the DLPI link instance associated with DLPI handle *dh*. This operation can be performed in any DLPI state of a handle.

Return Values Upon success, DLPI_SUCCESS is returned. If DL_SYSERR is returned, errno contains the specific UNIX system error value. Otherwise, a DLPI error value defined in <sys/dlpi.h> or DLPI_EINHANDLE is returned.

Errors
DLPI_EBADMSG	Bad DLPI message
DLPI_EINHANDLE	Invalid DLPI handle
DLPI_EINVAL	Invalid argument
DLPI_ETIMEDOUT	DLPI operation timed out

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libdlpi(3LIB), attributes(5)

Name   dlpi_enabnotify – enable DLPI notification

Synopsis   cc [*flag*...] *file*... -ldlpi [*library*...]
#include <libdlpi.h>

int dlpi_enabnotify(dlpi_handle_t *dh*, uint_t *notes*,
    dlpi_notifyfunc_t **funcp*, void **arg*, dlpi_notifyid_t *id*);

typedef void dlpi_notifyfunc_t(dlpi_handle_t,
    dlpi_notifyinfo_t *, void *);

Description   The dlpi_enabnotify() function enables a notification callback for the set of events specified
in *notes*, which must be one or more (by a logical OR operation) of the DLPI notifications
documented in dlpi(7P). The callback function *funcp* is registered with the DLPI handle *dh*
and is invoked when *dh* receives notification for any of the specified event types. Upon
success, *id* contains the identifier associated with the registration.

Multiple event types can be registered for a callback function on the DLPI handle *dh*.
Similarly, the same event type can be registered multiple times on the same handle.

Once a callback has been registered, libdlpi will check for notification events on the DLPI
handle *dh*, when exchanging DLPI messages with the underlying DLPI link instance. The
dlpi_recv(3DLPI) function will always check for notification events, but other libdlpi
operations may also lead to an event callback being invoked. Although there may be no
expected data messages to be received, dlpi_recv() can be called, as shown below, with a null
buffer to force a check for pending events on the underlying DLPI link instance.

dlpi_recv(dh, NULL, NULL, NULL, NULL, 0, NULL);

When a notification event of interest occurs, the callback function is invoked with the
arguments *arg*, originally passed to dlpi_disabnotify(3DLPI), and *infop*, whose members
are described below.

uint_t *dni_note*	Notification event type.
uint_t *dni_speed*	Current speed, in kilobits per second, of the DLPI link. Valid only for DL_NOTE_SPEED.
uint_t *dni_size*	Current maximum message size, in bytes, that the DLPI link is able to accept for transmission. Valid only for DL_NOTE_SDU_SIZE.
uchar_t *dni_physaddrlen*	Link-layer physical address length, in bytes. Valid only for DL_NOTE_PHYS_ADDR.
uchar_t *dni_physaddr*[]	Link-layer physical address of DLPI link. Valid only for DL_NOTE_PHYS_ADDR.

The libdlpi library will allocate and free the dlpi_notifyinfo_t structure and the caller
must not allocate the structure or perform any operations that require its size to be known.

The callback is not allowed to block. This precludes calling dlpi_enabnotify() from a callback, but non-blocking libdlpi functions, including dlpi_disabnotify(), can be called.

Return Values Upon success, DLPI_SUCCESS is returned. If DL_SYSERR is returned, errno contains the specific UNIX system error value. Otherwise, a DLPI error value defined in <sys/dlpi.h> or an error value listed in the following section is returned.

Errors DLPI_EINHANDLE   A DLPI handle is invalid.

DLPI_EINVAL   An argument is invalid.

DLPI_ENOTEIDINVAL   The DLPI notification ID is invalid.

DLPI_ENOTENOTSUP   The DLPI notification is not supported by the link.

DLPI_ETIMEDOUT   The DLPI operation timed out.

DLPI_FAILURE   The DLPI operation failed.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also dlpi_disabnotify(3DLPI), dlpi_recv(3DLPI), libdlpi(3LIB), attributes(5), dlpi(7P)

Name dlpi_fd – get DLPI file descriptor

Synopsis cc [*flag* ...] *file* ... -ldlpi [*library* ...]
#include <libdlpi.h>

int *dlpi_fd*(dlpi_handle_t *dh*);

Description The dlpi_fd() function returns the integer file descriptor that can be used to directly operate
on the open DLPI stream associated with the DLPI handle *dh*. This file descriptor can be used
to perform non-DLPI operations that do not alter the state of the DLPI stream, such as waiting
for an event using poll(2), or pushing and configuring additional STREAMS modules, such as
pfmod(7M). If DLPI operations are directly performed on the file descriptor, or a STREAMS
module is pushed that alters the message-passing interface such that DLPI operations can no
longer be issued, future operations on *dh* might not behave as documented.

The returned file descriptor is managed by libdlpi(3LIB) and the descriptor must not be
closed.

Return Values The function returns the integer file descriptor associated with the DLPI handle *dh*. If *dh* is
invalid, -1 is returned.

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also poll(2), libdlpi(3LIB), attributes(5), dlpi(7P), pfmod(7M)

Name dlpi_get_physaddr – get physical address using DLPI

Synopsis cc [*flag*...] *file*... -ldlpi [*library*...]
#include <libdlpi.h>

int dlpi_get_physaddr(dlpi_handle_t *dh*, uint_t *type*,
    void *addrp*,size_t *addrlenp*);

Description The dlpi_get_physaddr() function gets a physical address from the DLPI link instance
associated with DLPI handle *dh*. The retrieved address depends upon *type*, which can be:

DL_FACT_PHYS_ADDR   Factory physical address

DL_CURR_PHYS_ADDR   Current physical address

The operation can be performed in any DLPI state of *dh*.

The caller must ensure that *addrp* is at least DLPI_PHYSADDR_MAX bytes in size and *addrlenp*
must contain the length of *addrp*. Upon success, *addrp* contains the specified physical address,
and *addrlenp* contains the physical address length. If a physical address is not available, *addrp*
is not filled in and *addrlenp* is set to zero.

Return Values Upon success, DLPI_SUCCESS is returned. If DL_SYSERR is returned, errno contains the specific
UNIX system error value. Otherwise, a DLPI error value defined in <sys/dlpi.h> or an error
value listed in the following section is returned.

Errors DLPI_EBADMSG   Bad DLPI message

DLPI_EINHANDLE   Invalid DLPI handle

DLPI_EINVAL   Invalid argument

DLPI_ETIMEDOUT   DLPI operation timed out

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also dlpi_set_physaddr(3DLPI), libdlpi(3LIB), attributes(5)

Name dlpi_iftype – convert a DLPI MAC type to a BSD socket interface type

Synopsis cc [*flag*...] *file*... -ldlpi [*library*...]
#include <libdlpi.h>

uint_t dlpi_iftype(uint_t *dlpitype*);

Description The dlpi_iftype() function converts a DLPI MAC type to a BSD socket interface type defined in <net/if_types.h>.

Return Values Upon success, the corresponding BSD socket interface type is returned. Otherwise, zero is returned.

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libdlpi(3LIB), attributes(5)

Name dlpi_info – get DLPI information

Synopsis cc [*flag* ...] *file* ... -ldlpi [*library* ...]
#include <libdlpi.h>

int *dlpi_info*(dlpi_handle_t *dh*, dlpi_info_t **infop*,
    uint_t *version*);

Description The dlpi_info() function provides DLPI information about the open DLPI link instance
associated with DLPI handle *dh*. DLPI information can be retrieved in any state of *dh*, but
some of the information might not be available if *dh* is in the DL_UNBOUND DLPI state. The
DLPI information received is copied into *infop*, which must point to a dlpi_info_t allocated
by the caller. The *version* argument specifies the version of the dlpi_info_t structure
expected by the caller. Callers can use the macro DLPI_INFO_VERSION to specify the default
version, which is currently 0. Callers can request version 1 of the dlpi_info_t structure by
defining DLPI_INFO_VERSION to 1 before including <libdlpi.h> and passing the defined
DLPI_INFO_VERSION value of 1 as the version argument. See the description of *di_linkname*
below for the difference between version 0 and 1.

The dlpi_info_t is a structure defined in <libdlpi.h> as follows:

```
typedef struct {
    uint_t              di_opts;
    uint_t              di_max_sdu;
    uint_t              di_min_sdu;
    uint_t              di_state;
    uchar_t             di_mactype;
    char                di_linkname[DLPI_LINKNAME_MAX];
    uchar_t             di_physaddr[DLPI_PHYSADDR_MAX];
    uchar_t             di_physaddrlen;
    uchar_t             di_bcastaddr[DLPI_PHYSADDR_MAX];
    uchar_t             di_bcastaddrlen;
    uint_t              di_sap;
    int                 di_timeout;
    dl_qos_cl_sel1_t    di_qos_sel;
    dl_qos_cl_range1_t  di_qos_range;
} dlpi_info_t;
```

di_opts
    Reserved for future dlpi_info_t expansion.

di_max_sdu
    Maximum message size, in bytes, that the DLPI link is able to accept for transmission. The
    value is guaranteed to be greater than or equal to *di_min_sdu*.

di_min_sdu
    Minimum message size, in bytes, that the DLPI link is able to accept for transmission. The
    value is guaranteed to be greater than or equal to one.

di_state
> Current DLPI state of *dh*; either `DL_UNBOUND` or `DL_IDLE`.

di_mactype
> MAC type supported by the DLPI link associated with *dh*. See `<sys/dlpi.h>` for the list of possible MAC types.

di_linkname
> Link name associated with DLPI handle *dh*. If the caller specifies the default *version* argument value of 0, the size of this field is `DLPI_LINKNAME_MAX`. If the caller defines `DLPI_INFO_VERSION` to value 1 before including `<libdlpi.h>`, the size of this field is MAXLINKNAMESPECIFIER. See dlpi(7P) for information on link names and the supported maximum length of DLPI link names.

di_physaddr
> Link-layer physical address of bound *dh*. If *dh* is in the `DL_UNBOUND` DLPI state, the contents of *di_physaddr* are unspecified.

di_physaddrlen
> Physical address length, in bytes. If *dh* is in the `DL_UNBOUND` DLPI state, *di_physaddrlen* is set to zero.

di_bcastaddr
> Link-layer broadcast address. If the *di_mactype* of the DLPI link does not support broadcast, the contents of *di_bcastaddr* are unspecified.

di_bcastaddrlen
> Link-layer broadcast address length, in bytes. If the *di_mactype* of the DLPI link does not support broadcast, *di_bcastaddrlen* is set to zero.

di_sap
> SAP currently bound to handle. If *dh* is in the `DL_UNBOUND` DLPI state, *di_sap* is set to zero.

di_timeout
> Current timeout value, in seconds, set on the `dlpi` handle.

di_qos_sel
> Current QOS parameters supported by the DLPI link instance associated with *dh*. Unsupported QOS parameters are set to `DL_UNKNOWN`.

di_qos_range
> Available range of QOS parameters supported by a DLPI link instance associated with the DLPI handle *dh*. Unsupported QOS range values are set to `DL_UNKNOWN`.

Return Values Upon success, `DLPI_SUCCESS` is returned. If `DL_SYSERR` is returned, `errno` contains the specific UNIX system error value. Otherwise, a DLPI error value defined in `<sys/dlpi.h>` or an error value listed in the following section is returned.

Errors DLPI_EBADMSG   Bad DLPI message

DLPI_EINHANDLE   Invalid DLPI handle

DLPI_EINVAL   Invalid argument

DLPI_EMODENOTSUP   Unsupported DLPI connection mode

DLPI_ETIMEDOUT   DLPI operation timed out

DLPI_EVERNOTSUP   Unsupported DLPI Version

DLPI_FAILURE   DLPI operation failed

Examples EXAMPLE 1   Get link-layer broadcast address

The following example shows how dlpi_info() can be used.

```
#include <libdlpi.h>

uchar_t *
get_bcastaddr(const char *linkname, uchar_t *baddrlenp)
{
        dlpi_handle_t   dh;
        dlpi_info_t     dlinfo;
        uchar_t         *baddr;

        if (dlpi_open(linkname, &dh, 0) != DLPI_SUCCESS)
                return (NULL);

        if (dlpi_info(dh, &dlinfo, 0) != DLPI_SUCCESS) {
                dlpi_close(dh);
                return (NULL);
        }
        dlpi_close(dh);

        *baddrlenp = dlinfo.di_bcastaddrlen;
        if ((baddr = malloc(*baddrlenp)) == NULL)
                return (NULL);

        return (memcpy(baddr, dlinfo.di_bcastaddr, *baddrlenp));
}
```

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also dlpi_bind(3DLPI), libdlpi(3LIB), attributes(5), dlpi(7P)

Name dlpi_linkname – get DLPI link name

Synopsis cc [*flag* ...] *file* ... -ldlpi [*library* ...]
#include <libdlpi.h>

const char **dlpi_linkname*(dlpi_handle_t *dh*);

Description The dlpi_linkname() function returns a pointer to the link name of the DLPI link instance associated with the DLPI handle *dh*.

The returned string is managed by libdlpi and must not be modified or freed by the caller.

Return Values Upon success, the function returns a pointer to the link name associated with the DLPI handle.

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libdlpi(3LIB), attributes(5)

Name dlpi_mactype – convert a DLPI MAC type to a string

Synopsis cc [*flag* ...] *file* ... -ldlpi [*library* ...]
#include <libdlpi.h>

const char **dlpi_mactype*(uint_t *mactype*);

Description The dlpi_mactype() function returns a pointer to a string that describes the specified
mactype. Possible MAC types are defined in <sys/dlpi.h>. The string is not dynamically
allocated and must not be freed by the caller.

Return Values Upon success, the function returns a pointer string that describes the MAC type. If *mactype* is
unknown, the string "Unknown MAC Type" is returned.

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libdlpi(3LIB), attributes(5)

Name dlpi_open – open DLPI link

Synopsis cc [*flag* ...] *file* ... -ldlpi [*library* ...]
#include <libdlpi.h>

int dlpi_open(const char **linkname*, dlpi_handle_t **dhp*,
    uint_t *flags*);

Description The dlpi_open() function creates an open instance of the DLPI Version 2 link named by
linkname and associates it with a dynamically-allocated dlpi_handle_t, which is returned to
the caller in *dhp* upon success. The DLPI handle is left in the DL_UNBOUND DLPI state after a
successful open of the DLPI link. The DLPI handles can only be used by one thread at a time,
but multiple handles can be used by multiple threads. This function can open both DL_STYLE1
and DL_STYLE2 DLPI links.

By default (if DLPI_DEVIPNET is not set in *flags*), the dlpi_open() function scans the /dev/net
and /dev directories for DLPI links, in order. Within each scanned directory, dlpi_open()
first looks for a matching DL_STYLE1 link, then for a matching DL_STYLE2 link. If *provider* is
considered the *linkname* with its trailing digits removed, a matching DL_STYLE1 link has a
filename of *linkname*, and a matching DL_STYLE2 link has a filename of *provider*. If a
DL_STYLE2 link is opened, dlpi_open() automatically performs the necessary DLPI
operations to place the DLPI link instance and the associated DLPI handle in the DL_UNBOUND
state. See dlpi(7P) for the definition of *linkname* and the maximum supported length of the
Solaris DLPI *linkname*.

If DLPI_DEVIPNET is set in *flags*, dlpi_open() opens the file *linkname* in /dev/ipnet as a
DL_STYLE1 DLPI device and does not look in any other directories.

The value of *flags* is constructed by a bitwise-inclusive-OR of the flags listed below, defined in
<libdlpi.h>.

DLPI_DEVIPNET   Specify that the named DLPI device is an IP observability device (see
                 ipnet(7D)), and dl_open() will open the device from the /dev/ipnet/
                 directory.

DLPI_IPNETINFO   This flag is applicable only when opening IP Observability devices (with
                 DLPI_DEVIPNET or by opening the /dev/lo0 device). This flag causes the
                 ipnet driver to prepend an ipnet header to each received IP packet. See
                 ipnet(7D) for the contents of this header.

DLPI_NATIVE   Enable DLPI native mode (see DLIOCNATIVE in dlpi(7P)) on a DLPI link
                 instance. Native mode persists until the DLPI handle is closed by
                 dlpi_close(3DLPI).

DLPI_PASSIVE   Enable DLPI passive mode (see DL_PASSIVE_REQ in dlpi(7P)) on a DLPI
                 link instance. Passive mode persists until the DLPI handle is closed by
                 dlpi_close(3DLPI).

DLPI_RAW   Enable DLPI raw mode (see DLIOCRAW in dlpi(7P)) on a DLPI link
                        instance. Raw mode persists until the DLPI handle is closed by
                        dlpi_close(3DLPI).

Each DLPI handle has an associated timeout value that is used as a timeout interval for certain
libdlpi operations. The default timeout value ensures that DLPI_ETIMEDOUT is returned from
a libdlpi operation only in the event that the DLPI link becomes unresponsive. The timeout
value can be changed with dlpi_set_timeout(3DLPI), although this should seldom be
necessary.

Return Values   Upon success, DLPI_SUCCESS is returned. If DL_SYSERR is returned, errno contains the specific
UNIX system error value. Otherwise, a DLPI error value defined in <sys/dlpi.h> or listed in
the following section is returned.

Errors   The dlpi_open() function will fail if:

DLPI_EBADLINK   Bad DLPI link

DLPI_EIPNETINFONOTSUP   The DLPI_IPNETINFO flag was set but the device opened does
                        not support the DLIOCIPNETINFO ioctl.

DLPI_ELINKNAMEINVAL   Invalid DLPI *linkname*

DLPI_ENOLINK   DLPI link does not exist

DLPI_ERAWNOTSUP   DLPI raw mode not supported

DLPI_ETIMEDOUT   DLPI operation timed out

DLPI_FAILURE   DLPI operation failed

Attributes   See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   dlpi_close(3DLPI), dlpi_set_timeout(3DLPI), libdlpi(3LIB), attributes(5), dlpi(7P),
ipnet(7D)

Name dlpi_promiscon, dlpi_promiscoff – enable or disable DLPI promiscuous mode

Synopsis cc [*flag*...] *file*... -ldlpi [*library*...]
#include <libdlpi.h>

int dlpi_promiscon(dlpi_handle_t *dh*, uint_t *level*);

int dlpi_promiscoff(dlpi_handle_t *dh*, uint_t *level*);

Description The dlpi_promiscon() function enables promiscuous mode on a DLPI link instance associated with DLPI handle *dh*, at the specified *level*. After enabling promiscuous mode, the caller will be able to receive all messages destined for the DLPI link instance at the specified *level*. This operation can be performed in any DLPI state of a handle.

The dlpi_promiscoff() function disables promiscuous mode on a DLPI link instance associated with DLPI handle *dh*, at the specified level. This operation can be performed in any DLPI state of a handle in which promiscuous mode is enabled at the specified *level*.

The *level* modes are:

DL_PROMISC_PHYS   Promiscuous mode at the physical level

DL_PROMISC_SAP   Promiscuous mode at the SAP level

DL_PROMISC_MULTI   Promiscuous mode for all multicast addresses

Return Values Upon success, DLPI_SUCCESS is returned. If DL_SYSERR is returned, errno contains the specific UNIX system error value. Otherwise, a DLPI error value defined in <sys/dlpi.h> or an error value listed in the following section is returned.

Errors DLPI_EBADMSG   Bad DLPI message

DLPI_EINHANDLE   Invalid DLPI handle

DLPI_EINVAL   Invalid argument

DLPI_ETIMEDOUT   DLPI operation timed out

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libdlpi(3LIB), attributes(5)

Name   dlpi_recv – receive a data message using DLPI

Synopsis   cc [*flag* ...] *file* ... -ldlpi [*library* ...]
#include <libdlpi.h>

int dlpi_recv(dlpi_handle_t *dh*, void *saddrp*,
    size_t * *saddrlenp*, void *msgbuf*, size_t *msglenp*,
    int *msec*, dlpi_recvinfo_t *recvp*);

Description   The dlpi_recv() function attempts to receive data messages over the DLPI link instance
associated with the DLPI handle *dh*. If *dh* is not in the DL_IDLE DLPI state, the attempt fails.
The caller must ensure that *msgbuf* is at least *msglenp* bytes in size. Upon success, *msgbuf*
contains the data message received, *msglenp* contains the number of bytes placed in *msgbuf*.

The caller must ensure that *saddrp* is at least DLPI_PHYSADDR_MAX bytes in size and *saddrlenp*
must contain the length of *saddrp*. Upon success, *saddrp* contains the address of the source
sending the data message and *saddrlenp* contains the source address length. If the caller is not
interested in the source address, both *saddrp* and *saddrlenp* can be left as NULL. If the source
address is not available, *saddrp* is not filled in and *saddrlenp* is set to zero.

The *dlpi_recvinfo_t* is a structure defined in <libdlpi.h> as follows:

```
typedef struct {
        uchar_t         dri_destaddr[DLPI_PHYSADDR_MAX];
        uchar_t         dri_destaddrlen;
        dlpi_addrtype_t dri_destaddrtype;
        size_t          dri_totmsglen;
} dlpi_recvinfo_t;
```

Upon success, if *recvp* is not set to NULL, *dri_destaddr* contains the destination address,
dri_destaddrlen contains the destination address length, and *dri_totmsglen* contains the total
length of the message received. If the destination address is unicast, *dri_destaddrtype* is set to
DLPI_ADDRTYPE_UNICAST. Otherwise, it is set to DLPI_ADDRTYPE_GROUP.

The values of *msglenp* and *dri_totmsglen* might vary when a message larger than the size of
msgbuf is received. In that case, the caller can use *dri_totmsglen* to determine the original total
length of the message.

If the handle is in raw mode, as described in dlpi_open(3DLPI), *msgbuf* starts with the
link-layer header. See dlpi(7P). The values of *saddrp*, *saddrlenp*, and all the members of
dlpi_recvinfo_t except *dri_totmsglen* are invalid because the address information is already
included in the link-layer header returned by *msgbuf*.

If no message is received within *msec* milliseconds, dlpi_recv() returns DLPI_ETIMEDOUT. If
msec is 0, dlpi_recv() does not block. If *msec* is -1, dlpi_recv() does block until a data
message is received.

Return Values Upon success, DLPI_SUCCESS is returned. If DL_SYSERR is returned, errno contains the specific UNIX system error value. Otherwise, a DLPI error value defined in <sys/dlpi.h> or an error value listed in the following section is returned.

Errors
DLPI_EBADMSG   Bad DLPI message

DLPI_EINHANDLE   Invalid DLPI handle

DLPI_EINVAL   Invalid argument

DLPI_ETIMEDOUT   DLPI operation timed out

DLPI_EUNAVAILSAP   Unavailable DLPI SAP

DLPI_FAILURE   DLPI operation failed

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also dlpi_bind(3DLPI), dlpi_open(3DLPI), libdlpi(3LIB), attributes(5), dlpi(7P)

Name dlpi_send – send a data message using DLPI

Synopsis cc [*flag*...] *file*... -ldlpi [*library*...]
#include <libdlpi.h>

int dlpi_send(dlpi_handle_t *dh*, const void **daddrp*,
    size_t *daddrlen*, const void **msgbuf*, size_t *msglen*,
    const dlpi_sendinfo_t **sendp*);

Description The dlpi_send() function attempts to send the contents of *msgbuf* over the DLPI link
instance associated with the DLPI handle *dh* to the destination address specified by *daddrp*.
The size of *msgbuf* and *daddrp* are provided by the *msglen* and *daddrlen* arguments,
respectively. The attempt will fail if *dh* is not in the DL_IDLE DLPI state, the address named by
daddrp is invalid, *daddrlen* is larger than DLPI_PHYSADDR_MAX, or *msglen* is outside the range
reported by dlpi_info(3DLPI).

If the sendp argument is NULL, data is sent using the bound SAP associated with *dh* (see
dlpi_bind(3DLPI)) and with default priority. Otherwise, *sendp* must point to a
dlpi_sendinfo_t structure defined in <libdlpi.h> as follows:

```
typedef struct {
        uint_t          dsi_sap;
        dl_priority_t   dsi_prio;
} dlpi_sendinfo_t;
```

The *dsi_sap* value indicates the SAP to use for the message and the *dsi_prio* argument
indicates the priority. The priority range spans from 0 to 100, with 0 being the highest priority.
If one wishes to only alter the SAP or priority (but not both), the current SAP can be retrieved
using dlpi_info(3DLPI), and the default priority can be specified by using the
DL_QOS_DONT_CARE constant.

If the handle is in raw mode (see DLPI_RAW in dlpi_open(3DLPI)), *msgbuf* must start with the
link-layer header (see dlpi(7P)). In raw mode, the contents of *daddrp* and *sendp* are ignored,
as they are already specified by the link-layer header in *msgbuf*.

If *msgbuf* is accepted for delivery, no error is returned. However, because only
unacknowledged connectionless service (DL_CLDLS) is currently supported, a successful
return does not guarantee that the data will be successfully delivered to *daddrp*.

Return Values Upon success, DLPI_SUCCESS is returned. If DL_SYSERR is returned, errno contains the specific
UNIX system error value. Otherwise, a DLPI error value defined in <sys/dlpi.h> or an error
value listed in the following section is returned.

Errors DLPI_EINHANDLE   Invalid DLPI handle

DLPI_EINVAL   Invalid argument

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also dlpi_bind(3DLPI), dlpi_info(3DLPI), dlpi_open(3DLPI), libdlpi(3LIB), attributes(5), dlpi(7P)

Name dlpi_set_physaddr – set physical address using DLPI

Synopsis cc [*flag*...] *file*... -ldlpi [*library*...]
#include <libdlpi.h>

int dlpi_set_physaddr(dlpi_handle_t *dh*, uint_t *type*,
    const void *addrp*, size_t *addrlen*);

Description The dlpi_set_physaddr() function sets the physical address via DLPI handle *dh* associated
with the DLPI link instance. Upon success, the physical address is set to *addrp* with a length of
addrlen bytes.

In this release, *type* must be set to DL_CURR_PHYS_ADDR, which sets the current physical
address.

Return Values Upon success, DLPI_SUCCESS is returned. If DL_SYSERR is returned, errno contains the specific
UNIX system error value. Otherwise, a DLPI error value defined in <sys/dlpi.h> or an error
value listed in the following section is returned.

Errors
DLPI_EBADMSG   Bad DLPI message

DLPI_EINHANDLE   Invalid DLPI handle

DLPI_EINVAL   Invalid argument

DLPI_ETIMEDOUT   DLPI operation timed out

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also dlpi_get_physaddr(3DLPI), libdlpi(3LIB), attributes(5)

Name dlpi_set_timeout – set DLPI handle timeout interval

Synopsis cc [*flag* ...] *file* ... -ldlpi [*library* ...]
#include <libdlpi.h>

int *dlpi_set_timeout*(dlpi_handle_t *dh*, int *sec*);

Description The dlpi_set_timeout() function sets the timeout interval to *sec* seconds on DLPI handle
dh. This timeout is used by libdlpi(3LIB) functions that require explicit acknowledgment
from the associated DLPI link, and bounds the number of seconds that a function will wait for
an acknowledgment before returning DLPI_ETIMEDOUT. Except for dlpi_recv(3DLPI), which
has a *timeout* argument, any function that is documented to return DLPI_ETIMEDOUT can take
up to the *timeout* interval to complete.

Callers that do not require an upper bound on timeouts are strongly encouraged to never call
dlpi_set_timeout(), and allow libdlpi to use its default *timeout* value. The default *timeout*
value is intended to ensure that DLPI_ETIMEDOUT will only be returned if the DLPI link has
truly become unresponsive. The default *timeout* value is intended to ensure that
DLPI_ETIMEDOUT will be returned only if the DLPI link has truly become unresponsive.

Callers that do require an explicit upper bound can specify that value at any time by calling
dlpi_set_timeout(). However, note that values less than 5 seconds may trigger spurious
failures on certain DLPI links and systems under high load, and thus are discouraged.
Attempts to set the *timeout* value to less than 1 second will fail.

If *sec* is set to DLPI_DEF_TIMEOUT, the default *timeout* value is restored.

Return Values Upon success, DLPI_SUCCESS is returned. Otherwise, a DLPI error value is returned.

Errors DLPI_EINHANDLE   Invalid DLPI handle

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libdlpi(3LIB), attributes(5)

Name dlpi_strerror – get DLPI error message

Synopsis cc [*flag*...] *file*... -ldlpi [*library*...]
#include <libdlpi.h>

const char *dlpi_strerror(int *err*);

Description The dlpi_strerror() function maps the error code in *err* into an error message string and returns a pointer to that string.

If *err* is DL_SYSERR, a string that describes the current value of errno is returned. Otherwise, if *err* corresponds to an error code listed in <libdlpi.h> or <sys/dlpi.h>, a string which describes that error is returned.

The string is not dynamically allocated and must not be freed by the caller.

Return Values Upon success, the function returns a pointer to the error message string. If the error code is unknown, the string "Unknown DLPI error" is returned.

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libdlpi(3LIB), attributes(5)

Name   dlpi_unbind – unbind DLPI handle

Synopsis   cc [*flag* ...] *file* ... -ldlpi [*library* ...]
#include <libdlpi.h>

int dlpi_unbind(dlpi_handle_t *dh*);

Description   The dlpi_unbind() function unbinds to bind the DLPI handle *dh* from the bound SAP. The handle must be in the DL_IDLE DLPI state and upon success, the handle transitions to the DL_UNBOUND state.

Upon success, the caller will no longer be able to send or receive data using the DLPI link associated with *dh*.

Return Values   Upon success, DLPI_SUCCESS is returned. If DL_SYSERR is returned, errno contains the specific UNIX system error value. Otherwise, a DLPI error value defined in <sys/dlpi.h> or an error value DLPI_ETIMEDOUT will be returned.

Errors   DLPI_EBADMSG   Bad DLPI message

DLPI_EINHANDLE   Invalid DLPI handle

DLPI_ETIMEDOUT   DLPI operation timed out

Attributes   See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   dlpi_bind(3DLPI), libdlpi(3LIB), attributes(5)

Name dlpi_walk – traverse DLPI links

Synopsis cc [*flag*...] *file*... -ldlpi [*library*...]
#include <libdlpi.h>

void dlpi_walk(dlpi_walkfunc_t *fn, void *arg, uint_t *flags*);

typedef boolean_t dlpi_walkfunc_t(const char *name, void *arg);

Parameters *fn* Function to invoke for each link. Arguments are:

name The name of the DLPI interface.

arg The *arg* parameter passed in to dlpi_walk().

arg An opaque argument that is passed transparently to the user-supplied *fn*()
function.

flags This parameter is reserved for future use. The caller should pass in 0.

Description The dlpi_walk() function visits all DLPI links in the current zone. The walk does not visit
DLPI links in the non-global zones when called from the global zone. For each link visited, the
user-supplied *fn*() function is invoked. The walk terminates either when all links have been
visited or when *fn*() returns B_TRUE.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libdlpi(3LIB), attributes(5)

Name   DNSServiceBrowse – browse service instances with DNS

Synopsis   cc [*flag* ...] *file* ... -ldns_sd [*library* ...]
#include <dns_sd.h>

DNSServiceErrorType *DNSServiceBrowse*(DNSServiceRef **sdRef*,
    DNSServiceFlags *flags*, uint32_t *interfaceIndex*,
    const char **regtype*, const char **domain*,
    DNSServiceServiceBrowseReply *callBack*, void **context*);

typedef *void*(*DNSServiceBrowseReply)(DNSServiceRef *sdRef*,
    DNSServiceFlags *flags*, uint32_t *interfaceIndex*,
    DNSServiceErrorType *errorCode*, const char **serviceName*,
    const char **regtype*, const char **replyDomain*,
    void **context*);

Description   The DNSServiceBrowse() function is used to browse for service instances of a particular
service and protocol type. The *sdRef* argument points to an uninitialized *DNSServiceRef*. If the
call to DNSServiceBrowse succeeds *sdRef* is initialized. The *flags* argument to
DNSServiceBrowse() is currently unused and reserved for future use. A nonzero value to
interfaceIndex indicates DNSServiceBrowse() should do a browse on all interfaces. Most
applications will use an *interfaceIndex* value of 0 to perform a browse on all interfaces. See the
section "Constants for specifying an interface index" in <dns_sd.h> for more details.

The callback function is invoked for every service instance found matching the service type
and protocol. The *callback* argument points to a function of type DNSServiceBrowseReply
listed above. The DNSServiceBrowse() call returns browse results asynchronously. The
service browse call can be terminated by applications with a call to
DNSServiceRefDeallocate().

The *regtype* parameter is used to specify the service type and protocol (e.g. _ftp._tcp). The
protocol type must be TCP or UDP. The *domain* argument to DNSServiceBrowse() specifies the
domain on which to browse for services. Most applications will not specify a domain and will
perform a browse on the default domain(s). The *context* argument can be NULL and points to a
value passed to the callback function.

The *sdRef* argument passed to the callback function is initialized by DNSServiceBrowse() call.
The possible values passed to *flags* in the callback function are:
kDNSServiceFlagsMoreComing and kDNSServiceFlagsAdd. The
kDNSServiceFlagsMoreComing value indicates to a callback that at least one more result is
immediately available. The kDNSServiceFlagsAdd value indicates that a service instance was
found. The *errorCode* argument will be kDNSServiceErr_NoError on success. Otherwise,
failure is indicated. The discovered service name is returned to the *callback* via the
serviceName argument. The *regtype* argument in the callback holds the service type of the
found service instance. The discovered service type can be different from the requested service
type in the browse request when the discovered service type has subtypes. The domain
argument to the callback holds the domain name of the discovered service instance. The

service type and the domain name must be stored and passed along with the service name to resolve a service instance using DNSServiceResolve().

Return Values   The DNSServiceBrowse function returns kDNSServiceErr_NoError on success. Otherwise, an error code defined in <dns_sd.h> is returned to indicate an error has occurred. When an error is returned by DNSServiceBrowse, the callback function is not invoked and the *DNSServiceRef* argument is not initialized.

Attributes   See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   DNSServiceRefDeallocate(3DNS_SD), DNSServiceResolve(3DNS_SD), attributes(5)

Name DNSServiceConstructFullName – construct full name

Synopsis cc [*flag* ...] *file* ... -ldns_sd [*library* ...]
#include <dns_sd.h>

int *DNSServiceConstructFullName* (char **fullname*,
const char **service*, const char **regtype*, const char **domain*);

Description The DNSServiceConstructFullName() concatenates a three-part domain name that consists of a service name, service type, and domain name into a fully escaped full domain name.

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also attributes(5)

Name   DNSServiceCreateConnection, DNSServiceRegisterRecord, DNSServiceAddRecord, DNSServiceUpdateRecord, DNSServiceRemoveRecord – registering multiple records

Synopsis   cc [*flag* ...] *file* ... -ldns_sd [*library* ...]
#include <dns_sd.h>

DNSServiceErrorType *DNSServiceCreateConnection*(DNSServiceRef *sdRef*);

DNSServiceErrorType *DNSServiceRegisterRecord*(DNSServiceRef *sdRef*,
    DNSRecordRef *RecordRef*, DNSServiceFlags *flags*,
    uint32_t *interfaceIndex*, const char *fullname*,
    uint16_t *rrtype*, uint16_t *rrclass*, uint16_t *rdlen*,
    const void *rdata*, uint32_t *ttl*,
    DNSServiceServiceRegisterRecordReply *callBack*,
    void *context*);

typedef *void*(*DNSServiceRegisterRecordReply)(DNSServiceRef *sdRef*,
    DNSServiceRecordRef *RecordRef*, DNSServiceFlags *flags*,
    DNSServiceErrorType *errorCode*, void *context*);

DNSServiceErrorType *DNSServiceAddRecord*(DNSServiceRef *sdRef*,
    DNSRecordRef *RecordRef*, DNSServiceFlags *flags*,
    uint16_t *rrtype*, uint16_t *rdlen*, const void *rdata*,
    uint32_t *ttl*);

DNSServiceErrorType *DNSServiceUpdateRecord*(DNSServiceRef *sdRef*,
    DNSRecordRef *RecordRef*, DNSServiceFlags *flags*,
    uint16_t *rdlen*, const void *rdata*,
    uint32_t *ttl*);

DNSServiceErrorType *DNSServiceRemoveRecord*(DNSServiceRef *sdRef*,
    DNSRecordRef *RecordRef*, DNSServiceFlags *flags*);

Description   The DNSServiceCreateConnection() function allows the creation of a connection to the mDNS daemon in order to register multiple individual records.

The DNSServiceRegisterRecord() function uses the connection created by DNSServiceCreateConnection() to register a record. Name conflicts that occur from this function should be handled by the client in the callback.

The DNSServiceAddRecord() call adds a DNS record to a registered service. The name of the record is the same as registered service name. The *RecordRef* argument to DNSServiceAddRecord() points to an uninitialized DNSRecordRef. After successful completion of DNSServiceAddRecord(), the DNS record can be updated or deregistered by passing the DNSRecordRef initialized by DNSServiceAddRecord() to DNSServiceUpdateRecord() or to DNSServiceRemoveRecord().

The DNSServiceUpdateRecords() call updates the DNS record of the registered service. The DNS record must be the primary resource record registered using DNSServiceRegister() or a record added to a registered service using DNSServiceAddRecord() or an individual record added via DNSServiceRegisterRecord().

The DNSServiceRemoveRecord() call removes a record that was added using DNSServiceAddRecord() or DNSServiceRegisterRecord().

The *sdRef* argument points to DNSServiceRef initialized from a call to DNSServiceRegister(). If the *sdRef* argument is passed to DNSServiceRefDeallocate() and the service is deregistered DNS records added via DNSServiceAddRecord() are invalidated and cannot be further used. The *flags* argument is currently ignored and reserved for future use. The *rrtype* parameter value indicates the type of the DNS record. Suitable values for the *rrtype* parameter are defined in <dns_sd.h>: kDNSServiceType_TXT, for example. The *rdata* argument points to the raw rdata to be contained in the resource record. The *ttl* value indicates the time to live of the resource record in seconds. A *ttl* value of 0 should be passed to use a default value.

Return Values The DNSServiceCreateConnection function returns kDNSServiceErr_NoError on success. Otherwise, an error code defined in <dns_sd.h> is returned to indicate the specific failure that occurred.

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also DNSServiceRefDeallocate(3DNS_SD), DNSServiceRegister(3DNS_SD), attributes(5)

Name DNSServiceEnumerateDomains – enumerate recommended domains

Synopsis cc [*flag* ...] *file* ... -ldns_sd [*library* ...]
#include <dns_sd.h>

DNSServiceErrorType *DNSServiceEnumerateDomains*(DNSServiceRef *sdRef*,
    DNSServiceFlags *flags*, uint32_t *interfaceIndex*,
    DNSServiceDomainEnumReply *callBack*, void *context*);

typedef *void*(*DNSServiceDomainEnumReply)(DNSServiceRef *sdRef*,
    DNSServiceFlags *flags*, uint31_t *interfaceIndex*,
    DNSServiceErrorType *errorCode*, const char *replyDomain*,
    void *context*);

Description The DNSServiceEnumerateDomains() function allows applications to determine recommended browsing and registration domains for performing service discovery DNS queries. The *callback* argument points to a function to be called to return results or if the asynchronous call to DNSServiceEnumerateDomains() fails. The callback function should point to a function of type DNSServiceDomainEnumReply listed above.

A pointer to an uninitialized DNSServiceRef, *sdRef* must be passed to DNSServiceEnumerateDomains(). If the call succeeds, *sdRef* is initialized and kDNSServiceErr_NoError is returned. The enumeration call runs indefinitely until the client terminates the call. The enumeration call must be terminated by passing the DNSServiceRef initialized by the enumeration call to DNSServiceRefDeallocate() when no more domains are to be found.

The value of flags is constructed by a bitwise-inclusive-OR of the *flags* used in DNSService functions and defined in <dns_sd.h>. Possible values for *flags* to the DNSServiceEnumerateDomains() call are: kDNSServiceFlagsBrowseDomains and kDNSServiceFlagsRegistrationDomains. The kDNSServiceFlagsBrowseDomains value is passed to enumerate domains recommended for browsing. The kDNSServiceFlagsRegistrationDomains value is passed to enumerate domains recommended for registration. Possible values of flags returned in the callback function are: kDNSServiceFlagsMoreComing, kDNSServiceFlagsAdd, and kDNSServiceFlagsDefault.

The *interfaceIndex* parameter to the enumeration call specifies the interface index searched for domains. Most applications pass 0 to enumerate domains on all interfaces. See the section "Constants for specifying an interface index" in <dns_sd.h> for more details. The context parameter can be NULL and is passed to the enumeration callback function. The *interfaceIndex* value passed to the callback specifies the interface on which the domain exists.

Return Values The DNSServiceEnumerateDomains() function returns kDNSServiceErr_NoError on success. Otherwise, the function returns an error code defined in <dns_sd.h>. The callback is not invoked on error and the *DNSServiceRef* that is passed is not initialized. Upon a successful call to DNSServiceEnumerateDomains(), subsequent asynchronous errors are delivered to the callback.

Attributes   See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   DNSServiceRefDeallocate(3DNS_SD), attributes(5)

Name DNSServiceProcessResult – process results and invoke callback

Synopsis cc [*flag* ...] *file* ... -ldns_sd [*library* ...]
#include <dns_sd.h>

DNSServiceErrorType DNSServiceProcessResult (DNSServiceRef *sdRef*);

Description The `DNSServiceProcessResult()` call reads the returned results from the mDNS daemon and invokes the specified application callback. The *sdRef* points to a `DNSServiceRef` initialized by any of the `DNSService` calls that take a *callback* parameter. The `DNSServiceProcessResult()` call blocks until data is received from the mDNS daemon. The application is responsible for ensuring that `DNSServiceProcessResult()` is called whenever there is a reply from the daemon. The daemon may terminate its connection with a client that does not process the daemon's responses.

Return Values The `DNSServiceProcessResult()` call returns `kDNSServiceErr_NoError` on success. Otherwise, an error code defined in `<dns_sd.h>` is returned to indicate the specific failure that has occurred.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also DNSServiceBrowse(3DNS_SD), DNSServiceRegister(3DNS_SD), attributes(5)

Name DNSServiceQueryRecord – query records from DNS

Synopsis cc [*flag* ...] *file* ... -ldns_sd [*library* ...]
#include <dns_sd.h>

DNSServiceErrorType *DNSServiceQueryRecord*(DNSServiceRef **sdRef*,
    DNSServiceFlags *flags*, uint32_t *interfaceIndex*, const char **fullname*,
    uint16_t *rrtype*, uint16_t *rrclass*,
    DNSServiceServiceQueryRecordReply *callBack*, void **context*);

typedef *void*(*DNSServiceQueryRecordReply)(DNSServiceRef *DNSServiceRef*,
    DNSServiceFlags *flags*, uint32_t *interfaceIndex*,
    DNSServiceErrorType *errorCode*, const char **fullname*,
    uint16_t *rrtype*, uint16_t *rrclass*, uint16_t *rdlen*,
    const void **rdata*, uint32_t *ttl*, void **context*);

Description The DNSServiceQueryRecord() function is used to query the daemon for any DNS resource
record type. The *callback* argument to DNSServiceQueryRecord() points to a function of type
DNSServiceQueryRecordReply() listed above. The *sdRef* parameter in
DNSServiceQueryRecord() points to an uninitialized *DNSServiceRef*. The
DNSServiceQueryRecord() function returns kDNSServiceErr_NoError and initializes *sdRef*
on success. The query runs indefinitely until the client terminates by passing the initialized
sdRef from the query call to DNSServiceRefDeallocate().

The flag kDNSServiceFlagsLongLivedQuery should be passed in the *flags* argument of
DNSServiceQueryRecord() to create a "long-lived" unicast query in a non-local domain. This
flag has no effect on link local multicast queries. Without this flag, unicast queries will be
one-shot and only the results that are available at the time of the query will be returned. With
long-lived queries, add or remove events that are available after the first call generate
callbacks. The *interfaceIndex* argument specifies the interface on which to issue the query.
Most applications will pass a 0 as the *interfaceIndex* to make the query on all interfaces. See the
section "Constants for specifying an interface index" in <dns_sd.h>. The *fullname* argument
indicates the full domain name of the resource record to be queried. The *rrtype* argument
indicates the resource record type: kDNSServiceType_PTR, for example. The *rrclass* argument
holds the class of the resource record to be queried (kDNSServiceClass_IN). The *context*
argument can be NULL and points to a value passed to the callback function.

The *sdRef* argument passed to the callback function is initialized by the call to
DNSServiceQueryRecord(). Possible values for the *flags* parameter to the callback function
are kDNSServiceFlagsMoreComing and kDNSServiceFlagsAdd. The
kDNSServiceFlagsMoreComing value is set to indicate that additional results are immediately
available. The kDNSServiceFlagsAdd value indicates that the results returned to the callback
function are for a valid DNS record. If kDNSServiceFlagsAdd is not set, the results returned are
for a delete event. The *errorCode* passed to the callback is kDNSServiceErr_NoError on
success. Otherwise, failure is indicated and other parameter values are undefined. The
fullname parameter indicates the full domain name of the resource record . The *rrtype*
indicates the resource record type. The *rrclass* indicates the class of the DNS resource record.

The *rdlen* parameter indicates the length of the resource record *rdata* in bytes. The *rdata* parameter points to raw rdata of the resource record. The *ttl* parameter indicates the time to live of the resource record in seconds. The *context* parameter points to the value passed by the application in the context argument to the DNSServiceQueryRecord() call.

Return Values The DNSServiceQueryRecord function returns kDNSServiceErr_NoError on success. Otherwise, an error code defined in <dns_sd.h> is returned to indicate the specific failure that occurred.

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also DNSServiceBrowse(3DNS_SD), DNSServiceRegister(3DNS_SD), DNSServiceResolve(3DNS_SD), attributes(5)

Name DNSServiceReconfirmRecord – verify DNS record

Synopsis cc [*flag* ...] *file* ... -ldns_sd [*library* ...]
#include <dns_sd.h>

void DNSServiceRefSockFD (DNSServiceFlags *flags*, uint32_t *interfaceIndex*,
    const char **fullname*, uint16_t *rrtype*, uint16_t *rrclass*,
    uint16_t *rrlen* const void **rdata*);

Description The DNSServiceReconfirmRecord() function allows callers to verify whether a DNS record is valid. If an invalid record is found in the cache, the daemon flushes the record from the cache and from the cache of other daemons on the network.

Return Values The DNSServiceReconfirmRecord() function returns kDNSServiceErr_NoError on success. Otherwise, an error code defined in <dns_sd.h> is returned to indicate the specific failure that occurred.

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also DNSServiceBrowse(3DNS_SD), DNSServiceQueryRecord(3DNS_SD), DNSServiceRegister(3DNS_SD), DNSServiceResolve(3DNS_SD), attributes(5)

Name DNSServiceRefDeallocate – close connection

Synopsis cc [*flag* ...] *file* ... -ldns_sd [*library* ...]
#include <dns_sd.h>

void DNSServiceRefDeallocate (DNSServiceRef *sdRef*);

Description The DNSServiceRefDeallocate() call terminates connection to the mDNS daemon. Any services and resource records registered with the DNSServiceRef are de-registered. Any browse or resolve queries initiated using the DNSServiceRef are also terminated.

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also DNSServiceBrowse(3DNS_SD), DNSServiceRegister(3DNS_SD), attributes(5)

Name DNSServiceRefSockFD – access underlying UNIX domain socket descriptor

Synopsis cc [*flag* ...] *file* ... -ldns_sd [*library* ...]
#include <dns_sd.h>

DNSServiceRefSockFD(DNSServiceRef *sdRef*);

Description Access the underlying UNIX domain socket from the initialized DNSServiceRef returned from DNS Service calls. Applications should only access the underlying UNIX domain socket to poll for results from the mDNS daemon. Applications should not directly read or write to the socket. When results are available, applications should call DNSServiceProcessResult(). The application is responsible for processing the data on the socket in a timely fashion. The daemon can terminate its connection with a client that does not clear its socket buffer.

Return Values The underlying UNIX domain socket descriptor of the DNSServiceRef or -1 is returned in case of error.

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also DNSServiceBrowse(3DNS_SD), DNSServiceRegister(3DNS_SD), attributes(5)

Name DNSServiceRegister – register service with DNS

Synopsis cc [*flag* ...] *file* ... -ldns_sd [*library* ...]
#include <dns_sd.h>

DNSServiceErrorType *DNSServiceRegister*(DNSServiceRef *sdRef*,
    DNSServiceFlags *flags*, uint32_t *interfaceIndex*,
    const char *name*, const char *regtype*,
    const char *domain*, const char *host*,
    uint16_t *port*, uint16_t *txtLen*, const void *txtRecord*
    DNSServiceServiceRegisterReply *callBack*
    void *context*);

typedef *void*(*DNSServiceRegisterReply)(DNSServiceRef *sdRef*,
    DNSServiceFlags *flags*, DNSServiceErrorType *errorCode*,
    const char *name*, const char *regtype*,
    const char *domain*, void *context*);

Description The DNSServiceRegister function is used by clients to advertise a service that uses DNS. The service is registered with multicast DNS if the domain name is .local or the interface requested is local only. Otherwise, the service registration is attempted with the unicast DNS server. The callback argument should point to a function of type DNSServiceRegisterReply listed above.

The *sdRef* parameter points to an uninitialized DNSServiceRef instance. If the DNSServiceRegister() call succeeds, *sdRef* is initialized and kDNSServiceErr_NoError is returned. The service registration remains active until the client terminates the registration by passing the initialized *sdRef* to DNSServiceRefDeallocate(). The *interfaceIndex* when non-zero specifies the interface on which the service should be registered. Most applications pass 0 to register the service on all interfaces. See the section "Constants for specifying an interface index" in <dns_sd.h> for more details. The *flags* parameter determines the renaming behavior on a service name conflict. Most applications pass 0 to allow auto-rename of the service name in case of a name conflict. Applications can pass the flag kDNSServiceFlagsNoAutoRename defined in <dns_sd.h> to disable auto-rename.

The *regtype* indicates the service type followed by the protocol, separated by a dot, for example "_ftp._tcp.". The service type must be an underscore that is followed by 1 to 14 characters that can be letters, digits, or hyphens. The transport protocol must be _tcp or _udp. New service types should be registered at http://www.dns-sd.org/ServiceTypes.html. The *domain* parameter specifies the domain on which a service is advertised. Most applications leave the *domain* parameter NULL to register the service in default domains. The *host* parameter specifies the SRV target host name. Most applications do not specify the host parameter value. Instead, the default host name of the machine is used. The port value on which the service accepts connections must be passed in network byte order. A value of 0 for a port is passed to register *placeholder* services. Placeholder services are not found when browsing, but other clients cannot register with the same name as the placeholder service.

The *txtLen* parameter specifies the length of the passed *txtRecord* in bytes. The value must be zero if the *txtRecord* passed is NULL. The *txtRecord* points to the TXT record rdata. A non-NULL *txtRecord* must be a properly formatted DNSTXT record. For more details see the DNSServiceRegister call defined in <dns_sd.h>. The callback argument points to a function to be called when registration completes or when the call asynchronously fails. The client can pass NULL for the callback and not be notified of the registration results or asynchronous errors. The client may not pass the NoAutoRename flag if the callback is NULL. The client can unregister the service at any time via DNSServiceRefDeallocate().

The callback function argument *sdRef* is initialized by DNSServiceRegister(). The *flags* argument in the callback function is currently unused and reserved for future use. The error code returned to the callback is kDNSServiceErr_NoError on success. Otherwise, an error code defined in <dns_sd.h> is returned to indicate an error condition such as a name conflict in kDNSServiceFlagsNoAutoRename mode. The *name* argument holds the registered service name and the *regtype* argument is the registered service type passed to DNSServiceRegister(). The domain argument returned in the callback indicates the domain on which the service was registered.

Return Values   The DNSServiceRegister function returns kDNSServiceErr_NoError on success. Otherwise, an error code defined in <dns_sd.h> is returned. Upon registration, any subsequent asynchronous errors are delivered to the callback.

Attributes   See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   DNSServiceRefDeallocate(3DNS_SD), attributes(5)

Name DNSServiceResolve – resolve service instances with DNS

Synopsis cc [*flag* ...] *file* ... -ldns_sd [*library* ...]
#include <dns_sd.h>

DNSServiceErrorType *DNSServiceResolve*(DNSServiceRef *sdRef*,
    DNSServiceFlags *flags*, uint32_t *interfaceIndex*, const char *name*,
    const char *regtype*, const char *domain*,
    DNSServiceServiceResolveReply *callBack*, void *context*);

typedef *void*(*DNSServiceResolveReply)(DNSServiceRef *sdRef*,
    DNSServiceFlags *flags*, uint32_t *interfaceIndex*,
    DNSServiceErrorType *errorCode*, const char *fullname*,
    const char *hosttarget*, uint16_t *port*, uint16_t *txtLen*,
    const char *txtRecord*, void *context*);

Description The DNSServiceResolve() function is used to resolve a service name returned by
DNSServiceBrowse() to host IP address, port number, and TXT record. The
DNSServiceResolve() function returns results asynchronously. A DNSServiceResolve() call
to resolve service name can be ended by calling DNSServiceRefDeallocate(). The *callback*
argument points to a function of type *DNSServiceResolveReply* as listed above. The callback
function is invoked on finding a result or when the asynch resolve call fails. The *sdRef*
argument to DNSServiceResolve() points to an uninitialized *DNSServiceRef*. If the call to
DNSServiceResolve() succeeds, *sdRef* is initialized and kDNSServiceErr_NoError is
returned.

The *flags* argument to DNSServiceResolve() is currently unused and reserved for future use.
The *interfaceIndex* argument indicates the interface on which to resolve the service. If the
DNSServiceResolve() call is the result of an earlier DNSServiceBrowse() operation, pass the
interfaceIndex to perform a resolve on all interfaces. See the section "Constants for specifying
an interface index" in <dns_sd.h> for more details. The *name* parameter is the service
instance name to be resolved, as returned from a DNSServiceBrowse() call. The *regtype* holds
the service type and the *domain* parameter indicates the domain in which the service instance
was found. The *context* parameter points to a value that is passed to the callback function.

The *sdRef* argument passed to the callback function is initialized by DNSServiceResolve()
call. The *flags* parameter in the callback function is currently unused and reserved for future
use. The *errorCode* parameter is kDNSServiceErr_NoError on success. Otherwise, it will hold
the error defined in <dns_sd.h> and other parameters are undefined when *errorCode* is
nonzero. The *fullname* parameter in the callback holds the full service domain name in the
format *<servicename>.<protocol>.<domain>*. The full service domain name is escaped to
follow standard DNS rules. The *hosttarget* parameter holds the target hostname of the machine
providing the service. The *port* parameter indicates the port in network byte order on which
the service accepts connections. The *txtLen* and *txtRecord* parameters hold the length and the
TXT record of the service's primary TXT record. The *context* parameter points to the value that
was passed as context to the DNSServiceResolve() call.

Return Values The DNSServiceResolve function returns kDNSServiceErr_NoError on success. Otherwise, an error code defined in <dns_sd.h> is returned to indicate an error has occurred. When an error is returned by DNSServiceResolve, the callback function is not invoked and the *DNSServiceRef* argument is not initialized.

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also DNSServiceBrowse(3DNS_SD), DNSServiceRefDeallocate(3DNS_SD), attributes(5)

Name endhostent, gethostbyaddr, gethostbyname, gethostent, sethostent – network host database functions

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <netdb.h>
extern int h_errno;

void endhostent(void)

struct hostent *gethostbyaddr(const void **addr*, socklen_t *len*, int *type*);

struct hostent *gethostbyname(const char **name*);

struct hostent *gethostent(void)

void sethostent(int *stayopen*);

Description The gethostent(), gethostbyaddr(), and gethostbyname() functions each return a pointer to a hostent structure, the members of which contain the fields of an entry in the network host database.

The gethostent() function reads the next entry of the database, opening a connection to the database if necessary.

The gethostbyaddr() function searches the database and finds an entry which matches the address family specified by the type argument and which matches the address pointed to by the *addr* argument, opening a connection to the database if necessary. The *addr* argument is a pointer to the binary-format (that is, not null-terminated) address in network byte order, whose length is specified by the *len* argument. The datatype of the address depends on the address family. For an address of type AF_INET, this is an in_addr structure, defined in <netinet/in.h>. For an address of type AF_INET6, there is an in6_addr structure defined in <netinet/in.h>.

The gethostbyname() function searches the database and finds an entry which matches the host name specified by the *name* argument, opening a connection to the database if necessary. If *name* is an alias for a valid host name, the function returns information about the host name to which the alias refers, and *name* is included in the list of aliases returned.

The sethostent() function opens a connection to the network host database, and sets the position of the next entry to the first entry. If the *stayopen* argument is non-zero, the connection to the host database will not be closed after each call to gethostent() (either directly, or indirectly through one of the other gethost*( ) functions).

The endhostent() function closes the connection to the database.

Usage The gethostent(), gethostbyaddr(), and gethostbyname() functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

These functions are generally used with the Internet address family.

Return Values On successful completion, gethostbyaddr(), gethostbyname() and gethostent() return a pointer to a hostent structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found. Otherwise, a null pointer is returned.

On unsuccessful completion, gethostbyaddr() and gethostbyname() functions set *h_errno* to indicate the error.

Errors No errors are defined for endhostent(), gethostent() and sethostent().

The gethostbyaddr() and gethostbyname() functions will fail in the following cases, setting *h_errno* to the value shown in the list below. Any changes to errno are unspecified.

HOST_NOT_FOUND   No such host is known.

NO_DATA   The server recognised the request and the name but no address is available. Another type of request to the name server for the domain might return an answer.

NO_RECOVERY   An unexpected server failure occurred which can not be recovered.

TRY_AGAIN   A temporary and possibly transient error occurred, such as a failure of a server to respond.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also endservent(3XNET), htonl(3XNET), inet_addr(3XNET), attributes(5), standards(5)

Name endnetent, getnetbyaddr, getnetbyname, getnetent, setnetent – network database functions

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <netdb.h>

void endnetent(void);struct netent *getnetbyaddr(in_addr_t *net*, int *type*);

struct netent *getnetbyname(const char **name*);

struct netent *getnetent(void)

void setnetent(int *stayopen*);

Description The getnetbyaddr(), getnetbyname() and getnetent(), functions each return a pointer to a netent structure, the members of which contain the fields of an entry in the network database.

The getnetent() function reads the next entry of the database, opening a connection to the database if necessary.

The getnetbyaddr() function searches the database from the beginning, and finds the first entry for which the address family specified by type matches the n_addrtype member and the network number *net* matches the n_net member, opening a connection to the database if necessary. The *net* argument is the network number in host byte order.

The getnetbyname() function searches the database from the beginning and finds the first entry for which the network name specified by *name* matches the n_name member, opening a connection to the database if necessary.

The setnetent() function opens and rewinds the database. If the *stayopen* argument is non-zero, the connection to the net database will not be closed after each call to getnetent() (either directly, or indirectly through one of the other getnet*( ) functions).

The endnetent() function closes the database.

Usage The getnetbyaddr(), getnetbyname() and getnetent(), functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

These functions are generally used with the Internet address family.

Return Values On successful completion, getnetbyaddr(), getnetbyname() and getnetent(), return a pointer to a netent structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found. Otherwise, a null pointer is returned.

Errors No errors are defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also attributes(5), standards(5)

Name endprotoent, getprotobynumber, getprotobyname, getprotoent, setprotoent – network protocol database functions

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <netdb.h>

void endprotoent(void)

struct protoent *getprotobyname(const char *name);

struct protoent *getprotobynumber(int proto);

struct protoent *getprotoent(void)

void setprotoent(int stayopen);

Description The getprotobyname(), getprotobynumber() and getprotoent(), functions each return a pointer to a protoent structure, the members of which contain the fields of an entry in the network protocol database.

The getprotoent() function reads the next entry of the database, opening a connection to the database if necessary.

The getprotobyname() function searches the database from the beginning and finds the first entry for which the protocol name specified by *name* matches the p_name member, opening a connection to the database if necessary.

The getprotobynumber() function searches the database from the beginning and finds the first entry for which the protocol number specified by *number* matches the p_proto member, opening a connection to the database if necessary.

The setprotoent() function opens a connection to the database, and sets the next entry to the first entry. If the *stayopen* argument is non-zero, the connection to the network protocol database will not be closed after each call to getprotoent() (either directly, or indirectly through one of the other getproto*( ) functions).

The endprotoent() function closes the connection to the database.

Usage The getprotobyname(), getprotobynumber() and getprotoent() functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

These functions are generally used with the Internet address family.

Return Values On successful completion, getprotobyname(), getprotobynumber() and getprotoent() functions return a pointer to a protoent structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found. Otherwise, a null pointer is returned.

Errors   No errors are defined.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also   attributes(5), standards(5)

Name   endservent, getservbyport, getservbyname, getservent, setservent – network services database functions

Synopsis   cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <netdb.h>

void endservent(void)

struct servent *getservbyname(const char *_name_, const char *_proto_);

struct servent *getservbyport(int _port_, const char *_proto_);

struct servent *getservent(void)

void setservent(int _stayopen_);

Description   The getservbyname(), getservbyport() and getservent() functions each return a pointer to a servent structure, the members of which contain the fields of an entry in the network services database.

The getservent() function reads the next entry of the database, opening a connection to the database if necessary.

The getservbyname() function searches the database from the beginning and finds the first entry for which the service name specified by *name* matches the s_name member and the protocol name specified by *proto* matches the s_proto member, opening a connection to the database if necessary. If *proto* is a null pointer, any value of the s_proto member will be matched.

The getservbyport() function searches the database from the beginning and finds the first entry for which the port specified by *port* matches the s_port member and the protocol name specified by *proto* matches the s_proto member, opening a connection to the database if necessary. If *proto* is a null pointer, any value of the s_proto member will be matched. The *port* argument must be in network byte order.

The setservent() function opens a connection to the database, and sets the next entry to the first entry. If the *stayopen* argument is non-zero, the net database will not be closed after each call to the getservent() function, either directly, or indirectly through one of the other getserv*() functions.

The endservent() function closes the database.

Usage   The *port* argument of getservbyport() need not be compatible with the port values of all address families.

The getservent(), getservbyname() and getservbyport() functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

These functions are generally used with the Internet address family.

Return Values On successful completion, getservbyname(), getservbyport() and getservent() return a pointer to a servent structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found. Otherwise, a null pointer is returned.

Errors No errors are defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also endhostent(3XNET), endprotoent(3XNET), htonl(3XNET), inet_addr(3XNET), attributes(5), standards(5)

Name ethers, ether_ntoa, ether_aton, ether_ntohost, ether_hostton, ether_line – Ethernet address mapping operations

Synopsis cc [*flag* ...] *file* ... -lsocket -lnsl [*library* ...]
```
#include <sys/types.h>
#include <sys/ethernet.h>

char *ether_ntoa(const struct ether_addr *e);

struct ether_addr *ether_aton(const char *s);

int ether_ntohost(char *hostname, const struct ether_addr *e);

int ether_hostton(const char *hostname, struct ether_addr *e);

int ether_line(const char *l, struct ether_addr *e, char *hostname);
```

Description These routines are useful for mapping 48 bit Ethernet numbers to their ASCII representations or their corresponding host names, and vice versa.

The function ether_ntoa() converts a 48 bit Ethernet number pointed to by *e* to its standard ASCII representation; it returns a pointer to the ASCII string. The representation is of the form $x : x : x : x : x : x$ where x is a hexadecimal number between 0 and ff. The function ether_aton() converts an ASCII string in the standard representation back to a 48 bit Ethernet number; the function returns NULL if the string cannot be scanned successfully.

The function ether_ntohost() maps an Ethernet number (pointed to by *e*) to its associated hostname. The string pointed to by hostname must be long enough to hold the hostname and a NULL character. The function returns zero upon success and non-zero upon failure. Inversely, the function ether_hostton() maps a hostname string to its corresponding Ethernet number; the function modifies the Ethernet number pointed to by *e*. The function also returns zero upon success and non-zero upon failure. In order to do the mapping, both these functions may lookup one or more of the following sources: the ethers file, and the NIS maps ethers.byname and ethers.byaddr. The sources and their lookup order are specified in the /etc/nsswitch.conf file. See nsswitch.conf(4) for details.

The function ether_line() scans a line, pointed to by *l*, and sets the hostname and the Ethernet number, pointed to by *e*. The string pointed to by hostname must be long enough to hold the hostname and a NULL character. The function returns zero upon success and non-zero upon failure. The format of the scanned line is described by ethers(4).

Files /etc/ethers   Ethernet address to hostname database or domain

/etc/nsswitch.conf   configuration file for the name service switch

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also   ethers(4), nsswitch.conf(4), attributes(5)

Name freeaddrinfo, getaddrinfo – get address information

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <sys/socket.h>
#include <netdb.h>

void freeaddrinfo(struct addrinfo *ai);

int getaddrinfo(const char *restrict *nodename*,
    const char *restrict *servname*, const struct addrinfo *restrict *hints*,
    struct addrinfo **restrict *res*);

Description The freeaddrinfo() function frees one or more addrinfo structures returned by
getaddrinfo(), along with any additional storage associated with those structures. If the
ai_next member of the structure is not null, the entire list of structures is freed. The
freeaddrinfo() function supports the freeing of arbitrary sublists of an addrinfo list
originally returned by getaddrinfo().

The getaddrinfo() function translates the name of a service location (for example, a host
name) and/or a service name and returns a set of socket addresses and associated information
to be used in creating a socket with which to address the specified service.

The *nodename* and *servname* arguments are either null pointers or pointers to
null-terminated strings. One or both of these two arguments are supplied by the application as
a non-null pointer.

The format of a valid name depends on the address family or families. If a specific family is not
given and the name could be interpreted as valid within multiple supported families, the
implementation attempts to resolve the name in all supported families and, in absence of
errors, one or more results are returned.

If the *nodename* argument is not null, it can be a descriptive name or can be an address string.
If the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, valid descriptive names
include host names. If the specified address family is AF_INET or AF_UNSPEC, address strings
using Internet standard dot notation as specified in inet_addr(3XNET) are valid.

If the specified address family is AF_INET6 or AF_UNSPEC, standard IPv6 text forms described
in inet_ntop(3XNET) are valid.

If *nodename* is not null, the requested service location is named by *nodename*; otherwise, the
requested service location is local to the caller.

If *servname* is null, the call returns network-level addresses for the specified *nodename*. If
servname is not null, it is a null-terminated character string identifying the requested service.
This string can be either a descriptive name or a numeric representation suitable for use with
the address family or families. If the specified address family is AF_INET, AF_INET6, or
AF_UNSPEC, the service can be specified as a string specifying a decimal port number.

If the *hints* argument is not null, it refers to a structure containing input values that can direct the operation by providing options and by limiting the returned information to a specific socket type, address family and/or protocol. In this *hints* structure every member other than ai_flags, ai_family, ai_socktype, and ai_protocol is set to 0 or a null pointer. A value of AF_UNSPEC for ai_family means that the caller accepts any address family. A value of 0 for ai_socktype means that the caller accepts any socket type. A value of 0 for ai_protocol means that the caller accepts any protocol. If *hints* is a null pointer, the behavior is as if it referred to a structure containing the value 0 for the ai_flags, ai_socktype, and ai_protocol members, and AF_UNSPEC for the ai_family member.

The ai_flags member to which the *hints* parameter points is set to 0 or be the bitwise-inclusive OR of one or more of the values AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST, and AI_NUMERICSERV.

If the AI_PASSIVE flag is specified, the returned address information is suitable for use in binding a socket for accepting incoming connections for the specified service. In this case, if the *nodename* argument is null, then the IP address portion of the socket address structure is set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address. If the AI_PASSIVE flag is not specified, the returned address information is suitable for a call to connect(3XNET) (for a connection-mode protocol) or for a call to connect(), sendto(3XNET), or sendmsg(3XNET) (for a connectionless protocol). In this case, if the *nodename* argument is null, then the IP address portion of the socket address structure is set to the loopback address.

If the AI_CANONNAME flag is specified and the *nodename* argument is not null, the function attempts to determine the canonical name corresponding to *nodename* (for example, if *nodename* is an alias or shorthand notation for a complete name).

If the AI_NUMERICHOST flag is specified, then a non-null *nodename* string supplied is a numeric host address string. Otherwise, an EAI_NONAME error is returned. This flag prevents any type of name resolution service (for example, the DNS) from being invoked.

If the AI_NUMERICSERV flag is specified, then a non-null *servname* string supplied is a numeric port string. Otherwise, an EAI_NONAME error is returned. This flag prevents any type of name resolution service (for example, NIS) from being invoked.

If the AI_V4MAPPED flag is specified along with an ai_family of AF_INET6, then getaddrinfo() returns IPv4-mapped IPv6 addresses on finding no matching IPv6 addresses (ai_addrlen is 16). The AI_V4MAPPED flag is ignored unless ai_family equals AF_INET6. If the AI_ALL flag is used with the AI_V4MAPPED flag, then getaddrinfo() returns all matching IPv6 and IPv4 addresses. The AI_ALL flag without the AI_V4MAPPED flag is ignored.

The ai_socktype member to which argument hints points specifies the socket type for the service, as defined in socket(3XNET). If a specific socket type is not given (for example, a value of 0) and the service name could be interpreted as valid with multiple supported socket types, the implementation attempts to resolve the service name for all supported socket types

and, in the absence of errors, all possible results are returned. A non-zero socket type value limits the returned information to values with the specified socket type.

If the `ai_family` member to which hints points has the value `AF_UNSPEC`, addresses are returned for use with any address family that can be used with the specified *nodename* and/or *servname*. Otherwise, addresses are returned for use only with the specified address family. If `ai_family` is not `AF_UNSPEC` and `ai_protocol` is not 0, then addresses are returned for use only with the specified address family and protocol; the value of `ai_protocol` is interpreted as in a call to the `socket()` function with the corresponding values of `ai_family` and `ai_protocol`.

Return Values A 0 return value for `getaddrinfo()` indicates successful completion; a non-zero return value indicates failure. The possible values for the failures are listed in the ERRORS section.

Upon successful return of `getaddrinfo()`, the location to which res points refers to a linked list of `addrinfo` structures, each of which specifies a socket address and information for use in creating a socket with which to use that socket address. The list includes at least one `addrinfo` structure. The `ai_next` member of each structure contains a pointer to the next structure on the list, or a null pointer if it is the last structure on the list. Each structure on the list includes values for use with a call to the socket function, and a socket address for use with the connect function or, if the `AI_PASSIVE` flag was specified, for use with the bind(3XNET) function. The `ai_family`, `ai_socktype`, and `ai_protocol` members are usable as the arguments to the `socket()` function to create a socket suitable for use with the returned address. The `ai_addr` and `ai_addrlen` members are usable as the arguments to the `connect()` or `bind()` functions with such a socket, according to the `AI_PASSIVE` flag.

If *nodename* is not null, and if requested by the `AI_CANONNAME` flag, the `ai_canonname` member of the first returned `addrinfo` structure points to a null-terminated string containing the canonical name corresponding to the input *nodename*. If the canonical name is not available, then `ai_canonname` refers to the *nodename* argument or a string with the same contents. The contents of the `ai_flags` member of the returned structures are undefined.

All members in socket address structures returned by `getaddrinfo()` that are not filled in through an explicit argument (for example, `sin6_flowinfo`) are set to 0, making it easier to compare socket address structures.

Errors The `getaddrinfo()` function will fail if:

EAI_AGAIN	The name could not be resolved at this time. Future attempts might succeed.
EAI_BADFLAGS	The `ai_flags` member of the `addrinfo` structure had an invalid value.
EAI_FAIL	A non-recoverable error occurred when attempting to resolve the name.
EAI_FAMILY	The address family was not recognized.
EAI_MEMORY	There was a memory allocation failure when trying to allocate storage for the return value.

EAI_NONAME	he name does not resolve for the supplied parameters. Neither *nodename* nor *servname* were supplied. At least one of these must be supplied.
EAI_SERVICE	The service passed was not recognized for the specified socket type.
EAI_SOCKTYPE	The intended socket type was not recognized.
EAI_SYSTEM	A system error occurred. The error code can be found in errno.
EAI_OVERFLOW	An argument buffer overflowed.

Usage If the caller handles only TCP and not UDP, for example, then the ai_protocol member of the *hints* structure should be set to IPPROTO_TCP when getaddrinfo() is called.

If the caller handles only IPv4 and not IPv6, then the ai_family member of the *hints* structure should be set to AF_INET when getaddrinfo() is called.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also connect(3XNET), gai_strerror(3XNET), gethostbyname(3XNET), getnameinfo(3XNET), getservbyname(3XNET), inet_addr(3XNET), inet_ntop(3XNET), socket(3XNET), attributes(5), standards(5)

Name gai_strerror – address and name information error description

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <netdb.h>

const char *gai_strerror(int *ecode*);

Description The gai_strerror() function returns a text string describing an error value for the getaddrinfo(3XNET) and getnameinfo(3XNET) functions listed in the <netdb.h> header.

When the *ecode* argument is one of the following values listed in the <netdb.h> header:

EAI_AGAIN

EAI_BADFLAGS

EAI_FAIL

EAI_FAMILY

EAI_MEMORY

EAI_NONAME

EAI_SERVICE

EAI_SOCKTYPE

EAI_SYSTEM

the function return value points to a string describing the error. If the argument is not one of those values, the function returns a pointer to a string whose contents indicate an unknown error.

Return Values Upon successful completion, gai_strerror() returns a pointer to a string describing the error value.

Errors No errors are defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also getaddrinfo(3XNET), getnameinfo(3XNET), attributes(5), standards(5)

Name getaddrinfo, getnameinfo, freeaddrinfo, gai_strerror – translate between node name and address

Synopsis cc [*flag*...] *file* ... -lsocket -lnsl [*library* ...]
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *nodename*, const char *servname*,
    const struct addrinfo *hints*, struct addrinfo **res*);

int getnameinfo(const struct sockaddr *sa*, socklen_t *salen*,
    char *host*, socklen_t *hostlen*, char *serv*, socklen_t *servlen*,
    int *flags*);

void freeaddrinfo(struct addrinfo *ai*);

const char *gai_strerror(int *errcode*);

Description These functions perform translations from node name to address and from address to node name in a protocol-independent manner.

The getaddrinfo() function performs the node name to address translation. The *nodename* and *servname* arguments are pointers to null-terminated strings or NULL. One or both of these arguments must be a non-null pointer. In the normal client scenario, both the *nodename* and *servname* are specified. In the normal server scenario, only the *servname* is specified.

A non-null *nodename* string can be a node name or a numeric host address string. The *nodename* can also be an IPv6 zone-id in the form:

<address>%<zone-id>

The address is the literal IPv6 link-local address or host name of the destination. The zone-id is the interface ID of the IPv6 link used to send the packet. The zone-id can either be a numeric value, indicating a literal zone value, or an interface name such as hme0.

A non-null *servname* string can be either a service name or a decimal port number.

The caller can optionally pass an addrinfo structure, pointed to by the *hints* argument, to provide hints concerning the type of socket that the caller supports.

The addrinfo structure is defined as:

```
struct addrinfo {
int            ai_flags;    /* AI_PASSIVE, AI_CANONNAME,
                               AI_NUMERICHOST, AI_NUMERICSERV
                               AI_V4MAPPED, AI_ALL,
                               AI_ADDRCONFIG */
int            ai_family;   /* PF_xxx */
int            ai_socktype; /* SOCK_xxx */
int            ai_protocol; /* 0 or IPPROTO_xxx for IPv4 & IPv6 */
socklen_t      ai_addrlen;  /* length of ai_addr */
```

```
char           *ai_canonname; /* canonical name for nodename */
struct sockaddr *ai_addr;      /* binary address */
struct addrinfo *ai_next;      /* next structure in linked list */
};
```

In this *hints* structure, all members other than ai_flags, ai_family, ai_socktype, and ai_protocol must be 0 or a null pointer. A value of PF_UNSPEC for ai_family indicates that the caller will accept any protocol family. A value of 0 for ai_socktype indicates that the caller will accept any socket type. A value of 0 for ai_protocol indicates that the caller will accept any protocol. For example, if the caller handles only TCP and not UDP, then the ai_socktype member of the *hints* structure should be set to SOCK_STREAM when getaddrinfo() is called. If the caller handles only IPv4 and not IPv6, then the ai_family member of the *hints* structure should be set to PF_INET when getaddrinfo() is called. If the third argument to getaddrinfo() is a null pointer, it is as if the caller had filled in an addrinfo structure initialized to 0 with ai_family set to PF_UNSPEC.

Upon success, a pointer to a linked list of one or more addrinfo structures is returned through the final argument. The caller can process each addrinfo structure in this list by following the ai_next pointer, until a null pointer is encountered. In each returned addrinfo structure the three members ai_family, ai_socktype, and ai_protocol are the corresponding arguments for a call to the socket(3SOCKET) function. In each addrinfo structure the ai_addr member points to a filled-in socket address structure whose length is specified by the ai_addrlen member.

If the AI_PASSIVE bit is set in the ai_flags member of the *hints* structure, the caller plans to use the returned socket address structure in a call to bind(3SOCKET). In this case, if the *nodename* argument is a null pointer, the IP address portion of the socket address structure will be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address.

If the AI_PASSIVE bit is not set in the ai_flags member of the *hints* structure, then the returned socket address structure will be ready for a call to connect(3SOCKET) (for a connection-oriented protocol) or either connect(3SOCKET), sendto(3SOCKET), or sendmsg(3SOCKET) (for a connectionless protocol). If the *nodename* argument is a null pointer, the IP address portion of the socket address structure will be set to the loopback address.

If the AI_CANONNAME bit is set in the ai_flags member of the *hints* structure, then upon successful return the ai_canonname member of the first addrinfo structure in the linked list will point to a null-terminated string containing the canonical name of the specified *nodename*. A numeric host address string is not a name, and thus does not have a canonical name form; no address to host name translation is performed.

If the AI_NUMERICHOST bit is set in the ai_flags member of the *hints* structure, then a non-null *nodename* string must be a numeric host address string. Otherwise an error of EAI_NONAME is returned. This flag prevents any type of name resolution service (such as DNS) from being called.

If the `AI_NUMERICSERV` flag is specified, then a non-null servname string supplied will be a numeric port string. Otherwise, an [`EAI_NONAME`] error is returned. This flag prevents any type of name resolution service (for example, NIS) from being invoked.

If the `AI_V4MAPPED` flag is specified along with an `ai_family` of `AF_INET6`, then `getaddrinfo()` returns IPv4-mapped IPv6 addresses on finding no matching IPv6 addresses (`ai_addrlen` shall be 16). For example, if no AAAA records are found when using DNS, a query is made for A records. Any found records are returned as IPv4-mapped IPv6 addresses.

The `AI_V4MAPPED` flag is ignored unless `ai_family` equals `AF_INET6`.

If the AI_ALL flag is used with the AI_V4MAPPED flag, then `getaddrinfo()` returns all matching IPv6 and IPv4 addresses. For example, when using the DNS, queries are made for both AAAA records and A records, and `getaddrinfo()` returns the combined results of both queries. Any IPv4 addresses found are returned as IPv4-mapped IPv6 addresses.

The `AI_ALL` flag without the `AI_V4MAPPED` flag is ignored.

When `ai_family` is not specified (`AF_UNSPEC`), `AI_V4MAPPED` and `AI_ALL` flags are used only if `AF_INET6` is supported.

If the `AI_ADDRCONFIG` flag is specified, IPv4 addresses are returned only if an IPv4 address is configured on the local system, and IPv6 addresses are returned only if an IPv6 address is configured on the local system. For this case, the loopback address is not considered to be as valid as a configured address. For example, when using the DNS, a query for AAAA records should occur only if the node has at least one IPv6 address configured (other than IPv6 loopback) and a query for A records should occur only if the node has at least one IPv4 address configured (other than the IPv4 loopback).

All of the information returned by `getaddrinfo()` is dynamically allocated: the addrinfo structures as well as the socket address structures and canonical node name strings pointed to by the addrinfo structures. The `freeaddrinfo()` function is called to return this information to the system. For `freeaddrinfo()`, the addrinfo structure pointed to by the *ai* argument is freed, along with any dynamic storage pointed to by the structure. This operation is repeated until a null `ai_next` pointer is encountered.

To aid applications in printing error messages based on the EAI_* codes returned by `getaddrinfo()`, the `gai_strerror()` is defined. The argument is one of the `EAI_*` values defined below and the return value points to a string describing the error. If the argument is not one of the `EAI_*` values, the function still returns a pointer to a string whose contents indicate an unknown error.

The `getnameinfo()` function looks up an IP address and port number provided by the caller in the name service database and system-specific database, and returns text strings for both in buffers provided by the caller. The function indicates successful completion by a 0 return value; a non-zero return value indicates failure.

The first argument, *sa*, points to either a sockaddr_in structure (for IPv4) or a sockaddr_in6 structure (for IPv6) that holds the IP address and port number. The *salen* argument gives the length of the sockaddr_in or sockaddr_in6 structure.

The function returns the node name associated with the IP address in the buffer pointed to by the *host* argument.

The function can also return the IPv6 zone-id in the form:

```
<address>%<zone-id>
```

The caller provides the size of this buffer with the *hostlen* argument. The service name associated with the port number is returned in the buffer pointed to by *serv*, and the *servlen* argument gives the length of this buffer. The caller specifies not to return either string by providing a 0 value for the *hostlen* or *servlen* arguments. Otherwise, the caller must provide buffers large enough to hold the node name and the service name, including the terminating null characters.

To aid the application in allocating buffers for these two returned strings, the following constants are defined in <netdb.h>:

```
#define NI_MAXHOST   1025
#define NI_MAXSERV     32
```

The final argument is a flag that changes the default actions of this function. By default, the fully-qualified domain name (FQDN) for the host is looked up in the name service database and returned. If the flag bit NI_NOFQDN is set, only the node name portion of the FQDN is returned for local hosts.

If the flag bit NI_NUMERICHOST is set, or if the host's name cannot be located in the name service, the numeric form of the host's address is returned instead of its name, for example, by calling inet_ntop() (see inet(3SOCKET)) instead of getipnodebyname(3SOCKET). If the flag bit NI_NAMEREQD is set, an error is returned if the host's name cannot be located in the name service database.

If the flag bit NI_NUMERICSERV is set, the numeric form of the service address is returned (for example, its port number) instead of its name. The two NI_NUMERIC* flags are required to support the -n flag that many commands provide.

A fifth flag bit, NI_DGRAM, specifies that the service is a datagram service, and causes getservbyport(3SOCKET) to be called with a second argument of udp instead of the default tcp. This is required for the few ports (for example, 512-514) that have different services for UDP and TCP.

These NI_* flags are defined in <netdb.h> along with the AI_* flags already defined for getaddrinfo().

Return Values For getaddrinfo(), if the query is successful, a pointer to a linked list of one or more addrinfo structures is returned by the fourth argument and the function returns 0. The order of the addresses returned in the fourth argument is discussed in the ADDRESS ORDERING section. If the query fails, a non-zero error code will be returned. For getnameinfo(), if successful, the strings hostname and service are copied into *host* and *serv*, respectively. If unsuccessful, zero values for either *hostlen* or *servlen* will suppress the associated lookup; in this case no data is copied into the applicable buffer. If gai_strerror() is successful, a pointer to a string containing an error message appropriate for the EAI_* errors is returned. If *errcode* is not one of the EAI_* values, a pointer to a string indicating an unknown error is returned.

Address Ordering AF_INET6 addresses returned by the fourth argument of getaddrinfo() are ordered according to the algorithm described in *RFC 3484, Default Address Selection for Internet Protocol version 6 (IPv6)*. The addresses are ordered using a list of pair-wise comparison rules which are applied in order. If a rule determines that one address is better than another, the remaining rules are irrelevant to the comparison of those two addresses. If two addresses are equivalent according to one rule, the remaining rules act as a tie-breaker. The address ordering list of pair-wise comparison rules follow below:

Avoid unusable destinations.	Prefer a destination that is reachable through the IP routing table.
Prefer matching scope.	Prefer a destination whose scope is equal to the scope of its source address. See inet6(7P) for the definition of scope used by this rule.
Avoid link-local source.	Avoid selecting a link-local source address when the destination address is not a link-local address.
Avoid deprecated addresses.	Prefer a destination that is not deprecated (IFF_DEPRECATED).
Prefer matching label. This rule uses labels that are obtained through the IPv6 default address selection policy table. See ipaddrsel(1M) for a description of the default contents of the table and how the table is configured.	Prefer a destination whose label is equal to the label of its source address.
Prefer higher precedence. This rule uses precedence values that are obtained through the IPv6 default address selection policy table. See ipaddrsel(1M) for a description of the default contents of the table and how the table is configured.	Prefer the destination whose precedence is higher than the other destination.
Prefer native transport.	Prefer a destination if the interface that is used for sending packets to that destination is not an IP over IP tunnel.
Prefer smaller scope. See inet6(7P) for the definition of this rule.	Prefer the destination whose scope is smaller than the other destination.

Use longest matching prefix.	When the two destinations belong to the same address family, prefer the destination that has the longer matching prefix with its source address.

Errors The following names are the error values returned by getaddrinfo() and are defined in <netdb.h>:

EAI_ADDRFAMILY   Address family for nodename is not supported.

EAI_AGAIN   Temporary failure in name resolution has occurred .

EAI_BADFLAGS   Invalid value specified for ai_flags.

EAI_FAIL   Non-recoverable failure in name resolution has occurred.

EAI_FAMILY   The ai_family is not supported.

EAI_MEMORY   Memory allocation failure has occurred.

EAI_NODATA   No address is associated with *nodename.*

EAI_NONAME   Neither *nodename* nor *servname* is provided or known.

EAI_SERVICE   The *servname* is not supported for ai_socktype.

EAI_SOCKTYPE   The ai_socktype is not supported.

EAI_OVERFLOW   Argument buffer has overflowed.

EAI_SYSTEM   System error was returned in errno.

Files /etc/inet/hosts   local database that associates names of nodes with IP addresses

/etc/netconfig   network configuration database

/etc/nsswitch.conf   configuration file for the name service switch

Attributes See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also ipaddrsel(1M), gethostbyname(3NSL), getipnodebyname(3SOCKET), htonl(3SOCKET), inet(3SOCKET), netdb.h(3HEAD), socket(3SOCKET), hosts(4), nsswitch.conf(4), attributes(5), standards(5), inet6(7P)

Draves, R. *RFC 3484, Default Address Selection for Internet Protocol version 6 (IPv6)*. Network Working Group. February 2003.

Notes IPv4-mapped addresses are not recommended.

Name gethostbyname, gethostbyname_r, gethostbyaddr, gethostbyaddr_r, gethostent, gethostent_r, sethostent, endhostent – get network host entry

Synopsis cc [*flag*...] *file*... -lnsl [*library*...]
#include <netdb.h>

```
struct hostent *gethostbyname(const char *name);

struct hostent *gethostbyname_r(const char *name,
    struct hostent *result, char *buffer, int buflen,
    int *h_errnop);

struct hostent *gethostbyaddr(const char *addr, int len,
    int type);

struct hostent *gethostbyaddr_r(const char *addr, int length,
    int type, struct hostent *result, char *buffer,
    int buflen, int *h_errnop);

struct hostent *gethostent(void);

struct hostent *gethostent_r(struct hostent *result,
    char *buffer, int buflen, int *h_errnop);

int sethostent(int stayopen);

int endhostent(void);
```

Description These functions are used to obtain entries describing hosts. An entry can come from any of the sources for hosts specified in the /etc/nsswitch.conf file. See nsswitch.conf(4). These functions have been superseded by getipnodebyname(3SOCKET), getipnodebyaddr(3SOCKET), and getaddrinfo(3SOCKET), which provide greater portability to applications when multithreading is performed or technologies such as IPv6 are used. For example, the functions described in the following cannot be used with applications targeted to work with IPv6.

The gethostbyname() function searches for information for a host with the hostname specified by the character-string parameter *name*.

The gethostbyaddr() function searches for information for a host with a given host address. The parameter type specifies the family of the address. This should be one of the address families defined in <sys/socket.h>. See the NOTES section for more information. Also see the EXAMPLES section for information on how to convert an Internet IP address notation that is separated by periods (.) into an *addr* parameter. The parameter *len* specifies the length of the buffer indicated by *addr*.

All addresses are returned in network order. In order to interpret the addresses, byteorder(3SOCKET) must be used for byte order conversion.

The sethostent(), gethostent(), and endhostent() functions are used to enumerate host entries from the database.

The `sethostent()` function sets or resets the enumeration to the beginning of the set of host entries. This function should be called before the first call to `gethostent()`. Calls to `gethostbyname()` and `gethostbyaddr()` leave the enumeration position in an indeterminate state. If the *stayopen* flag is non-zero, the system can keep allocated resources such as open file descriptors until a subsequent call to `endhostent()`.

Successive calls to the `gethostent()` function return either successive entries or `NULL`, indicating the end of the enumeration.

The `endhostent()` function can be called to indicate that the caller expects to do no further host entry retrieval operations; the system can then deallocate resources it was using. It is still allowed, but possibly less efficient, for the process to call more host retrieval functions after calling `endhostent()`.

Reentrant Interfaces   The `gethostbyname()`, `gethostbyaddr()`, and `gethostent()` functions use static storage that is reused in each call, making these functions unsafe for use in multithreaded applications.

The `gethostbyname_r()`, `gethostbyaddr_r()`, and `gethostent_r()` functions provide reentrant interfaces for these operations.

Each reentrant interface performs the same operation as its non-reentrant counterpart, named by removing the _r suffix. The reentrant interfaces, however, use buffers supplied by the caller to store returned results and the interfaces are safe for use in both single-threaded and multithreaded applications.

Each reentrant interface takes the same parameters as its non-reentrant counterpart, as well as the following additional parameters. The parameter *result* must be a pointer to a `struct hostent` structure allocated by the caller. On successful completion, the function returns the host entry in this structure. The parameter *buffer* must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the host data. All of the pointers within the returned `struct hostent` *result* point to data stored within this buffer. See the RETURN VALUES section for more information. The buffer must be large enough to hold all of the data associated with the host entry. The parameter *buflen* should give the size in bytes of the buffer indicated by *buffer*. The parameter *h_errnop* should be a pointer to an integer. An integer error status value is stored there on certain error conditions. See the ERRORS section for more information.

For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. The `sethostent()` function can be used in a multithreaded application but resets the enumeration position for all threads. If multiple threads interleave calls to `gethostent_r()`, the threads will enumerate disjoint subsets of the host database.

Like their non-reentrant counterparts, `gethostbyname_r()` and `gethostbyaddr_r()` leave the enumeration position in an indeterminate state.

Return Values   Host entries are represented by the struct hostent structure defined in <netdb.h>:

```
struct hostent {
    char    *h_name;        /* canonical name of host */
    char    **h_aliases;    /* alias list */
    int     h_addrtype;     /* host address type */
    int     h_length;       /* length of address */
    char    **h_addr_list;  /* list of addresses */
};
```

See the EXAMPLES section for information about how to retrieve a "." separated Internet IP address string from the *h_addr_list* field of struct hostent.

The gethostbyname(), gethostbyname_r(), gethostbyaddr(), and gethostbyaddr_r() functions each return a pointer to a struct hostent if they successfully locate the requested entry; otherwise they return NULL.

The gethostent() and gethostent_r() functions each return a pointer to a struct hostent if they successfully enumerate an entry; otherwise they return NULL, indicating the end of the enumeration.

The gethostbyname(), gethostbyaddr(), and gethostent() functions use static storage, so returned data must be copied before a subsequent call to any of these functions if the data is to be saved.

When the pointer returned by the reentrant functions gethostbyname_r(), gethostbyaddr_r(), and gethostent_r() is not NULL, it is always equal to the *result* pointer that was supplied by the caller.

The sethostent() and endhostent() functions return 0 on success.

Errors   The reentrant functions gethostbyname_r(), gethostbyaddr_r(), and gethostent_r() will return NULL and set *errno* to ERANGE if the length of the buffer supplied by caller is not large enough to store the result. See Intro(2) for the proper usage and interpretation of errno in multithreaded applications.

The reentrant functions gethostbyname_r() and gethostbyaddr_r() set the integer pointed to by *h_errnop* to one of these values in case of error.

On failures, the non-reentrant functions gethostbyname() and gethostbyaddr() set a global integer *h_errno* to indicate one of these error codes (defined in <netdb.h>): HOST_NOT_FOUND, TRY_AGAIN, NO_RECOVERY, NO_DATA, and NO_ADDRESS.

If a resolver is provided with a malformed address, or if any other error occurs before gethostbyname() is resolved, then gethostbyname() returns an internal error with a value of −1.

The gethostbyname() function will set *h_errno* to NETDB_INTERNAL when it returns a NULL value.

Examples EXAMPLE 1 Using gethostbyaddr()

Here is a sample program that gets the canonical name, aliases, and "." separated Internet IP addresses for a given "." separated IP address:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
int main(int argc, const char **argv)
{
    in_addr_t addr;
    struct hostent *hp;
    char **p;
    if (argc != 2) {
        (void) printf("usage: %s IP-address\n", argv[0]);
        exit (1);
    }
    if ((int)(addr = inet_addr(argv[1])) == -1) {
        (void) printf("IP-address must be of the form a.b.c.d\n");
        exit (2);
    }
    hp = gethostbyaddr((char *)&addr, 4, AF_INET);
    if (hp == NULL) {
        (void) printf("host information for %s not found\n", argv[1]);
        exit (3);
    }
    for (p = hp->h_addr_list; *p != 0; p++) {
        struct in_addr in;
        char **q;
        (void) memcpy(&in.s_addr, *p, sizeof (in.s_addr));
         (void) printf("%s %s", inet_ntoa(in), hp—>h_name);
        for (q = hp->h_aliases; *q != 0; q++)
            (void) printf(" %s", *q);
        (void) putchar('\n');
    }
    exit (0);
}
```

Note that the preceding sample program is unsafe for use in multithreaded applications.

Files /etc/hosts   hosts file that associates the names of hosts with their Internet Protocol (IP) addresses

/etc/netconfig   network configuration database

/etc/nsswitch.conf   configuration file for the name service switch

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	See Reentrant Interfaces in the DESCRIPTION section.

See Also Intro(2), Intro(3), byteorder(3SOCKET), inet(3SOCKET), netdb.h(3HEAD), netdir(3NSL), hosts(4), netconfig(4), nss(4), nsswitch.conf(4), attributes(5)

Warnings The reentrant interfaces gethostbyname_r(), gethostbyaddr_r(), and gethostent_r() are included in this release on an uncommitted basis only and are subject to change or removal in future minor releases.

Notes To ensure that they all return consistent results, gethostbyname(), gethostbyname_r(), and netdir_getbyname() are implemented in terms of the same internal library function. This function obtains the system-wide source lookup policy based on the inet family entries in netconfig(4) and the hosts: entry in nsswitch.conf(4). Similarly, gethostbyaddr(), gethostbyaddr_r(), and netdir_getbyaddr() are implemented in terms of the same internal library function. If the inet family entries in netconfig(4) have a "-" in the last column for nametoaddr libraries, then the entry for hosts in nsswitch.conf will be used; nametoaddr libraries in that column will be used, and nsswitch.conf will not be consulted.

There is no analogue of gethostent() and gethostent_r() in the netdir functions, so these enumeration functions go straight to the hosts entry in nsswitch.conf. Thus enumeration can return results from a different source than that used by gethostbyname(), gethostbyname_r(), gethostbyaddr(), and gethostbyaddr_r().

All the functions that return a struct hostent must always return the *canonical name* in the *h_name* field. This name, by definition, is the well-known and official hostname shared between all aliases and all addresses. The underlying source that satisfies the request determines the mapping of the input name or address into the set of names and addresses in hostent. Different sources might do that in different ways. If there is more than one alias and more than one address in hostent, no pairing is implied between them.

The system attempts to put those addresses that are on the same subnet as the caller before addresses that are on different subnets. However, if address sorting is disabled by setting SORT_ADDRS to FALSE in the /etc/default/nss file, the system does not put the local subnet addresses first. See nss(4) for more information.

When compiling multithreaded applications, see Intro(3), MULTITHREADED APPLICATIONS, for information about the use of the _REENTRANT flag.

Use of the enumeration interfaces gethostent() and gethostent_r() is discouraged; enumeration might not be supported for all database sources. The semantics of enumeration are discussed further in nsswitch.conf(4).

The current implementations of these functions only return or accept addresses for the Internet address family (type AF_INET).

The form for an address of type AF_INET is a struct in_addr defined in <netinet/in.h>. The functions described in inet(3SOCKET), and illustrated in the EXAMPLES section, are helpful in constructing and manipulating addresses in this form.

When the caller provides the IP address (the *addr* argument of gethostbyaddr() and gethostbyaddr_r()), the *addr* argument should be aligned on a word boundary or the code must be changed to memcpy(3C) the argument to an aligned area; otherwise an error such as a SIGBUS may result.

Name gethostname – get name of current host

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <unistd.h>

int gethostname(char **name*, size_t *namelen*);

Description The gethostname() function returns the standard host name for the current machine. The *namelen* argument specifies the size of the array pointed to by the *name* argument. The returned name is null-terminated, except that if *namelen* is an insufficient length to hold the host name, then the returned name is truncated and it is unspecified whether the returned name is null-terminated.

Host names are limited to 255 bytes.

Return Values On successful completion, 0 is returned. Otherwise, –1 is returned.

Errors No errors are defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also uname(1), gethostid(3C), attributes(5), standards(5)

Name getifaddrs, freeifaddrs – get interface addresses

Synopsis cc [*flag*...] *file* ... -lsocket -lnsl [*library* ...]
```
#include <sys/types.h>
#include <sys/socket.h>
#include <ifaddrs.h>

int getifaddrs(struct ifaddrs **ifap);

void freeifaddrs(struct ifaddrs *ifp);
```

Description The getifaddrs() function stores a reference to a linked list of network interface addresses
on the local machine in the memory referenced by *ifap*. The list consists of ifaddrs structures,
as defined in the include file <ifaddrs.h>. Each element of the list describes one network
interface address. The caller can process each ifaddrs structure in this list by following the
ifa_next pointer, until a null pointer is encountered.

```
struct ifaddrs {
 struct ifaddrs          *ifa_next;    /* next structure in linked list*/
 char                    *ifa_name;    /* Interface name */
 uint64_t                ifa_flags;    /* Interface flags (if_tcp(7P)) */
 struct sockaddr         *ifa_addr;    /* Interface address */
 struct sockaddr         *ifa_netmask; /* Interface netmask */
 union {
        /* Interface broadcast address */
        struct sockaddr        *ifa_dstaddr;
        /* P2P interface destination */
        struct sockaddr        *ifa_broadaddr;
  } ifa_ifu;
 void                    *ifa_data; /* Address specific data (unused) */
};
#ifndef ifa_broadaddr
#define ifa_broadaddr    ifa_ifu.ifu_broadaddr
#endif
#ifndef ifa_dstaddr
#define ifa_dstaddr      ifa_ifu.ifu_dstaddr
#endif
```

The *ifa_name* member contains the interface name.

The *ifa_flags* member contains the interface flags.

The *ifa_addr* member references the address of the interface. (The *sa_family* member of the
ifa_addr member should be consulted to determine the format of the *ifa_addr* address.)

The *ifa_netmask* member references the netmask associated with *ifa_addr*, if one is set,
otherwise it is NULL.

The *ifa_broadaddr* member, which should only be referenced for non-P2P interfaces,
references the broadcast address associated with *ifa_addr*, if one exists, otherwise it is NULL.

The *ifa_dstaddr* member references the destination address on a P2P inter face, if one exists, otherwise it is NULL.

The *ifa_data* member is currently unused.

The data returned by getifaddrs() is dynamically allocated and should be freed using freeifaddrs() when no longer needed.

Return Values The getifaddrs() function returns the value 0 if successful; otherwise -1 is returned and errno is set to indicate the error.

Errors The getifaddrs() function may fail and set errno for any of the errors specified for ioctl(2), socket(3SOCKET), and malloc(3C).

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also ipadm(1M), ifconfig(1M), ioctl(2), malloc(3C), socket(3SOCKET), if_tcp(7P), attributes(5)

Name getipnodebyname, getipnodebyaddr, freehostent – get IP node entry

Synopsis cc [*flag*...] *file*... -lsocket -lnsl [*library*...]
#include <sys/socket.h>
#include <netdb.h>

struct hostent *getipnodebyname(const char **name*, int *af*,
    int *flags*, int *error_num*);

struct hostent *getipnodebyaddr(const void **src*, size_t *len*,
    int *af*, int *error_num*);

void freehostent(struct hostent **ptr*);

Parameters *af*   Address family

flags   Various flags

name   Name of host

error_num   Error storage

src   Address for lookup

len   Length of address

ptr   Pointer to hostent structure

Description The getipnodebyname() function searches the ipnodes database from the beginning. The function finds the first h_name member that matches the hostname specified by *name*. The function takes an *af* argument that specifies the address family. The address family can be AF_INET for IPv4 addresses or AF_INET6 for IPv6 addresses. The *flags* argument determines what results are returned based on the value of *flags*. If the *flags* argument is set to 0 (zero), the default operation of the function is specified as follows:

- If the *af* argument is AF_INET, a query is made for an IPv4 address. If successful, IPv4 addresses are returned and the h_length member of the hostent structure is 4. Otherwise, the function returns a NULL pointer.

- If the *af* argument is AF_INET6, a query is made for an IPv6 address. If successful, IPv6 addresses are returned and the h_length member of the hostent structure is 16. Otherwise, the function returns a NULL pointer.

The *flags* argument changes the default actions of the function. Set the *flags* argument with a logical OR operation on any of combination of the following values:

```
AI_V4MAPPED
AI_ALL
AI_ADDRCONFIG
```

The special flags value, AI_DEFAULT, should handle most applications. Porting simple applications to use IPv6 replaces the call

```
hptr = gethostbyname(name);
```

with

```
hptr = getipnodebyname(name, AF_INET6, AI_DEFAULT, &error_num);
```

The *flags* value 0 (zero) implies a strict interpretation of the *af* argument:

- If *flags* is 0 and *af* is AF_INET, the caller wants only IPv4 addresses. A query is made for A records. If successful, IPv4 addresses are returned and the h_length member of the hostent structure is 4. Otherwise, the function returns a NULL pointer.

- If *flags* is 0 and *af* is AF_INET6, the caller wants only IPv6 addresses. A query is made for AAAA records. If successful, IPv6 addresses are returned and the h_length member of the hostent structure is 16. Otherwise, the function returns a NULL pointer.

Logically OR other constants into the *flags* argument to modify the behavior of the getipnodebyname() function.

- If the AI_V4MAPPED flag is specified with *af* set to AF_INET6, the caller can accept IPv4-mapped IPv6 addresses. If no AAAA records are found, a query is made for A records. Any A records found are returned as IPv4-mapped IPv6 addresses and the h_length is 16. The AI_V4MAPPED flag is ignored unless *af* equals AF_INET6.

- The AI_ALL flag is used in conjunction with the AI_V4MAPPED flag, exclusively with the IPv6 address family. When AI_ALL is logically ORed with AI_V4MAPPED flag, the caller wants all addresses: IPv6 and IPv4-mapped IPv6 addresses. A query is first made for AAAA records and, if successful, IPv6 addresses are returned. Another query is then made for A records. Any A records found are returned as IPv4-mapped IPv6 addresses and the h_length is 16. Only when both queries fail does the function return a NULL pointer. The AI_ALL flag is ignored unless *af* is set to AF_INET6.

- The AI_ADDRCONFIG flag specifies that a query for AAAA records should occur only when the node is configured with at least one IPv6 source address. A query for A records should occur only when the node is configured with at least one IPv4 source address. For example, if a node is configured with no IPv6 source addresses, *af* equals AF_INET6, and the node name queried has both AAAA and A records, then:

  - A NULL pointer is returned when only the AI_ADDRCONFIG value is specified.

  - The A records are returned as IPv4-mapped IPv6 addresses when the AI_ADDRCONFIG and AI_V4MAPPED values are specified.

The special flags value, AI_DEFAULT, is defined as

```
#define  AI_DEFAULT  (AI_V4MAPPED | AI_ADDRCONFIG)
```

The getipnodebyname() function allows the *name* argument to be a node name or a literal address string: a dotted-decimal IPv4 address or an IPv6 hex address. Applications do not have to call inet_pton(3SOCKET) to handle literal address strings.

Four scenarios arise based on the type of literal address string and the value of the *af* argument. The two simple cases occur when *name* is a dotted-decimal IPv4 address and *af* equals AF_INET and when *name* is an IPv6 hex address and *af* equals AF_INET6. The members of the returned hostent structure are:

h_name   Pointer to a copy of the name argument

h_aliases   NULL pointer.

h_addrtype   Copy of the *af* argument.

h_length   4 for AF_INET or 16 for AF_INET6.

h_addr_list   Array of pointers to 4-byte or 16-byte binary addresses. The array is
                terminated by a NULL pointer.

Return Values Upon successful completion, getipnodebyname() and getipnodebyaddr() return a hostent structure. Otherwise they return NULL.

The hostent structure does not change from the existing definition when used with gethostbyname(3NSL). For example, host entries are represented by the struct hostent structure defined in <netdb.h>:

```
struct  hostent {
        char    *h_name;        /* canonical name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    **h_addr_list;  /* list of addresses */
};
```

An error occurs when *name* is an IPv6 hex address and *af* equals AF_INET. The return value of the function is a NULL pointer and error_num equals HOST_NOT_FOUND.

The getipnodebyaddr() function has the same arguments as the existing gethostbyaddr(3NSL) function, but adds an error number. As with getipnodebyname(), getipnodebyaddr() is thread-safe. The error_num value is returned to the caller with the appropriate error code to support thread-safe error code returns. The following error conditions can be returned for error_num:

HOST_NOT_FOUND   Host is unknown.

NO_DATA   No address is available for the *name* specified in the server request. This
                  error is not a soft error. Another type of *name* server request might be
                  successful.

NO_RECOVERY   An unexpected server failure occurred, which is a non-recoverable error.

TRY_AGAIN   This error is a soft error that indicates that the local server did not
                  receive a response from an authoritative server. A retry at some later

time might be successful.

One possible source of confusion is the handling of IPv4-mapped IPv6 addresses and IPv4-compatible IPv6 addresses, but the following logic should apply:

1. If *af* is AF_INET6, and if *len* equals 16, and if the IPv6 address is an IPv4-mapped IPv6 address or an IPv4-compatible IPv6 address, then skip over the first 12 bytes of the IPv6 address, set *af* to AF_INET, and set *len* to 4.

2. If *af* is AF_INET, lookup the *name* for the given IPv4 address.

3. If *af* is AF_INET6, lookup the *name* for the given IPv6 address.

4. If the function is returning success, then the single address that is returned in the hostent structure is a copy of the first argument to the function with the same address family that was passed as an argument to this function.

All four steps listed are performed in order.

This structure, and the information pointed to by this structure, are dynamically allocated by getipnodebyname() and getipnodebyaddr(). The freehostent() function frees this memory.

Examples   EXAMPLE 1   Getting the Canonical Name, Aliases, and Internet IP Addresses for a Given Hostname

The following is a sample program that retrieves the canonical name, aliases, and all Internet IP addresses, both version 6 and version 4, for a given hostname.

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

main(int argc, const char **argv)
{
char abuf[INET6_ADDRSTRLEN];
int error_num;
struct hostent *hp;
char **p;

    if (argc != 2) {
        (void) printf("usage: %s hostname\n", argv[0]);
        exit (1);
    }

    /* argv[1] can be a pointer to a hostname or literal IP address */
    hp = getipnodebyname(argv[1], AF_INET6, AI_ALL | AI_ADDRCONFIG |
```

EXAMPLE 1 Getting the Canonical Name, Aliases, and Internet IP Addresses for a Given Hostname
(Continued)

```
        AI_V4MAPPED, &error_num);
    if (hp == NULL) {
        if (error_num == TRY_AGAIN) {
            printf("%s: unknown host or invalid literal address "
                "(try again later)\n", argv[1]);
        } else {
            printf("%s: unknown host or invalid literal address\n",
                argv[1]);
        }
        exit (1);
    }
    for (p = hp->h_addr_list; *p != 0; p++) {
        struct in6_addr in6;
        char **q;

        bcopy(*p, (caddr_t)&in6, hp->h_length);
        (void) printf("%s\t%s", inet_ntop(AF_INET6, (void *)&in6,
            abuf, sizeof(abuf)), hp->h_name);
        for (q = hp->h_aliases; *q != 0; q++)
        (void) printf(" %s", *q);
        (void) putchar('\n');
    }
    freehostent(hp);
    exit (0);
    }
```

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   getaddrinfo(3SOCKET), gethostbyname(3NSL), htonl(3SOCKET), inet(3SOCKET),
netdb.h(3HEAD), hosts(4), nsswitch.conf(4), attributes(5)

Notes   No enumeration functions are provided for IPv6. Existing enumeration functions such as
sethostent(3NSL) do not work in combination with the getipnodebyname() and
getipnodebyaddr() functions.

All the functions that return a struct hostent must always return the canonical in the
h_name field. This name, by definition, is the well-known and official hostname shared
between all aliases and all addresses. The underlying source that satisfies the request

determines the mapping of the input name or address into the set of names and addresses in hostent. Different sources might make such as determination in different ways. If more than one alias and more than one address in hostent exist, no pairing is implied between the alias and address.

The current implementations of these functions return or accept only addresses for the Internet address family (type AF_INET) or the Internet address family Version 6 (type AF_INET6).

IPv4-mapped addresses are not recommended. The getaddrinfo(3SOCKET) function is preferred over getipnodebyaddr() because it allows applications to lookup IPv4 and IPv6 addresses without relying on IPv4-mapped addresses.

The form for an address of type AF_INET is a struct in_addr defined in <netinet/in.h>. The form for an address of type AF_INET6 is a struct in6_addr, also defined in <netinet/in.h>. The functions described in inet_ntop(3SOCKET) and inet_pton(3SOCKET) that are illustrated in the EXAMPLES section are helpful in constructing and manipulating addresses in either of these forms.

Name getipsecalgbyname, getipsecalgbynum, freeipsecalgent – query algorithm mapping entries

Synopsis
```
cc [ flag... ] file... -lnsl [ library... ]
#include <netdb.h>

struct ipsecalgent *getipsecalgbyname
    (const char *alg_name, int protocol_num, int *errnop);

struct ipsecalgent *getipsecalgbynum(int alg_num,
    int protocol_num, int *errnop);

void freeipsecalgent(struct ipsecalgent *ptr);
```

Description Use the getipsecalgbyname(), getipsecalgbynum(), freeipsecalgent() functions to obtain the IPsec algorithm mappings that are defined by ipsecalgs(1M). The IPsec algorithms and associated protocol name spaces are defined by *RFC 2407*.

getipsecalgbyname() and getipsecalgbynum() return a structure that describes the algorithm entry found. This structure is described in the RETURN VALUES section below.

freeipsecalgent() must be used by the caller to free the structures returned by getipsecalgbyname() and getipsecalgbynum() when they are no longer needed.

Both getipsecalgbyname() and getipsecalgbynum() take as parameter the protocol identifier in which the algorithm is defined. See getipsecprotobyname(3NSL) and getipsecprotobyname(3NSL).

The following protocol numbers are pre-defined:

IPSEC_PROTO_ESP   Defines the encryption algorithms (transforms) that can be used by IPsec to provide data confidentiality.

IPSEC_PROTO_AH   Defines the authentication algorithms (transforms) that can be used by IPsec to provide authentication.

getipsecalgbyname() looks up the algorithm by its name, while getipsecalgbynum() looks up the algorithm by its assigned number.

Parameters *errnop*   A pointer to an integer used to return an error status value on certain error conditions. See ERRORS.

Return Values The getipsecalgbyname() and getipsecalgbynum() functions return a pointer to the structure ipsecalgent_t, defined in <netdb.h>. If the requested algorithm cannot be found, these functions return NULL.

The structure ipsecalgent_t is defined as follows:

```
typedef struct ipsecalgent {
    char **a_names;      /* algorithm names */
    int a_proto_num;     /* protocol number */
    int a_alg_num;       /* algorithm number */
```

```
        char *a_mech_name;    /* mechanism name */
        int *a_block_sizes;   /* supported block sizes */
        int *a_key_sizes;     /* supported key sizes */
        int a_key_increment;  /* key size increment */
        int *a_mech_params;   /* mechanism specific parameters */
        int a_alg_flags;      /* algorithm flags */
} ipsecalgent_t;
```

If a_key_increment is non-zero, a_key_sizes[0] contains the default key size for the algorithm. a_key_sizes[1] and a_key_sizes[2] specify the smallest and biggest key sizes support by the algorithm, and a_key_increment specifies the valid key size increments in that range.

If a_key_increment is zero, the array a_key_sizes contains the set of key sizes, in bits, supported by the algorithm. The last key length in the array is followed by an element of value 0. The first element of this array is used as the default key size for the algorithm.

a_name is an array of algorithm names, terminated by an element containing a NULL pointer. a_name[0] is the primary name for the algorithm.

a_proto_num is the protocol identifer of this algorithm. a_alg_num is the algorithm number. a_mech_name contains the mechanism name associated with the algorithm.

a_block_sizes is an array containing the supported block lengths or MAC lengths, in bytes, supported by the algorithm. The last valid value in the array is followed by an element containing the value 0.

a_block_sizes is an array containing the supported block lengths or MAC lengths, in bytes, supported by the algorithm. The last valid value in the array is followed by an element containing the value 0.

Errors When the specified algorithm cannot be returned to the caller, getipsecalgbyname() and getipsecalgbynum() return a value of NULL and set the integer pointed to by the *errnop* parameter to one of the following values:

ENOMEM   Not enough memory

ENOENT   Specified algorithm not found

EINVAL   Specified protocol number not found

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library (32 bit)

ATTRIBUTE TYPE	ATTRIBUTE VALUE
	SUNWcslx (64 bit)
MT-Level	MT-Safe
Interface Stability	Committed

See Also   cryptoadm(1M), ipsecalgs(1M), getipsecprotobyname(3NSL), getipsecprotobyname(3NSL), attributes(5)

Piper, D. *RFC 2407, The Internet IP Security Domain of Interpretation for ISAKMP*. Network Working Group. November, 1998.

Name getipsecprotobyname, getipsecprotobynum – query IPsec protocols entries

Synopsis cc -flag ... *file* ...-lnsl [-library ...]
#include <netdb.h>

int getipsecprotobyname(const char *_proto_name_

char *getipsecprotobynum(int *proto_num*p

Description Use the getipsecprotobyname() and getipsecprotobynum() functions to obtain the IPsec algorithm mappings that are defined by ipsecalgs(1M). You can also use the getipsecprotobyname() and getipsecprotobynum() functions in conjunction with getipsecalgbyname(3NSL) and getipsecalgbynum(3NSL) to obtain information about the supported IPsec algorithms. The IPsec algorithms and associated protocol name spaces are defined by *RFC 2407*.

getipsecprotobyname() takes as an argument the name of an IPsec protocol and returns its assigned protocol number. The character string returned by the getipsecprotobyname() function must be freed by the called when it is no longer needed.

getipsecprotobynum() takes as an argument a protocol number and returns the corresponding protocol name.

The following protocol numbers are pre-defined:

IPSEC_PROTO_ESP   Defines the encryption algorithms (transforms) that can be used by IPsec to provide data confidentiality.

IPSEC_PROTO_AH   Defines the authentication algorithms (transforms) that can be used by IPsec to provide authentication.

Parameters *proto_name*   A pointer to the name of an IPsec protocol.

proto_num   A pointer to a protocol number. conditions.

Return Values The getipsecprotobyname() function returns a protocol number upon success, or –1 if the protocol specified does not exist.

The getipsecprotobynum() function returns a protocol name upon success, or the NULL value if the protocol number specified does not exist.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library (32 bit)
	SUNWcslx (64 bit)
MT Level	MT Safe

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also ipsecalgs(1M), getipsecalgbyname(3NSL), getipsecalgbyname(3NSL), attributes(5)

Piper, D. *RFC 2407, The Internet IP Security Domain of Interpretation for ISAKMP*. Network Working Group. November, 1998.

Name getnameinfo – get name information

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *restrict *sa*, socklen_t *salen*,
    char *restrict *node*, socklen_t *nodelen*, char *restrict *service*,
    socklen_t *servicelen*, unsigned *flags*);

Description The getnameinfo() function translates a socket address to a node name and service location, all of which are defined as in getaddrinfo(3XNET).

The *sa* argument points to a socket address structure to be translated. If the socket address structure contains an IPv4-mapped IPv6 address or an IPv4-compatible IPv6 address, the implementation extracts the embedded IPv4 address and lookup the node name for that IPv4 address.

If the *node* argument is non-NULL and the nodelen argument is non-zero, then the *node* argument points to a buffer able to contain up to *nodelen* characters that receives the node name as a null-terminated string. If the *node* argument is NULL or the *nodelen* argument is zero, the node name is not returned. If the node's name cannot be located, the numeric form of the node's address is returned instead of its name.

If the *service* argument is non-NULL and the *servicelen* argument is non-zero, then the *service* argument points to a buffer able to contain up to *servicelen* bytes that receives the service name as a null-terminated string. If the *service* argument is NULL or the *servicelen* argument is zero, the service name is not returned. If the service's name cannot be located, the numeric form of the service address (for example, its port number) is returned instead of its name.

The *flags* argument is a flag that changes the default actions of the function. By default the fully-qualified domain name (FQDN) for the host is returned, but:

- If the flag bit NI_NOFQDN is set, only the node name portion of the FQDN is returned for local hosts.

- If the flag bit NI_NUMERICHOST is set, the numeric form of the host's address is returned instead of its name, under all circumstances.

- If the flag bit NI_NAMEREQD is set, an error is returned if the host's name cannot be located.

- If the flag bit NI_NUMERICSERV is set, the numeric form of the service address is returned (for example, its port number) instead of its name, under all circumstances.

- If the flag bit NI_DGRAM is set, this indicates that the service is a datagram service (SOCK_DGRAM). The default behavior assumes that the service is a stream service (SOCK_STREAM).

Return Values A 0 return value for getnameinfo() indicates successful completion; a non-zero return value indicates failure. The possible values for the failures are listed in the ERRORS section.

Upon successful completion, getnameinfo() returns the node and service names, if requested, in the buffers provided. The returned names are always null-terminated strings.

Errors The getnameinfo() function will fail if:

EAI_AGAIN   The name could not be resolved at this time. Future attempts might succeed.

EAI_BADFLAGS   The *flags* argument had an invalid value.

EAI_FAIL   A non-recoverable error occurred.

EAI_FAMILY   The address family was not recognized or the address length was invalid for the specified family.

EAI_MEMORY   There was a memory allocation failure.

EAI_NONAME   The name does not resolve for the supplied parameters. NI_NAMEREQD is set and the host's name cannot be located, or both *nodename* and *servname* were NULL.

EAI_SYSTEM   A system error occurred. The error code can be found in errno.

Usage If the returned values are to be used as part of any further name resolution (for example, passed to getaddrinfo()), applications should provide buffers large enough to store any result possible on the system.

Given the IPv4-mapped IPv6 address "::ffff:1.2.3.4", the implementation performs a lookup as if the socket address structure contains the IPv4 address "1.2.3.4".

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also gai_strerror(3XNET), getaddrinfo(3XNET), getservbyname(3XNET), socket(3XNET), attributes(5), standards(5)

Notes The IPv6 unspecified address ("::") and the IPv6 loopback address ("::1") are not IPv4-compatible addresses. If the address is the IPv6 unspecified address ("::"), a lookup is not performed, and the EAI_NONAME error is returned.

The two `NI_NUMERIC`*xxx* flags are required to support the `-n` flag that many commands provide.

The `NI_DGRAM` flag is required for the few `AF_INET` and `AF_INET6` port numbers (for example, [512,514]) that represent different services for UDP and TCP.

Name getnetbyname, getnetbyname_r, getnetbyaddr, getnetbyaddr_r, getnetent, getnetent_r, setnetent, endnetent – get network entry

Synopsis cc [*flag* ...] *file* ... -lsocket -lnsl [*library* ...]
#include <netdb.h>

struct netent *getnetbyname(const char **name*);

struct netent *getnetbyname_r(const char **name*, struct netent **result*,
    char **buffer*, int *buflen*);

struct netent *getnetbyaddr(long *net*, int *type*);

struct netent *getnetbyaddr_r(long *net*, int *type*, struct netent **result*,
    char **buffer*, int *buflen*);

struct netent *getnetent(void);

struct netent *getnetent_r(struct netent **result*, char **buffer*,
    int *buflen*);

int setnetent(int *stayopen*);

int endnetent(void);

Description These functions are used to obtain entries for networks. An entry may come from any of the sources for networks specified in the /etc/nsswitch.conf file. See nsswitch.conf(4).

getnetbyname() searches for a network entry with the network name specified by the character string parameter *name*.

getnetbyaddr() searches for a network entry with the network address specified by *net*. The parameter type specifies the family of the address. This should be one of the address families defined in <sys/socket.h>. See the NOTES section below for more information.

Network numbers and local address parts are returned as machine format integer values, that is, in host byte order. See also inet(3SOCKET).

The netent.n_net member in the netent structure pointed to by the return value of the above functions is calculated by inet_network(). The inet_network() function returns a value in host byte order that is aligned based upon the input string. For example:

Text	Value
"10"	0x0000000a
"10.0"	0x00000a00
"10.0.1"	0a000a0001
"10.0.1.28"	0x0a000180

Commonly, the alignment of the returned value is used as a crude approximate of pre-CIDR (Classless Inter-Domain Routing) subnet mask. For example:

```
in_addr_t addr, mask;

addr = inet_network(net_name);
mask= ~(in_addr_t)0;
if ((addr & IN_CLASSA_NET) == 0)
        addr <<= 8, mask <<= 8;
if ((addr & IN_CLASSA_NET) == 0)
        addr <<= 8, mask <<= 8;
if ((addr & IN_CLASSA_NET) == 0)
        addr <<= 8, mask <<= 8;
```

This usage is deprecated by the CIDR requirements. See Fuller, V., Li, T., Yu, J., and Varadhan, K. *RFC 1519, Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy*. Network Working Group. September 1993.

The functions setnetent(), getnetent(), and endnetent() are used to enumerate network entries from the database.

setnetent() sets (or resets) the enumeration to the beginning of the set of network entries. This function should be called before the first call to getnetent(). Calls to getnetbyname() and getnetbyaddr() leave the enumeration position in an indeterminate state. If the *stayopen* flag is non-zero, the system may keep allocated resources such as open file descriptors until a subsequent call to endnetent().

Successive calls to getnetent() return either successive entries or NULL, indicating the end of the enumeration.

endnetent() may be called to indicate that the caller expects to do no further network entry retrieval operations; the system may then deallocate resources it was using. It is still allowed, but possibly less efficient, for the process to call more network entry retrieval functions after calling endnetent().

Reentrant Interfaces   The functions getnetbyname(), getnetbyaddr(), and getnetent() use static storage that is reused in each call, making these routines unsafe for use in multi-threaded applications.

The functions getnetbyname_r(), getnetbyaddr_r(), and getnetent_r() provide reentrant interfaces for these operations.

Each reentrant interface performs the same operation as its non-reentrant counterpart, named by removing the "_r" suffix. The reentrant interfaces, however, use buffers supplied by the caller to store returned results, and are safe for use in both single-threaded and multi-threaded applications.

Each reentrant interface takes the same parameters as its non-reentrant counterpart, as well as the following additional parameters. The parameter *result* must be a pointer to a struct netent structure allocated by the caller. On successful completion, the function returns the

network entry in this structure. The parameter *buffer* must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the network entry data. All of the pointers within the returned struct netent *result* point to data stored within this buffer. See RETURN VALUES. The buffer must be large enough to hold all of the data associated with the network entry. The parameter *buflen* should give the size in bytes of the buffer indicated by *buffer*.

For enumeration in multi-threaded applications, the position within the enumeration is a process-wide property shared by all threads. setnetent() may be used in a multi-threaded application but resets the enumeration position for all threads. If multiple threads interleave calls to getnetent_r(), the threads will enumerate disjointed subsets of the network database.

Like their non-reentrant counterparts, getnetbyname_r() and getnetbyaddr_r() leave the enumeration position in an indeterminate state.

Return Values   Network entries are represented by the struct netent structure defined in <netdb.h>.

The functions getnetbyname(), getnetbyname_r, getnetbyaddr, and getnetbyaddr_r() each return a pointer to a struct netent if they successfully locate the requested entry; otherwise they return NULL.

The functions getnetent() and getnetent_r() each return a pointer to a struct netent if they successfully enumerate an entry; otherwise they return NULL, indicating the end of the enumeration.

The functions getnetbyname(), getnetbyaddr(), and getnetent() use static storage, so returned data must be copied before a subsequent call to any of these functions if the data is to be saved.

When the pointer returned by the reentrant functions getnetbyname_r(), getnetbyaddr_r(), and getnetent_r() is non-NULL, it is always equal to the *result* pointer that was supplied by the caller.

The functions setnetent() and endnetent() return 0 on success.

Errors   The reentrant functions getnetbyname_r(), getnetbyaddr_r and getnetent_r() will return NULL and set errno to ERANGE if the length of the buffer supplied by caller is not large enough to store the result. See Intro(2) for the proper usage and interpretation of errno in multi-threaded applications.

Files   /etc/networks   network name database

/etc/nsswitch.conf   configuration file for the name service switch

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also   Intro(2), Intro(3), byteorder(3SOCKET), inet(3SOCKET), netdb.h(3HEAD), networks(4), nsswitch.conf(4), attributes(5)

Fuller, V., Li, T., Yu, J., and Varadhan, K. *RFC 1519, Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy*. Network Working Group. September 1993.

Warnings   The reentrant interfaces getnetbyname_r(), getnetbyaddr_r(), and getnetent_r() are included in this release on an uncommitted basis only, and are subject to change or removal in future minor releases.

Notes   The current implementation of these functions only return or accept network numbers for the Internet address family (type AF_INET). The functions described in inet(3SOCKET) may be helpful in constructing and manipulating addresses and network numbers in this form.

When compiling multi-threaded applications, see Intro(3), *Notes On Multithread Applications*, for information about the use of the _REENTRANT flag.

Use of the enumeration interfaces getnetent() and getnetent_r() is discouraged; enumeration may not be supported for all database sources. The semantics of enumeration are discussed further in nsswitch.conf(4).

Name   getnetconfig, setnetconfig, endnetconfig, getnetconfigent, freenetconfigent, nc_perror, nc_sperror – get network configuration database entry

Synopsis   `#include <netconfig.h>`

`struct netconfig *getnetconfig(void *`*handlep*`);`

`void *setnetconfig(void);`

`int endnetconfig(void *`*handlep*`);`

`struct netconfig *getnetconfigent(const char *`*netid*`);`

`void freenetconfigent(struct netconfig *`*netconfigp*`);`

`void nc_perror(const char *`*msg*`);`

`char *nc_sperror(void);`

Description   The library routines described on this page are part of the Network Selection component. They provide the application access to the system network configuration database, /etc/netconfig. In addition to the routines for accessing the netconfig database, Network Selection includes the environment variable NETPATH (see environ(5)) and the NETPATH access routines described in getnetpath(3NSL).

getnetconfig() returns a pointer to the current entry in the netconfig database, formatted as a struct netconfig. Successive calls will return successive netconfig entries in the netconfig database. getnetconfig() can be used to search the entire netconfig file. getnetconfig() returns NULL at the end of the file. *handlep* is the handle obtained through setnetconfig().

A call to setnetconfig() has the effect of "binding" to or "rewinding" the netconfig database. setnetconfig() must be called before the first call to getnetconfig() and may be called at any other time. setnetconfig() need *not* be called before a call to getnetconfigent(). setnetconfig() returns a unique handle to be used by getnetconfig().

endnetconfig() should be called when processing is complete to release resources for reuse. *handlep* is the handle obtained through setnetconfig(). Programmers should be aware, however, that the last call to endnetconfig() frees all memory allocated by getnetconfig() for the struct netconfig data structure. endnetconfig() may not be called before setnetconfig().

getnetconfigent() returns a pointer to the struct netconfig structure corresponding to *netid*. It returns NULL if *netid* is invalid (that is, does not name an entry in the netconfig database).

freenetconfigent() frees the netconfig structure pointed to by *netconfigp* (previously returned by getnetconfigent()).

nc_perror() prints a message to the standard error indicating why any of the above routines failed. The message is prepended with the string *msg* and a colon. A NEWLINE is appended at the end of the message.

nc_sperror() is similar to nc_perror() but instead of sending the message to the standard error, will return a pointer to a string that contains the error message.

nc_perror() and nc_sperror() can also be used with the NETPATH access routines defined in getnetpath(3NSL).

Return Values setnetconfig() returns a unique handle to be used by getnetconfig(). In the case of an error, setnetconfig() returns NULL and nc_perror() or nc_sperror() can be used to print the reason for failure.

getnetconfig() returns a pointer to the current entry in the netconfig() database, formatted as a struct netconfig. getnetconfig() returns NULL at the end of the file, or upon failure.

endnetconfig() returns 0 on success and −1 on failure (for example, if setnetconfig() was not called previously).

On success, getnetconfigent() returns a pointer to the struct netconfig structure corresponding to *netid*; otherwise it returns NULL.

nc_sperror() returns a pointer to a buffer which contains the error message string. This buffer is overwritten on each call. In multithreaded applications, this buffer is implemented as thread-specific data.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also getnetpath(3NSL), netconfig(4), attributes(5), environ(5)

Name getnetpath, setnetpath, endnetpath – get /etc/netconfig entry corresponding to NETPATH component

Synopsis #include <netconfig.h>

```
struct netconfig *getnetpath(void *handlep);

void *setnetpath(void);

int endnetpath(void *handlep);
```

Description The routines described on this page are part of the Network Selection component. They provide the application access to the system network configuration database, /etc/netconfig, as it is "filtered" by the NETPATH environment variable. See environ(5). See getnetconfig(3NSL) for other routines that also access the network configuration database directly. The NETPATH variable is a list of colon-separated network identifiers.

getnetpath() returns a pointer to the netconfig database entry corresponding to the first valid NETPATH component. The netconfig entry is formatted as a struct netconfig. On each subsequent call, getnetpath() returns a pointer to the netconfig entry that corresponds to the next valid NETPATH component. getnetpath() can thus be used to search the netconfig database for all networks included in the NETPATH variable. When NETPATH has been exhausted, getnetpath() returns NULL.

A call to setnetpath() "binds" to or "rewinds" NETPATH. setnetpath() must be called before the first call to getnetpath() and may be called at any other time. It returns a handle that is used by getnetpath().

getnetpath() silently ignores invalid NETPATH components. A NETPATH component is invalid if there is no corresponding entry in the netconfig database.

If the NETPATH variable is unset, getnetpath() behaves as if NETPATH were set to the sequence of "default" or "visible" networks in the netconfig database, in the order in which they are listed.

endnetpath() may be called to "unbind" from NETPATH when processing is complete, releasing resources for reuse. Programmers should be aware, however, that endnetpath() frees all memory allocated by getnetpath() for the struct netconfig data structure. endnetpath() returns 0 on success and -1 on failure (for example, if setnetpath() was not called previously).

Return Values setnetpath() returns a handle that is used by getnetpath(). In case of an error, setnetpath() returns NULL. nc_perror() or nc_sperror() can be used to print out the reason for failure. See getnetconfig(3NSL).

When first called, getnetpath() returns a pointer to the netconfig database entry corresponding to the first valid NETPATH component. When NETPATH has been exhausted, getnetpath() returns NULL.

endnetpath() returns 0 on success and -1 on failure (for example, if setnetpath() was not called previously).

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also getnetconfig(3NSL), netconfig(4), attributes(5), environ(5)

Name getpeername – get name of connected peer

Synopsis cc [*flag* ...] *file* ... -lsocket -lnsl [*library* ...]
#include <sys/socket.h>

int getpeername(int *s*, struct sockaddr **name*, socklen_t **namelen*);

Description getpeername() returns the name of the peer connected to socket *s*. The int pointed to by the
namelen parameter should be initialized to indicate the amount of space pointed to by *name*.
On return it contains the actual size of the name returned (in bytes), prior to any truncation.
The name is truncated if the buffer provided is too small.

Return Values If successful, getpeername() returns 0; otherwise it returns −1 and sets errno to indicate the
error.

Errors The call succeeds unless:

 EBADF   The argument *s* is not a valid descriptor.

 ENOMEM   There was insufficient user memory for the operation to complete.

 ENOSR   There were insufficient STREAMS resources available for the operation to
                complete.

 ENOTCONN   The socket is not connected.

 ENOTSOCK   The argument *s* is not a socket.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also accept(3SOCKET), bind(3SOCKET), getsockname(3SOCKET), socket(3SOCKET),
attributes(5), socket.h(3HEAD)

Name getpeername – get the name of the peer socket

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <sys/socket.h>

int getpeername(int *socket*, struct sockaddr *restrict *address*,
    socklen_t *restrict *address_len*);

Description The getpeername() function retrieves the peer address of the specified socket, stores this
address in the sockaddr structure pointed to by the *address* argument, and stores the length of
this address in the object pointed to by the *address_len* argument.

If the actual length of the address is greater than the length of the supplied sockaddr structure,
the stored address will be truncated.

If the protocol permits connections by unbound clients, and the peer is not bound, then the
value stored in the object pointed to by *address* is unspecified.

Return Values Upon successful completion, 0 is returned. Otherwise, −1 is returned and errno is set to
indicate the error.

Errors The getpeername() function will fail if:

EBADF   The *socket* argument is not a valid file descriptor.

EFAULT   The *address* or *address_len* parameter can not be accessed or written.

EINVAL   The socket has been shut down.

ENOTCONN   The socket is not connected or otherwise has not had the peer prespecified.

ENOTSOCK   The *socket* argument does not refer to a socket.

EOPNOTSUPP   The operation is not supported for the socket protocol.

The getpeername() function may fail if:

ENOBUFS   Insufficient resources were available in the system to complete the call.

ENOSR   There were insufficient STREAMS resources available for the operation to
            complete.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also   accept(3XNET), bind(3XNET), getsockname(3XNET), socket(3XNET), attributes(5), standards(5)

Name getprotobyname, getprotobyname_r, getprotobynumber, getprotobynumber_r, getprotoent, getprotoent_r, setprotoent, endprotoent – get protocol entry

Synopsis cc [*flag* ...] *file* ... -lsocket -lnsl [*library* ...]
#include <netdb.h>

struct protoent *getprotobyname(const char *name);

struct protoent *getprotobyname_r(const char *name,
    struct protoent *result, char *buffer,
    int buflen);

struct protoent *getprotobynumber(int proto);

struct protoent *getprotobynumber_r(int proto, struct protoent *result,
    char *buffer, int buflen);

struct protoent *getprotoent(void);

struct protoent *getprotoent_r(struct protoent *result, char *buffer,
    int buflen);

int setprotoent(int stayopen);

int endprotoent(void);

Description These functions return a protocol entry. Two types of interfaces are supported: reentrant (getprotobyname_r(), getprotobynumber_r(), and getprotoent_r()) and non-reentrant (getprotobyname(), getprotobynumber(), and getprotoent()). The reentrant functions can be used in single-threaded applications and are safe for multithreaded applications, making them the preferred interfaces.

The reentrant routines require additional parameters which are used to return results data. *result* is a pointer to a struct protoent structure and will be where the returned results will be stored. *buffer* is used as storage space for elements of the returned results. *buflen* is the size of *buffer* and should be large enough to contain all returned data. *buflen* must be at least 1024 bytes.

getprotobyname_r(), getprotobynumber_r(), and getprotoent_r() each return a protocol entry.

The entry may come from one of the following sources: the protocols file (see protocols(4)), and the NIS maps "protocols.byname" and "protocols.bynumber". The sources and their lookup order are specified in the /etc/nsswitch.conf file (see nsswitch.conf(4) for details). Some name services such as NIS will return only one name for a host, whereas others such as DNS will return all aliases.

The getprotobyname_r() and getprotobynumber_r() functions sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until an EOF is encountered.

getprotobyname() and getprotobynumber() have the same functionality as getprotobyname_r() and getprotobynumber_r() except that a static buffer is used to store returned results. These functions are Unsafe in a multithreaded application.

getprotoent_r() enumerates protocol entries: successive calls to getprotoent_r() will return either successive protocol entries or NULL. Enumeration might not be supported by some sources. If multiple threads call getprotoent_r(), each will retrieve a subset of the protocol database.

getprotent() has the same functionality as getprotent_r() except that a static buffer is used to store returned results. This routine is unsafe in a multithreaded application.

setprotoent() "rewinds" to the beginning of the enumeration of protocol entries. If the *stayopen* flag is non-zero, resources such as open file descriptors are not deallocated after each call to getprotobynumber_r() and getprotobyname_r(). Calls to getprotobyname_r() , The getprotobyname(), getprotobynumber_r(), and getprotobynumber() functions might leave the enumeration in an indeterminate state, so setprotoent() should be called before the first call to getprotoent_r() or getprotoent(). The setprotoent() function has process-wide scope, and "rewinds" the protocol entries for all threads calling getprotoent_r() as well as main-thread calls to getprotoent().

The endprotoent() function can be called to indicate that protocol processing is complete; the system may then close any open protocols file, deallocate storage, and so forth. It is legitimate, but possibly less efficient, to call more protocol functions after endprotoent().

The internal representation of a protocol entry is a protoent structure defined in <netdb.h> with the following members:

```
char   *p_name;
char   **p_aliases;
int    p_proto;
```

Return Values The getprotobyname_r(), getprotobyname(), getprotobynumber_r(), and getprotobynumber() functions return a pointer to a struct protoent if they successfully locate the requested entry; otherwise they return NULL.

The getprotoent_r() and getprotoent() functions return a pointer to a struct protoent if they successfully enumerate an entry; otherwise they return NULL, indicating the end of the enumeration.

Errors The getprotobyname_r(), getprotobynumber_r(), and getprotoent_r() functions will fail if:

ERANGE   The length of the buffer supplied by the caller is not large enough to store the result.

Files /etc/protocols

/etc/nsswitch.conf

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	See NOTES below.

See Also Intro(3), nsswitch.conf(4), protocols(4), attributes(5), netdb.h(3HEAD)

Notes Although getprotobyname_r(), getprotobynumber_r(), and getprotoent_r() are not mentioned by POSIX 1003.1:2001, they were added to complete the functionality provided by similar thread-safe functions.

When compiling multithreaded applications, see Intro(3), *Notes On Multithread Applications*, for information about the use of the _REENTRANT flag.

The getprotobyname_r(), getprotobynumber_r(), and getprotoent_r() functions are reentrant and multithread safe. The reentrant interfaces can be used in single-threaded as well as multithreaded applications and are therefore the preferred interfaces.

The getprotobyname(), getprotobyaddr(), and getprotoent() functions use static storage, so returned data must be copied if it is to be saved. Because of their use of static storage for returned data, these functions are not safe for multithreaded applications.

The setprotoent() and endprotoent() functions have process-wide scope, and are therefore not safe in multi-threaded applications.

Use of getprotoent_r() and getprotoent() is discouraged; enumeration is well-defined for the protocols file and is supported (albeit inefficiently) for NIS, but in general may not be well-defined. The semantics of enumeration are discussed in nsswitch.conf(4).

Bugs Only the Internet protocols are currently understood.

Name getpublickey, getsecretkey, publickey – retrieve public or secret key

Synopsis #include <rpc/rpc.h>
#include <rpc/key_prot.h>

int getpublickey(const char *netname[MAXNETNAMELEN]*,
  char *publickey[HEXKEYBYTES+1]*);

int getsecretkey(const char *netname[MAXNETNAMELEN]*,
  char *secretkey[HEXKEYBYTES+1]*,const char *passwd*);

Description The getpublickey() and getsecretkey() functions get public and secret keys for *netname*. The key may come from one of the following sources:

- /etc/publickey file. See publickey(4).

- NIS map "publickey.byname". The sources and their lookup order are specified in the /etc/nsswitch.conf file. See nsswitch.conf(4).

getsecretkey() has an extra argument, passwd, which is used to decrypt the encrypted secret key stored in the database.

Return Values Both routines return 1 if they are successful in finding the key. Otherwise, the routines return 0. The keys are returned as null-terminated, hexadecimal strings. If the password supplied to getsecretkey() fails to decrypt the secret key, the routine will return 1 but the *secretkey* [0] will be set to NULL.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also secure_rpc(3NSL), nsswitch.conf(4), publickey(4), attributes(5)

Name getrpcbyname, getrpcbyname_r, getrpcbynumber, getrpcbynumber_r, getrpcent, getrpcent_r, setrpcent, endrpcent – get RPC entry

Synopsis cc [*flag* ...] *file* ... -lnsl [*library* ...]
#include <rpc/rpcent.h>

struct rpcent *getrpcbyname(const char **name*);

struct rpcent *getrpcbyname_r(const char **name*, struct rpcent **result*,
    char **buffer*, int *buflen*);

struct rpcent *getrpcbynumber(const int *number*);

struct rpcent *getrpcbynumber_r(const int *number*, struct rpcent **result*,
    char **buffer*, int *buflen*);

struct rpcent *getrpcent(void);

struct rpcent *getrpcent_r(struct rpcent **result*, char **buffer*,
    int *buflen*);

void setrpcent(const int *stayopen*);

void endrpcent(void);

Description These functions are used to obtain entries for RPC (Remote Procedure Call) services. An entry may come from any of the sources for rpc specified in the /etc/nsswitch.conf file (see nsswitch.conf(4)).

getrpcbyname() searches for an entry with the RPC service name specified by the parameter *name*.

getrpcbynumber() searches for an entry with the RPC program number *number*.

The functions setrpcent(), getrpcent(), and endrpcent() are used to enumerate RPC entries from the database.

setrpcent() sets (or resets) the enumeration to the beginning of the set of RPC entries. This function should be called before the first call to getrpcent(). Calls to getrpcbyname() and getrpcbynumber() leave the enumeration position in an indeterminate state. If the *stayopen* flag is non-zero, the system may keep allocated resources such as open file descriptors until a subsequent call to endrpcent().

Successive calls to getrpcent() return either successive entries or NULL, indicating the end of the enumeration.

endrpcent() may be called to indicate that the caller expects to do no further RPC entry retrieval operations; the system may then deallocate resources it was using. It is still allowed, but possibly less efficient, for the process to call more RPC entry retrieval functions after calling endrpcent().

Reentrant Interfaces   The functions getrpcbyname(), getrpcbynumber(), and getrpcent() use static storage that is re-used in each call, making these routines unsafe for use in multithreaded applications.

The functions getrpcbyname_r(), getrpcbynumber_r(), and getrpcent_r() provide reentrant interfaces for these operations.

Each reentrant interface performs the same operation as its non-reentrant counterpart, named by removing the "_r" suffix. The reentrant interfaces, however, use buffers supplied by the caller to store returned results, and are safe for use in both single-threaded and multithreaded applications.

Each reentrant interface takes the same parameters as its non-reentrant counterpart, as well as the following additional parameters. The parameter *result* must be a pointer to a struct rpcent structure allocated by the caller. On successful completion, the function returns the RPC entry in this structure. The parameter *buffer* must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the RPC entry data. All of the pointers within the returned struct rpcent *result* point to data stored within this buffer (see RETURN VALUES). The buffer must be large enough to hold all of the data associated with the RPC entry. The parameter *buflen* should give the size in bytes of the buffer indicated by *buffer*.

For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. setrpcent() may be used in a multithreaded application but resets the enumeration position for all threads. If multiple threads interleave calls to getrpcent_r(), the threads will enumerate disjoint subsets of the RPC entry database.

Like their non-reentrant counterparts, getrpcbyname_r() and getrpcbynumber_r() leave the enumeration position in an indeterminate state.

Return Values   RPC entries are represented by the struct rpcent structure defined in <rpc/rpcent.h>:

```
struct rpcent {
   char *r_name;        /* name of this rpc service
   char **r_aliases;    /* zero-terminated list of alternate names */
   int r_number;        /* rpc program number */
};
```

The functions getrpcbyname(), getrpcbyname_r( ), getrpcbynumber( ), and getrpcbynumber_r() each return a pointer to a struct rpcent if they successfully locate the requested entry; otherwise they return NULL.

The functions getrpcent() and getrpcent_r() each return a pointer to a struct rpcent if they successfully enumerate an entry; otherwise they return NULL, indicating the end of the enumeration.

The functions getrpcbyname(), getrpcbynumber(), and getrpcent() use static storage, so returned data must be copied before a subsequent call to any of these functions if the data is to be saved.

When the pointer returned by the reentrant functions getrpcbyname_r(),
getrpcbynumber_r(), and getrpcent_r() is non-NULL, it is always equal to the *result*
pointer that was supplied by the caller.

Errors The reentrant functions getrpcyname_r(), getrpcbynumber_r( ) and getrpcent_r() will
return NULL and set errno to ERANGE if the length of the buffer supplied by caller is not large
enough to store the result. See Intro(2) for the proper usage and interpretation of errno in
multithreaded applications.

Files /etc/rpc

/etc/nsswitch.conf

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	See "Reentrant Interfaces" in DESCRIPTION.

See Also rpcinfo(1M), rpc(3NSL), nsswitch.conf(4), rpc(4), attributes(5)

Warnings The reentrant interfaces getrpcbyname_r(), getrpcbynumber_r(), and getrpcent_r() are
included in this release on an uncommitted basis only, and are subject to change or removal in
future minor releases.

Notes When compiling multithreaded applications, see Intro(3), *Notes On Multithreaded
Applications*, for information about the use of the _REENTRANT flag.

Use of the enumeration interfaces getrpcent() and getrpcent_r() is discouraged;
enumeration may not be supported for all database sources. The semantics of enumeration are
discussed further in nsswitch.conf(4).

Name getservbyname, getservbyname_r, getservbyport, getservbyport_r, getservent, getservent_r, setservent, endservent – get service entry

Synopsis cc [*flag* ...] *file* ... -lsocket -lnsl [*library* ...]
#include <netdb.h>

struct servent *getservbyname(const char **name*, const char **proto*);

struct servent *getservbyname_r(const char **name*, const char **proto*,
    struct servent **result*, char **buffer*, int *buflen*);

struct servent *getservbyport(int *port*, const char **proto*);

struct servent *getservbyport_r(int *port*, const char **proto*,
    struct servent **result*, char **buffer*, int *buflen*);

struct servent *getservent(void);

struct servent *getservent_r(struct servent **result*, char **buffer*,
    int *buflen*);

int setservent(int *stayopen*);

int endservent(void);

Description These functions are used to obtain entries for Internet services. An entry may come from any of the sources for services specified in the /etc/nsswitch.conf file. See nsswitch.conf(4).

The getservbyname() and getservbyport() functions sequentially search from the beginning of the file until a matching protocol name or port number is found, or until end-of-file is encountered. If a protocol name is also supplied (non-null), searches must also match the protocol.

The getservbyname() function searches for an entry with the Internet service name specified by the *name* parameter.

The getservbyport() function searches for an entry with the Internet port number *port*.

All addresses are returned in network order. In order to interpret the addresses, byteorder(3SOCKET) must be used for byte order conversion. The string *proto* is used by both getservbyname() and getservbyport() to restrict the search to entries with the specified protocol. If *proto* is NULL, entries with any protocol can be returned.

The functions setservent(), getservent(), and endservent() are used to enumerate entries from the services database.

The setservent() function sets (or resets) the enumeration to the beginning of the set of service entries. This function should be called before the first call to getservent(). Calls to the functions getservbyname() and getservbyport() leave the enumeration position in an indeterminate state. If the *stayopen* flag is non-zero, the system may keep allocated resources such as open file descriptors until a subsequent call to endservent().

The getservent() function reads the next line of the file, opening the file if necessary. getservent() opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to getservent() (either directly, or indirectly through one of the other "getserv"calls).

Successive calls to getservent() return either successive entries or NULL, indicating the end of the enumeration.

The endservent() function closes the file. The endservent() function can be called to indicate that the caller expects to do no further service entry retrieval operations; the system can then deallocate resources it was using. It is still allowed, but possibly less efficient, for the process to call more service entry retrieval functions after calling endservent().

Reentrant Interfaces   The functions getservbyname(), getservbyport(), and getservent() use static storage that is re-used in each call, making these functions unsafe for use in multithreaded applications.

The functions getservbyname_r(), getservbyport_r(), and getservent_r() provide reentrant interfaces for these operations.

Each reentrant interface performs the same operation as its non-reentrant counterpart, named by removing the "_r" suffix. The reentrant interfaces, however, use buffers supplied by the caller to store returned results, and are safe for use in both single-threaded and multithreaded applications.

Each reentrant interface takes the same parameters as its non-reentrant counterpart, as well as the following additional parameters. The parameter *result* must be a pointer to a struct servent structure allocated by the caller. On successful completion, the function returns the service entry in this structure. The parameter *buffer* must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the service entry data. All of the pointers within the returned struct servent *result* point to data stored within this buffer. See the RETURN VALUES section of this manual page. The buffer must be large enough to hold all of the data associated with the service entry. The parameter *buflen* should give the size in bytes of the buffer indicated by *buffer*.

For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. The setservent() function can be used in a multithreaded application but resets the enumeration position for all threads. If multiple threads interleave calls to getservent_r(), the threads will enumerate disjoint subsets of the service database.

Like their non-reentrant counterparts, getservbyname_r() and getservbyport_r() leave the enumeration position in an indeterminate state.

Return Values   Service entries are represented by the struct servent structure defined in <netdb.h>:

```
struct  servent {
    char    *s_name;                /* official name of service */
    char    **s_aliases;             /* alias list */
```

```
    int    s_port;                /* port service resides at */
    char   *s_proto;               /* protocol to use */
};
```

The members of this structure are:

s_name   The official name of the service.

s_aliases   A zero terminated list of alternate names for the service.

s_port   The port number at which the service resides. Port numbers are returned in
            network byte order.

s_proto   The name of the protocol to use when contacting the service

The functions getservbyname(), getservbyname_r(), getservbyport(), and
getservbyport_r() each return a pointer to a struct servent if they successfully locate the
requested entry; otherwise they return NULL.

The functions getservent() and getservent_r() each return a pointer to a struct servent
if they successfully enumerate an entry; otherwise they return NULL, indicating the end of the
enumeration.

The functions getservbyname(), getservbyport(), and getservent() use static storage, so
returned data must be copied before a subsequent call to any of these functions if the data is to
be saved.

When the pointer returned by the reentrant functions getservbyname_r(),
getservbyport_r(), and getservent_r() is non-null, it is always equal to the *result* pointer
that was supplied by the caller.

Errors   The reentrant functions getservbyname_r(), getservbyport_r(), and getservent_r()
return NULL and set errno to ERANGE if the length of the buffer supplied by caller is not large
enough to store the result. See Intro(2) for the proper usage and interpretation of errno in
multithreaded applications.

Files   /etc/services   Internet network services

/etc/netconfig   network configuration file

/etc/nsswitch.conf   configuration file for the name-service switch

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	See "Reentrant Interfaces" in DESCRIPTION.

See Also Intro(2), Intro(3), byteorder(3SOCKET), netdir(3NSL), netconfig(4), nsswitch.conf(4), services(4), attributes(5), netdb.h(3HEAD)

Warnings The reentrant interfaces getservbyname_r(), getservbyport_r(), and getservent_r() are included in this release on an uncommitted basis only, and are subject to change or removal in future minor releases.

Notes The functions that return struct servent return the least significant 16-bits of the *s_port* field in *network byte order*. getservbyport() and getservbyport_r() also expect the input parameter *port* in the *network byte order*. See htons(3SOCKET) for more details on converting between host and network byte orders.

To ensure that they all return consistent results, getservbyname(), getservbyname_r(), and netdir_getbyname() are implemented in terms of the same internal library function. This function obtains the system-wide source lookup policy based on the inet family entries in netconfig(4) and the services: entry in nsswitch.conf(4). Similarly, getservbyport(), getservbyport_r(), and netdir_getbyaddr() are implemented in terms of the same internal library function. If the inet family entries in netconfig(4) have a "-" in the last column for nametoaddr libraries, then the entry for services in nsswitch.conf will be used; otherwise the nametoaddr libraries in that column will be used, and nsswitch.conf will not be consulted.

There is no analogue of getservent() and getservent_r() in the netdir functions, so these enumeration functions go straight to the services entry in nsswitch.conf. Thus enumeration may return results from a different source than that used by getservbyname(), getservbyname_r(), getservbyport(), and getservbyport_r().

When compiling multithreaded applications, see Intro(3), *Notes On Multithread Applications*, for information about the use of the _REENTRANT flag.

Use of the enumeration interfaces getservent() and getservent_r() is discouraged; enumeration may not be supported for all database sources. The semantics of enumeration are discussed further in nsswitch.conf(4).

Name   getsockname – get socket name

Synopsis   cc [*flag* ...] *file* ... -lsocket -lnsl [*library* ...]
#include <sys/socket.h>

int getsockname(int *s*, struct sockaddr **name*, socklen_t **namelen*);

Description   getsockname() returns the current *name* for socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size in bytes of the name returned.

Return Values   If successful, getsockname() returns 0; otherwise it returns −1 and sets *errno* to indicate the error.

Errors   The call succeeds unless:

EBADF   The argument *s* is not a valid file descriptor.

ENOMEM   There was insufficient memory available for the operation to complete.

ENOSR   There were insufficient STREAMS resources available for the operation to complete.

ENOTSOCK   The argument *s* is not a socket.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   bind(3SOCKET), getpeername(3SOCKET), socket(3SOCKET), attributes(5)

Name getsockname – get the socket name

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <sys/socket.h>

int getsockname(int *socket*, struct sockaddr *restrict *address*,
    socklen_t *restrict *address_len*);

Description The getsockname() function retrieves the locally-bound name of the specified socket, stores this address in the sockaddr structure pointed to by the *address* argument, and stores the length of this address in the object pointed to by the *address_len* argument.

If the actual length of the address is greater than the length of the supplied sockaddr structure, the stored address will be truncated.

If the socket has not been bound to a local name, the value stored in the object pointed to by *address* is unspecified.

Return Values Upon successful completion, 0 is returned, the *address* argument points to the address of the socket, and the *address_len* argument points to the length of the address. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The getsockname() function will fail:

EBADF   The *socket* argument is not a valid file descriptor.

EFAULT   The *address* or *address_len* parameter can not be accessed or written.

ENOTSOCK   The *socket* argument does not refer to a socket.

EOPNOTSUPP   The operation is not supported for this socket's protocol.

The getsockname() function may fail if:

EINVAL   The socket has been shut down.

ENOBUFS   Insufficient resources were available in the system to complete the call.

ENOSR   There were insufficient STREAMS resources available for the operation to complete.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also   accept(3XNET), bind(3XNET), getpeername(3XNET), socket(3XNET) attributes(5), standards(5)

Name getsockopt, setsockopt – get and set options on sockets

Synopsis cc [*flag* ...] *file* ... -lsocket -lnsl [*library* ...]
#include <sys/socket.h>

int getsockopt(int *s*, int *level*, int *optname*, void **optval*,
    socklen_t **optlen*);

int setsockopt(int *s*, int *level*, int *optname*, const void **optval*,
    socklen_t *optlen*);

Description The getsockopt() and setsockopt() functions manipulate options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level.

The *level* argument specifies the protocol level at which the option resides. To manipulate options at the socket level, specify the *level* argument as SOL_SOCKET. To manipulate options at the protocol level, supply the appropriate protocol number for the protocol controlling the option. For example, to indicate that an option will be interpreted by the TCP, set *level* to the protocol number of TCP, as defined in the <netinet/in.h> header, or as determined by using [getprotobyname(3SOCKET)](). Some socket protocol families may also define additional levels, such as SOL_ROUTE. Only socket-level options are described here.

The parameters *optval* and *optlen* are used to access option values for setsockopt(). For getsockopt(), they identify a buffer in which the value(s) for the requested option(s) are to be returned. For getsockopt(), *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. Use a 0 *optval* if no option value is to be supplied or returned.

The *optname* and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file <sys/socket.h> contains definitions for the socket-level options described below. Options at other protocol levels vary in format and name.

Most socket-level options take an int for *optval*. For setsockopt(), the *optval* parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. SO_LINGER uses a struct linger parameter that specifies the desired state of the option and the linger interval. struct linger is defined in <sys/socket.h>. struct linger contains the following members:

l_onoff   on = 1/off = 0

l_linger   linger time, in seconds

The following options are recognized at the socket level. Except as noted, each may be examined with getsockopt() and set with setsockopt().

SO_DEBUG   enable/disable recording of debugging information

SO_REUSEADDR   enable/disable local address reuse

SO_REUSEPORT	enable/disable local port reuse for PF_INET/PF_INET6 socket
SO_KEEPALIVE	enable/disable keep connections alive
SO_DONTROUTE	enable/disable routing bypass for outgoing messages
SO_LINGER	linger on close if data is present
SO_BROADCAST	enable/disable permission to transmit broadcast messages
SO_OOBINLINE	enable/disable reception of out-of-band data in band
SO_SNDBUF	set buffer size for output
SO_RCVBUF	set buffer size for input
SO_DGRAM_ERRIND	application wants delayed error
SO_TIMESTAMP	enable/disable reception of timestamp with datagrams
SO_EXCLBIND	enable/disable exclusive binding of the socket
SO_TYPE	get the type of the socket (get only)
SO_ERROR	get and clear error on the socket (get only)
SO_MAC_EXEMPT	get or set mandatory access control on the socket. This option is available only when the system is configured with Trusted Extensions.
SO_ALLZONES	bypass zone boundaries (privileged).
SO_DOMAIN	get the domain used in the socket (get only)
SO_PROTOTYPE	for socket in domains PF_INET and PF_INET6, get the underlying protocol number used in the socket. For socket in domain PF_ROUTE, get the address family used in the socket.
SO_PASSIVE_CONNECT	modify connect(3SOCKET) to wait for connection request from a peer instead of initiating a connection request to it. It is applicable to TCP/SCTP PF_INET/PF_INET6 socket.

The SO_DEBUG option enables debugging in the underlying protocol modules. The SO_REUSEADDR/SO_REUSEPORT options indicate that the rules used in validating addresses and ports supplied in a bind(3SOCKET) call should allow reuse of local addresses or ports. The SO_KEEPALIVE option enables the periodic transmission of messages on a connected socket. If the connected party fails to respond to these messages, the connection is considered broken and threads using the socket are notified using a SIGPIPE signal. The SO_DONTROUTE option indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

The SO_LINGER option controls the action taken when unsent messages are queued on a socket and a close(2) is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the thread on the close() attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the setsockopt() call when SO_LINGER is requested). If SO_LINGER is disabled and a close() is issued, the system will process the close() in a manner that allows the thread to continue as quickly as possible.

The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with recv() or read() calls without the MSG_OOB flag.

The SO_SNDBUF and SO_RCVBUF options adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections or may be decreased to limit the possible backlog of incoming data. The maximum buffer size for UDP/TCP is determined by the value of the ipadm variable *max_buf* for that particular protocol. Use the ipadm(1M) utility to determine the current default values. See the *Solaris Tunable Parameters Reference Manual* for information on setting the values of max_buf for either TCP, UDP or both. At present, lowering SO_RCVBUF on a TCP connection after it has been established has no effect.

By default, delayed errors (such as ICMP port unreachable packets) are returned only for connected datagram sockets. The SO_DGRAM_ERRIND option makes it possible to receive errors for datagram sockets that are not connected. When this option is set, certain delayed errors received after completion of a sendto() or sendmsg() operation will cause a subsequent sendto() or sendmsg() operation using the same destination address (*to* parameter) to fail with the appropriate error. See send(3SOCKET).

If the SO_TIMESTAMP option is enabled on a SO_DGRAM or a SO_RAW socket, the recvmsg(3XNET) call will return a timestamp in the native data format, corresponding to when the datagram was received.

The SO_EXCLBIND option is used to enable or disable the exclusive binding of a socket. It overrides the use of the SO_REUSEADDR option to reuse an address on bind(3SOCKET). The actual semantics of the SO_EXCLBIND option depend on the underlying protocol. See tcp(7P) or udp(7P) for more information.

The SO_TYPE and SO_ERROR options are used only with getsockopt(). The SO_TYPE option returns the type of the socket, for example, SOCK_STREAM. It is useful for servers that inherit sockets on startup. The SO_ERROR option returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

The SO_MAC_EXEMPT option is used to toggle socket behavior with unlabeled peers. A socket that has this option enabled can communicate with an unlabeled peer if it is in the global zone or has a label that dominates the default label of the peer. Otherwise, the socket must have a label that is equal to the default label of the unlabeled peer. Calling setsockopt() with this option returns an EACCES error if the process lacks the NET_MAC_AWARE privilege or if the socket is bound. The SO_MAC_EXEMPT option is available only when the system is configured with Trusted Extensions.

The SO_ALLZONES option can be used to bypass zone boundaries between shared-IP zones. Normally, the system prevents a socket from being bound to an address that is not assigned to the current zone. It also prevents a socket that is bound to a wildcard address from receiving traffic for other zones. However, some daemons which run in the global zone might need to send and receive traffic using addresses that belong to other shared-IP zones. If set before a socket is bound, SO_ALLZONES causes the socket to ignore zone boundaries between shared-IP zones and permits the socket to be bound to any address assigned to the shared-IP zones. If the socket is bound to a wildcard address, it receives traffic intended for all shared-IP zones and behaves as if an equivalent socket were bound in each active shared-IP zone. Applications that use the SO_ALLZONES option to initiate connections or send datagram traffic should specify the source address for outbound traffic by binding to a specific address. There is no effect from setting this option in an exclusive-IP zone. Setting this option requires the sys_net_config privilege. See zones(5).

The SO_PASSIVE_CONNECT option can be used to modify connect() semantics for TCP and SCTP socket. After this option is set, calling connect() on the socket will not initiate a connection setup sequence. Instead, the transport end point is in listen state waiting for a connection request from the remote peer specified in connect(). After the expected connection is established, connect() returns.

Return Values If successful, getsockopt() and setsockopt() return 0. Otherwise, the functions return −1 and set errno to indicate the error.

Errors The getsockopt() and setsockopt() calls succeed unless:

EBADF	The argument *s* is not a valid file descriptor.
EACCES	Permission denied.
EADDRINUSE	Address already joined for IP_ADD_MEMBERSHIP.
EADDRNOTAVAIL	Bad interface address for IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP.
EHOSTUNREACH	Invalid address for IP_MULTICAST_IF.
EINVAL	Invalid length for IP_OPTIONS.
	Not a multicast address for IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP.

	The specified option is invalid at the specified socket level, or the socket has been shut down.
ENOBUFS	SO_SNDBUF or SO_RCVBUF exceeds a system limit.
ENOENT	Address not joined for IP_DROP_MEMBERSHIP.
ENOMEM	There was insufficient memory available for the operation to complete.
ENOPROTOOPT	The option is unknown at the level indicated.
ENOSR	There were insufficient STREAMS resources available for the operation to complete.
ENOTSOCK	The argument *s* is not a socket.
EPERM	No permissions.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   ipadm(1M), close(2), ioctl(2), read(2), bind(3SOCKET), connect(3SOCKET), getprotobyname(3SOCKET), recv(3SOCKET), recvmsg(3XNET), send(3SOCKET), socket(3SOCKET), socket.h(3HEAD), attributes(5), zones(5), tcp(7P), udp(7P)

Solaris Tunable Parameters Reference Manual

Name getsockopt – get the socket options

Synopsis cc [*flag*...] *file*... -lxnet [*library*...]
#include <sys/socket.h>

int getsockopt(int *socket*, int *level*, int *option_name*,
    void *restrict *option_value*, socklen_t *restrict *option_len*);

Description The getsockopt() function retrieves the value for the option specified by the *option_name*
argument for the socket specified by the *socket* argument. If the size of the option value is
greater than *option_len*, the value stored in the object pointed to by the *option_value*
argument will be silently truncated. Otherwise, the object pointed to by the *option_len*
argument will be modified to indicate the actual length of the value.

The *level* argument specifies the protocol level at which the option resides. To retrieve options
at the socket level, specify the *level* argument as SOL_SOCKET. To retrieve options at other
levels, supply the appropriate protocol number for the protocol controlling the option. For
example, to indicate that an option will be interpreted by the TCP (Transport Control
Protocol), set *level* to the protocol number of TCP, as defined in the <netinet/in.h> header,
or as determined by using getprotobyname(3XNET) function.

The socket in use might require the process to have appropriate privileges to use the
getsockopt() function.

The *option_name* argument specifies a single option to be retrieved. It can be one of the
following values defined in <sys/socket.h>:

SO_DEBUG   Reports whether debugging information is being recorded. This option
                stores an int value. This is a boolean option.

SO_ACCEPTCONN   Reports whether socket listening is enabled. This option stores an int
                value.

SO_BROADCAST   Reports whether transmission of broadcast messages is supported, if this
                is supported by the protocol. This option stores an int value. This is a
                boolean option.

SO_REUSEADDR   Reports whether the rules used in validating addresses supplied to
                bind(3XNET) should allow reuse of local addresses, if this is supported by
                the protocol. This option stores an int value. This is a boolean option.

SO_KEEPALIVE   Reports whether connections are kept active with periodic transmission
                of messages, if this is supported by the protocol.

                If the connected socket fails to respond to these messages, the connection
                is broken and threads writing to that socket are notified with a SIGPIPE
                signal. This option stores an int value.

                This is a boolean option.

SO_LINGER	Reports whether the socket lingers on close(2) if data is present. If SO_LINGER is set, the system blocks the process during close(2) until it can transmit the data or until the end of the interval indicated by the l_linger member, whichever comes first. If SO_LINGER is not specified, and close(2) is issued, the system handles the call in a way that allows the process to continue as quickly as possible. This option stores a linger structure.
SO_OOBINLINE	Reports whether the socket leaves received out-of-band data (data marked urgent) in line. This option stores an int value. This is a boolean option.
SO_SNDBUF	Reports send buffer size information. This option stores an int value.
SO_RCVBUF	Reports receive buffer size information. This option stores an int value.
SO_ERROR	Reports information about error status and clears it. This option stores an int value.
SO_TYPE	Reports the socket type. This option stores an int value.
SO_DONTROUTE	Reports whether outgoing messages bypass the standard routing facilities. The destination must be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. The effect, if any, of this option depends on what protocol is in use. This option stores an int value. This is a boolean option.
SO_MAC_EXEMPT	Gets the mandatory access control status of the socket. A socket that has this option enabled can communicate with an unlabeled peer if the socket is in the global zone or has a label that dominates the default label of the peer. Otherwise, the socket must have a label that is equal to the default label of the unlabeled peer. SO_MAC_EXEMPT is a boolean option that is available only when the system is configured with Trusted Extensions.
SO_ALLZONES	Bypasses zone boundaries (privileged). This option stores an int value. This is a boolean option.
	The SO_ALLZONES option can be used to bypass zone boundaries between shared-IP zones. Normally, the system prevents a socket from being bound to an address that is not assigned to the current zone. It also prevents a socket that is bound to a wildcard address from receiving traffic for other zones. However, some daemons which run in the global zone might need to send and receive traffic using addresses that belong to other shared-IP zones. If set before a socket is bound, SO_ALLZONES causes the socket to ignore zone boundaries between shared-IP zones and permits the socket to be bound to any address assigned to the shared-IP zones. If the socket is bound to a wildcard address, it receives traffic

intended for all shared-IP zones and behaves as if an equivalent socket were bound in each active shared-IP zone. Applications that use the SO_ALLZONES option to initiate connections or send datagram traffic should specify the source address for outbound traffic by binding to a specific address. There is no effect from setting this option in an exclusive-IP zone. Setting this option requires the sys_net_config privilege. See zones(5).

SO_DOMAIN   get the domain used in the socket (get only)

SO_PROTOTYPE   for socket in domains AF_INET and AF_INET6, get the underlying protocol number used in the socket. For socket in domain AF_ROUTE, get the address family used in the socket.

For boolean options, a zero value indicates that the option is disabled and a non-zero value indicates that the option is enabled.

Options at other protocol levels vary in format and name.

The socket in use may require the process to have appropriate privileges to use the getsockopt() function.

Return Values   Upon successful completion, getsockopt() returns 0. Otherwise, −1 is returned and errno is set to indicate the error.

Errors   The getsockopt() function will fail if:

EBADF   The *socket* argument is not a valid file descriptor.

EFAULT   The *option_value* or *option_len* parameter can not be accessed or written.

EINVAL   The specified option is invalid at the specified socket level.

ENOPROTOOPT   The option is not supported by the protocol.

ENOTSOCK   The *socket* argument does not refer to a socket.

The getsockopt() function may fail if:

EACCES   The calling process does not have the appropriate privileges.

EINVAL   The socket has been shut down.

ENOBUFS   Insufficient resources are available in the system to complete the call.

ENOSR   There were insufficient STREAMS resources available for the operation to complete.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also   close(2), bind(3XNET), endprotoent(3XNET), setsockopt(3XNET), socket(3XNET), attributes, standards(5)

Name getsourcefilter, setsourcefilter, getipv4sourcefilter, setipv4sourcefilter – retrieve and set a socket's multicast filter

Synopsis cc [*flag*...] *file*... -lsocket [*library*...]
#include <netinet/in.h>

int getsourcefilter(int *s*, uint32_t *interface*,
    struct sockaddr **group*, socklen_t *grouplen*, uint32_t **fmode*,
    uint_t **numsrc*, struct sockaddr_storage **slist*);

int setsourcefilter(int *s*, uint32_t *interface*,
    struct sockaddr **group*, socklen_t *grouplen*, uint32_t *fmode*,
    uint_t *numsrc*, struct sockaddr_storage **slist*);

int getipv4sourcefilter(int *s*, struct in_addr *interface*,
    struct in_addr *group*, uint32_t **fmode*, uint32_t **numsrc*,
    struct in_addr **slist*);

int setipv4sourcefilter(int *s*, struct in_addr *interface*,
    struct in_addr *group*,uint32_t *fmode*, uint32_t *numsrc*,
    struct in_addr **slist*);

Description These functions allow applications to retrieve and modify the multicast filtering state for a tuple consisting of socket, interface, and multicast group values.

A multicast filter is described by a filter mode, which is MODE_INCLUDE or MODE_EXCLUDE, and a list of source addresses which are filtered. If a group is simply joined with no source address restrictions, the filter mode is MODE_EXCLUDE and the source list is empty.

The getsourcefilter() and setsourcefilter() functions are protocol-independent. They can be used on either PF_INET or PF_INET6 sockets. The getipv4sourcefilter() and setipv4sourcefilter() functions are IPv4-specific. They must be used only on PF_INET sockets.

For the protocol-independent functions, the first four arguments identify the socket, interface, multicast group tuple values. The argument *s* is an open socket of type SOCK_DGRAM or SOCK_RAW. The *interface* argument is the interface index. The interface name can be mapped to the index using if_nametoindex(3SOCKET). The *group* points to either a sockaddr_in containing an IPv4 multicast address if the socket is PF_INET or a sockaddr_in6 containing an IPv6 multicast address if the socket is PF_INET6. The *grouplen* is the size of the structure pointed to by *group*.

For the IPv4-specific functions, the first three arguments identify the same socket, interface, multicast group tuple values. The argument *s* is an open socket of type SOCK_DGRAM or SOCK_RAW and protocol family PF_INET. The *interface* argument is the IPv4 address assigned to the local interface. The *group* argument is the IPv4 multicast address.

The getsourcefilter() and getipv4sourcefilter() functions retrieve the current filter for the given tuple consisting of socket, interface, and multicast group values. On successful return, *fmode* contains either MODE_INCLUDE or MODE_EXCLUDE, indicating the filter mode. On

input, the *numsrc* argument holds the number of addresses that can fit in the *slist* array. On return, *slist* contains as many addresses as fit, while *numsrc* contains the total number of source addresses in the filter. It is possible that *numsrc* can contain a number larger than the number of addresses in the *slist* array. An application might determine the required buffer size by calling getsourcefilter() with *numsrc* containing 0 and *slist* a NULL pointer. On return, *numsrc* contains the number of elements that the *slist* buffer must be able to hold. Alternatively, the maximum number of source addresses allowed by this implementation is defined in <netinet/in.h>:

```
#define MAX_SRC_FILTER_SIZE    64
```

The setsourcefilter() and setipv4sourcefilter functions replace the current filter with the filter specified in the arguments *fmode*, *numsrc*, and *slist*. The *fmode* argument must be set to either MODE_INCLUDE or MODE_EXCLUDE. The *numsrc* argument is the number of addresses in the *slist* array. The *slist* argument points to the array of source addresses to be included or excluded, depending on the *fmode* value.

Return Values If successful, all four functions return 0. Otherwise, they return −1 and set errno to indicate the error.

Errors These functions will fail if:

EBADF	The *s* argument is not a valid descriptor.
EAFNOSUPPORT	The address family of the passed-in *sockaddr* is not AF_INET or AF_INET6.
ENOPROTOOPT	The socket *s* is not of type SOCK_DGRAM or SOCK_RAW.
ENOPROTOOPT	The address family of the group parameter does not match the protocol family of the socket.
ENOSR	Insufficient STREAMS resources available for the operation to complete.
ENXIO	The *interface* argument, either an index or an IPv4 address, does not identify a valid interface.

The getsourcefilter() and getipv4sourcefilter() functions will fail if:

EADDRNOTAVAIL	The tuple consisting of socket, interface, and multicast group values does not exist; *group* is not being listened to on *interface* by *socket*.

The functions setsourcefilter() and setipv4sourcefilter() can fail in the following additional case:

ENOBUFS	The source filter list is larger than that allowed by the implementation.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	**ATTRIBUTE VALUE**
Interface Stability	Committed
MT-Level	Safe

See Also if_nametoindex(3SOCKET), socket(3SOCKET), attributes(5)

RFC 3678

Name gss_accept_sec_context – accept a security context initiated by a peer application

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

OM_uint32 gss_accept_sec_context(OM_uint32 *minor_status*,
    gss_ctx_id_t *context_handle*,
    const gss_cred_id_t *acceptor_cred_handle*,
    const gss_buffer_t *input_token*,
    const gss_channel_bindings_t *input_chan_bindings*,
    const gss_name_t * *src_name*, gss_OID * *mech_type*,
    gss_buffer_t *output_token*, OM_uint32 *ret_flags*,
    OM_uint32 * *time_rec*, gss_cred_id_t *delegated_cred_handle*);

Parameters The parameter descriptions for gss_accept_sec_context() follow:

minor_status
> The status code returned by the underlying mechanism.

context_handle
> The context handle to return to the initiator. This should be set to GSS_C_NO_CONTEXT before the loop begins.

acceptor_cred_handle
> The handle for the credentials acquired by the acceptor, typically through gss_acquire_cred(). It may be initialized to GSS_C_NO_CREDENTIAL to indicate a default credential to use. If no default credential is defined, the function returns GSS_C_NO_CRED.

input_token_buffer
> Token received from the context initiative.

input_chan_bindings
> Optional application-specified bindings. Allows application to securely bind channel identification information to the security context. Set to GSS_C_NO_CHANNEL_BINDINGS if you do not want to use channel bindings.

src_name
> The authenticated name of the context initiator. After use, this name should be deallocated by passing it to gss_release_name(). See gss_release_name(3GSS). If not required, specify NULL.

mech_type
> The security mechanism used. Set to NULL if it does not matter which mechanism is used.

output_token
> The token to send to the acceptor. Initialize it to GSS_C_NO_BUFFER before the function is called (or its length field set to zero). If the length is zero, no token need be sent.

ret_flags
> Contains various independent flags, each of which indicates that the context supports a specific service option. If not needed, specify NULL. Test the returned bit-mask *ret_flags*

value against its symbolic name to determine if the given option is supported by the context. *ret_flags* may contain one of the following values:

GSS_C_DELEG_FLAG
  If true, delegated credentials are available by means of the *delegated_cred_handle* parameter. If false, no credentials were delegated.

GSS_C_MUTUAL_FLAG
  If true, a remote peer asked for mutual authentication. If false, no remote peer asked for mutual authentication.

GSS_C_REPLAY_FLAG
  If true, replay of protected messages will be detected. If false, replayed messages will not be detected.

GSS_C_SEQUENCE_FLAG
  If true, out of sequence protected messages will be detected. If false, they will not be detected.

GSS_C_CONF_FLAG
  If true, confidentiality service may be invoked by calling the gss_wrap() routine. If false, no confidentiality service is available by means of gss_wrap(). gss_wrap() will provide message encapsulation, data-origin authentication and integrity services only.

GSS_C_INTEG_FLAG
  If true, integrity service may be invoked by calling either the gss_get_mic(3GSS) or the gss_wrap(3GSS) routine. If false, per-message integrity service is not available.

GSS_C_ANON_FLAG
  If true, the initiator does not wish to be authenticated. The *src_name* parameter, if requested, contains an anonymous internal name. If false, the initiator has been authenticated normally.

GSS_C_PROT_READY_FLAG
  If true, the protection services specified by the states of GSS_C_CONF_FLAG and GSS_C_INTEG_FLAG are available if the accompanying major status return value is either GSS_S_COMPLETE or GSS_S_CONTINUE_NEEDED. If false, the protection services are available only if the accompanying major status return value is GSS_S_COMPLETE.

GSS_C_TRANS_FLAG
  If true, the resultant security context may be transferred to other processes by means of a call to gss_export_sec_context(3GSS). If false, the security context cannot be transferred.

time_rec
The number of sections for which the context will remain value Specify NULL if not required.

delegated_cred_handle
> The credential value for credentials received from the context's initiator. It is valid only if the initiator has requested that the acceptor act as a proxy: that is, if the *ret_flag* argument resolves to GSS_C_DELEG_FLAG.

Description   The gss_accept_sec_context() function allows a remotely initiated security context between the application and a remote peer to be established. The routine may return an *output_token*, which should be transferred to the peer application, where the peer application will present it to gss_init_sec_context(). See gss_init_sec_context(3GSS). If no token need be sent, gss_accept_sec_context() will indicate this by setting the length field of the *output_token* argument to zero. To complete the context establishment, one or more reply tokens may be required from the peer application; if so, gss_accept_sec_context() will return a status flag of GSS_S_CONTINUE_NEEDED, in which case it should be called again when the reply token is received from the peer application, passing the token to gss_accept_sec_context() by means of the *input_token* parameters.

Portable applications should be constructed to use the token length and return status to determine whether to send or to wait for a token.

Whenever gss_accept_sec_context() returns a major status that includes the value GSS_S_CONTINUE_NEEDED, the context is not fully established, and the following restrictions apply to the output parameters:

- The value returned by means of the *time_rec* parameter is undefined.
- Unless the accompanying *ret_flags* parameter contains the bit GSS_C_PROT_READY_FLAG, which indicates that per-message services may be applied in advance of a successful completion status, the value returned by the *mech_type* parameter may be undefined until gss_accept_sec_context() returns a major status value of GSS_S_COMPLETE.

The values of the GSS_C_DELEG_FLAG, GSS_C_MUTUAL_FLAG, GSS_C_REPLAY_FLAG, GSS_C_SEQUENCE_FLAG, GSS_C_CONF_FLAG, GSS_C_INTEG_FLAG and GSS_C_ANON_FLAG bits returned by means of the *ret_flags* parameter are values that would be valid if context establishment were to succeed.

The values of the GSS_C_PROT_READY_FLAG and GSS_C_TRANS_FLAG bits within *ret_flags* indicate the actual state at the time gss_accept_sec_context() returns, whether or not the context is fully established. However, applications should not rely on this behavior, as GSS_C_PROT_READY_FLAG was not defined in Version 1 of the GSS-API. Instead, applications should be prepared to use per-message services after a successful context establishment, based upon the GSS_C_INTEG_FLAG and GSS_C_CONF_FLAG values.

All other bits within the *ret_flags* argument are set to zero.

While gss_accept_sec_context() returns GSS_S_CONTINUE_NEEDED, the values returned by means of the the *ret_flags* argument indicate the services available from the established context. If the initial call of gss_accept_sec_context() fails, no context object is created, and

the value of the *context_handle* parameter is set to GSS_C_NO_CONTEXT. In the event of a failure on a subsequent call, the security context and the *context_handle* parameter are left untouched for the application to delete using gss_delete_sec_context(3GSS). During context establishment, the informational status bits GSS_S_OLD_TOKEN and GSS_S_DUPLICATE_TOKEN indicate fatal errors; GSS-API mechanisms always return them in association with a routine error of GSS_S_FAILURE. This pairing requirement did not exist in version 1 of the GSS-API specification, so applications that wish to run over version 1 implementations must special-case these codes.

Errors gss_accept_sec_context() may return the following status codes:

GSS_S_COMPLETE	Successful completion.
GSS_S_CONTINUE_NEEDED	A token from the peer application is required to complete the context, and that gss_accept_sec_context() must be called again with that token.
GSS_S_DEFECTIVE_TOKEN	Consistency checks performed on the *input_token* failed.
GSS_S_DEFECTIVE_CREDENTIAL	Consistency checks performed on the credential failed.
GSS_S_NO_CRED	The supplied credentials were not valid for context acceptance, or the credential handle did not reference any credentials.
GSS_S_CREDENTIALS_EXPIRED	The referenced credentials have expired.
GSS_S_BAD_BINDINGS	The *input_token* contains different channel bindings than those specified by means of the *input_chan_bindings* parameter.
GSS_S_NO_CONTEXT	The supplied context handle did not refer to a valid context.
GSS_S_BAD_SIG	The *input_token* contains an invalid MIC.
GSS_S_OLD_TOKEN	The *input_token* was too old. This is a fatal error while establishing context.
GSS_S_DUPLICATE_TOKEN	The *input_token* is valid, but it is duplicate of a token already processed. This is a fatal error while establishing context.
GSS_S_BAD_MECH	The token received specified a mechanism that is not supported by the implementation or the provided credential.
GSS_S_FAILURE	The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Examples   EXAMPLE 1   Invoking gss_accept_sec_context() Within a Loop

A typical portable caller should always invoke gss_accept_sec_context() within a loop:

```
gss_ctx_id_t context_hdl = GSS_C_NO_CONTEXT;

do {
   receive_token_from_peer(input_token);
   maj_stat = gss_accept_sec_context(&min_stat,
                                     &context_hdl,
                                     cred_hdl,
                                     input_token,
                                     input_bindings,
                                     &client_name,
                                     &mech_type,
                                     output_token,
                                     &ret_flags,
                                     &time_rec,
                                     &deleg_cred);
   if (GSS_ERROR(maj_stat)) {
      report_error(maj_stat, min_stat);
   };
   if (output_token->length != 0) {
      send_token_to_peer(output_token);
      gss_release_buffer(&min_stat, output_token);
   };
   if (GSS_ERROR(maj_stat)) {
      if (context_hdl != GSS_C_NO_CONTEXT)
         gss_delete_sec_context(&min_stat,
                                &context_hdl,
                                GSS_C_NO_BUFFER);
      break;
   };
} while (maj_stat & GSS_S_CONTINUE_NEEDED);

/* Check client_name authorization */
...

(void) gss_release_name(&min_stat, &client_name);

/* Use and/or store delegated credential */
...

(void) gss_release_cred(&min_stat, &deleg_cred);
```

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   gss_delete_sec_context(3GSS), gss_export_sec_context(3GSS), gss_get_mic(3GSS),
gss_init_sec_context(3GSS), gss_release_cred(3GSS), gss_release_name(3GSS),
gss_store_cred(3GSS), gss_wrap(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_acquire_cred – acquire a handle for a pre-existing credential by name

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

```
OM_uint32 gss_acquire_cred(OM_uint32 *minor_status,
    const gss_name_t desired_name, OM_uint32 time_req,
    const gss_OID_set desired_mech, gss_cred_usage_t cred_usage,
    gss_cred_id_t *output_cred_handle, gss_OID_set *actual_mechs,
    OM_uint32 *time_rec);
```

Description The gss_acquire_cred() function allows an application to acquire a handle for a pre-existing credential by name. This routine is not intended as a function to login to the network; a function for login to the network would involve creating new credentials rather than merely acquiring a handle to existing credentials.

If *desired_name* is GSS_C_NO_NAME, the call is interpreted as a request for a credential handle that will invoke default behavior when passed to gss_init_sec_context(3GSS) (if *cred_usage* is GSS_C_INITIATE or GSS_C_BOTH) or gss_accept_sec_context(3GSS) (if *cred_usage* is GSS_C_ACCEPT or GSS_C_BOTH).

Normally gss_acquire_cred() returns a credential that is valid only for the mechanisms requested by the *desired_mechs* argument. However, if multiple mechanisms can share a single credential element, the function returns all the mechanisms for which the credential is valid in the *actual_mechs* argument.

gss_acquire_cred() is intended to be used primarily by context acceptors, since the GSS-API routines obtain initiator credentials through the system login process. Accordingly, you may not acquire GSS_C_INITIATE or GSS_C_BOTH credentials by means of gss_acquire_cred() for any name other than GSS_C_NO_NAME. Alternatively, you may acquire GSS_C_INITIATE or GSS_C_BOTH credentials for a name produced when gss_inquire_cred(3GSS) is applied to a valid credential, or when gss_inquire_context(3GSS) is applied to an active context.

If credential acquisition is time-consuming for a mechanism, the mechanism may choose to delay the actual acquisition until the credential is required, for example, by gss_init_sec_context(3GSS) or by gss_accept_sec_context(3GSS). Such mechanism-specific implementations are, however, invisible to the calling application; thus a call of gss_inquire_cred(3GSS) immediately following the call of gss_acquire_cred() will return valid credential data and incur the overhead of a deferred credential acquisition.

Parameters The parameter descriptions for gss_acquire_cred() follow:

desired_name	The name of the principal for which a credential should be acquired.
time_req	The number of seconds that credentials remain valid. Specify GSS_C_INDEFINITE to request that the credentials have the maximum permitted lifetime

desired_mechs	The set of underlying security mechanisms that may be used. GSS_C_NO_OID_SET may be used to obtain a default.
cred_usage	A flag that indicates how this credential should be used. If the flag is GSS_C_ACCEPT, then credentials will be used only to accept security credentials. GSS_C_INITIATE indicates that credentials will be used only to initiate security credentials. If the flag is GSS_C_BOTH, then credentials may be used either to initiate or accept security contexts.
output_cred_handle	The returned credential handle. Resources associated with this credential handle must be released by the application after use with a call to gss_release_cred(3GSS)
actual_mechs	The set of mechanisms for which the credential is valid. Storage associated with the returned OID-set must be released by the application after use with a call to gss_release_oid_set(3GSS). Specify NULL if not required.
time_rec	Actual number of seconds for which the returned credentials will remain valid. Specify NULL if not required.
minor_status	Mechanism specific status code.

Errors gss_acquire_cred() may return the following status code:

GSS_S_COMPLETE	Successful completion.
GSS_S_BAD_MECH	An unavailable mechanism has been requested.
GSS_S_BAD_NAMETYPE	The type contained within the *desired_name* parameter is not supported.
GSS_S_BAD_NAME	The value supplied for *desired_name* parameter is ill formed.
GSS_S_CREDENTIALS_EXPIRED	The credentials could not be acquired because they have expired.
GSS_S_NO_CRED	No credentials were found for the specified name.
GSS_S_FAILURE	The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also gss_accept_sec_context(3GSS), gss_init_sec_context(3GSS),
gss_inquire_context(3GSS), gss_inquire_cred(3GSS), gss_release_cred(3GSS),
gss_release_oid_set(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_add_cred – add a credential-element to a credential

Synopsis cc [*flag...*] *file...* -lgss [*library...*]
#include <gssapi/gssapi.h>

OM_uint32 gss_add_cred(OM_uint32 *minor_status*,
    const gss_cred_id_t *input_cred_handle*,
    const gss_name_t *desired_name*,
    const gss_OID *desired_mech*,
    gss_cred_usage_t *cred_usage*,
    OM_uint32 *initiator_time_req*,
    OM_uint32 *acceptor_time_req*,
    gss_cred_id_t *output_cred_handle*,
    gss_OID_set *actual_mechs*,
    OM_uint32 *initiator_time_rec*,
    OM_uint32 *acceptor_time_rec*);

Parameters The parameter descriptions for gss_add_cred() follow:

minor_status	Mechanism specific status code.
input_cred_handle	Credential to which the credential-element is added. If GSS_C_NO_CREDENTIAL is specified, the function composes the new credential based on default behavior. While the credential-handle is not modified by gss_add_cred(), the underlying credential is modified if *output_credential_handle* is NULL.
desired_name	Name of the principal for which a credential should be acquired.
desired_mech	Underlying security mechanism with which the credential can be used. GSS_C_NULL_OID can be used to obtain a default.
cred_usage	Flag that indicates how a credential is used to initiate or accept security credentials. If the flag is GSS_C_ACCEPT, the credentials are used only to accept security credentials. If the flag is GSS_C_INITIATE, the credentials are used only to initiate security credentials. If the flag is GSS_C_BOTH, the credentials can be used to either initiate or accept security contexts.
initiator_time_req	Number of seconds that the credential may remain valid for initiating security contexts. This argument is ignored if the composed credentials are of the GSS_C_ACCEPT type. Specify GSS_C_INDEFINITE to request that the credentials have the maximum permitted initiator lifetime.
acceptor_time_req	Number of seconds that the credential may remain valid for accepting security contexts. This argument is ignored if the composed credentials are of the GSS_C_INITIATE type. Specify GSS_C_INDEFINITE to request that the credentials have the maximum permitted initiator lifetime.

output_cred_handle   Returned credential handle that contains the new credential-element and all the credential-elements from *input_cred_handle*. If a valid pointer to a gss_cred_id_t is supplied for this parameter, gss_add_cred() creates a new credential handle that contains all credential-elements from *input_cred_handle* and the newly acquired credential-element. If NULL is specified for this parameter, the newly acquired credential-element is added to the credential identified by *input_cred_handle*.

The resources associated with any credential handle returned by means of this parameter must be released by the application after use by a call to gss_release_cred(3GSS).

actual_mechs   Complete set of mechanisms for which the new credential is valid. Storage for the returned OID-set must be freed by the application after use by a call to gss_release_oid_set(3GSS). Specify NULL if this parameter is not required.

initiator_time_rec   Actual number of seconds for which the returned credentials remain valid for initiating contexts using the specified mechanism. If a mechanism does not support expiration of credentials, the value GSS_C_INDEFINITE is returned. Specify NULL if this parameter is not required.

acceptor_time_rec   Actual number of seconds for which the returned credentials remain valid for accepting security contexts using the specified mechanism. If a mechanism does not support expiration of credentials, the value GSS_C_INDEFINITE is returned. Specify NULL if this parameter is not required.

Description   The gss_add_cred() function adds a credential-element to a credential. The credential-element is identified by the name of the principal to which it refers. This function is not intended as a function to login to the network. A function for login to the network would involve creating new mechanism-specific authentication data, rather than acquiring a handle to existing data.

If the value of *desired_name* is GSS_C_NO_NAME, the call is interpreted as a request to add a credential-element to invoke default behavior when passed to gss_init_sec_context(3GSS) if the value of *cred_usage* is GSS_C_INITIATE or GSS_C_BOTH. The call is also interpreted as a request to add a credential-element to the invoke default behavior when passed to gss_accept_sec_context(3GSS) if the value of *cred_usage* is GSS_C_ACCEPT or GSS_C_BOTH.

The gss_add_cred() function is expected to be used primarily by context acceptors. The GSS-API provides mechanism-specific ways to obtain GSS-API initiator credentials through

the system login process. Consequently, the GSS-API does not support acquiring GSS_C_INITIATE or GSS_C_BOTH credentials by means of gss_acquire_cred(3GSS) for any name other than the following:

- GSS_C_NO_NAME
- Name produced by gss_inquire_cred(3GSS) applied to a valid credential
- Name produced by gss_inquire_context(3GSS) applied to an active context

If credential acquisition is time consuming for a mechanism, the mechanism can choose to delay the actual acquisition until the credential is required by gss_init_sec_context(3GSS), for example, or by gss_accept_sec_context(3GSS). Such mechanism-specific implementation decisions are invisible to the calling application. A call to gss_inquire_cred(3GSS) immediately following the call gss_add_cred() returns valid credential data as well as incurring the overhead of deferred credential acquisition.

The gss_add_cred() function can be used either to compose a new credential that contains all credential-elements of the original in addition to the newly-acquired credential-element. The function can also be used to add the new credential-element to an existing credential. If the value of the *output_cred_handle* parameter is NULL, the new credential-element is added to the credential identified by *input_cred_handle*. If a valid pointer is specified for the *output_cred_handle* parameter, a new credential handle is created.

If the value of *input_cred_handle* is GSS_C_NO_CREDENTIAL, the gss_add_cred() function composes a credential and sets the *output_cred_handle* parameter based on the default behavior. The call has the same effect as a call first made by the application to gss_acquire_cred(3GSS) to specify the same usage and to pass GSS_C_NO_NAME as the *desired_name* parameter. Such an application call obtains an explicit credential handle that incorporates the default behaviors, then passes the credential handle to gss_add_cred(), and finally calls gss_release_cred(3GSS) on the first credential handle.

If the value of the *input_cred_handle* parameter is GSS_C_NO_CREDENTIAL, a non-NULL value must be supplied for the *output_cred_handle* parameter.

Return Values   The gss_add_cred() function can return the following status codes:

GSS_S_COMPLETE	Successful completion.
GSS_S_BAD_MECH	An unavailable mechanism has been requested.
GSS_S_BAD_NAMETYPE	The type contained within the *desired_name* parameter is not supported.
GSS_S_BAD_NAME	The value supplied for *desired_name* parameter is ill formed.
GSS_S_DUPLICATE_ELEMENT	The credential already contains an element for the requested mechanism that has overlapping usage and validity period.

GSS_S_CREDENTIALS_EXPIRED   The credentials could not be added because they have
                                 expired.

GSS_S_NO_CRED   No credentials were found for the specified name.

GSS_S_FAILURE   The underlying mechanism detected an error for which no
                                 specific GSS status code is defined. The
                                 mechanism-specific status code reported by means of the
                                 minor_status parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also gss_accept_sec_context(3GSS), gss_acquire_cred(3GSS),
gss_init_sec_context(3GSS), gss_inquire_context(3GSS), gss_inquire_cred(3GSS),
gss_release_cred(3GSS), gss_release_oid_set(3GSS), libgss(3LIB), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_add_oid_set_member – add an object identifier to an object identifier set

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

OM_uint32 gss_add_oid_set_member(OM_uint32 *minor_status*,
    const gss_OID *member_oid*, gss_OID_set *oid_set*);

Parameters The parameter descriptions for gss_add_oid_set_member() follow:

minor_status   A mechanism specific status code.

member_oid   Object identifier to be copied into the set.

oid_set   Set in which the object identifier should be inserted.

Description The gss_add_oid_set_member() function adds an object identifier to an object identifier set.
You should use this function in conjunction with gss_create_empty_oid_set(3GSS) when
constructing a set of mechanism OIDs for input to gss_acquire_cred(3GSS). The *oid_set*
parameter must refer to an OID-set created by GSS-API, that is, a set returned by
gss_create_empty_oid_set(3GSS).

The GSS-API creates a copy of the *member_oid* and inserts this copy into the set, expanding
the storage allocated to the OID-set elements array, if necessary. New members are always
added to the end of the OID set's elements. If the *member_oid* is already present, the *oid_set*
should remain unchanged.

Errors The gss_add_oid_set_member() function can return the following status codes:

GSS_S_COMPLETE
  Successful completion.

GSS_S_FAILURE
  The underlying mechanism detected an error for which no specific GSS status code is
  defined. The mechanism-specific status code reported by means of the *minor_status*
  parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also gss_acquire_cred(3GSS), gss_create_empty_oid_set(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_canonicalize_name – convert an internal name to a mechanism name

Synopsis cc [*flag ...*] *file*... -lgss [*library ...*]
#include <gssapi/gssapi.h>

OM_uint32 gss_canonicalize_name(OM_uint32 **minor_status*,
    const gss_name_t *input_name*,const gss_OID *mech_type*,
    gss_name_t *output_name*);

Description The gss_canonicalize_name() function generates a canonical mechanism name from an arbitrary internal name. The mechanism name is the name that would be returned to a context acceptor on successful authentication of a context where the initiator used the *input_name* in a successful call to gss_acquire_cred(3GSS), specifying an OID set containing *mech_type* as its only member, followed by a call to gss_init_sec_context(3GSS), specifying *mech_type* as the authentication mechanism.

Parameters The parameter descriptions for gss_canonicalize_name() follow:

minor_status Mechanism-specific status code.

input_name The name for which a canonical form is desired.

mech_type The authentication mechanism for which the canonical form of the name is desired. The desired mechanism must be specified explicitly; no default is provided.

output_name The resultant canonical name. Storage associated with this name must be freed by the application after use with a call to gss_release_name(3GSS).

Errors The gss_canonicalize_name() function may return the status codes:

GSS_S_COMPLETE Successful completion.

GSS_S_BAD_MECH The identified mechanism is not supported.

GSS_S_BAD_NAMETYPE The provided internal name contains no elements that could be processed by the specified mechanism.

GSS_S_BAD_NAME The provided internal name was ill-formed.

GSS_S_FAILURE The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   gss_acquire_cred(3GSS), gss_init_sec_context(3GSS), gss_release_name(3GSS),
attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_compare_name – compare two internal-form names

Synopsis cc [*flag* ...] *file*... -lgss [*library* ...]
#include <gssapi/gssapi.h>

OM_uint32 gss_compare_name(OM_uint32 **minor_status*,
    const gss_name_t *name1*,const gss_name_t *name2*,
    int **name_equal*);

Description The gss_compare_name() function allows an application to compare two internal-form names to determine whether they refer to the same entity.

If either name presented to gss_compare_name() denotes an anonymous principal, the routines indicate that the two names do not refer to the same identity.

Parameters The parameter descriptions for gss_compare_name() follow:

minor_status   Mechanism-specific status code.

name1   Internal-form name.

name2   Internal-form name.

name_equal   If non-zero, the names refer to same entity. If 0, the names refer to different entities. Strictly, the names are not known to refer to the same identity.

Errors The gss_compare_name() function may return the following status codes:

GSS_S_COMPLETE   Successful completion.

GSS_S_BAD_NAMETYPE   The two names were of incomparable types.

GSS_S_BAD_NAME   One or both of *name1* or *name2* was ill-formed.

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_context_time – determine how long a context will remain valid

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

OM_uint32 gss_context_time(OM_uint32 *_minor_status_,
    gss_ctx_id_t *_context_handle_,OM_uint32 *_time_rec_);

Description The gss_context_time() function determines the number of seconds for which the specified context will remain valid.

Parameters The parameter descriptions for gss_context_time() are as follows:

minor_status   A mechanism-specific status code.

context_handle A read-only value. Identifies the context to be interrogated.

time_rec   Modifies the number of seconds that the context remains valid. If the context has already expired, returns zero.

Errors The gss_context_time() function returns one of the following status codes:

GSS_S_COMPLETE   Successful completion.

GSS_S_CONTEXT_EXPIRED   The context has already expired.

GSS_S_NO_CONTEXT   The *context_handle* parameter did not identify a valid context.

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also gss_init_sec_context(3GSS), gss_accept_sec_context(3GSS), gss_delete_sec_context(3GSS), gss_process_context_token(3GSS), gss_inquire_context(3GSS), gss_wrap_size_limit(3GSS), gss_export_sec_context(3GSS), gss_import_sec_context(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_create_empty_oid_set – create an object-identifier set containing no object identifiers

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

OM_uint32 gss_create_empty_oid_set(OM_uint32 *minor_status,
    gss_OID_set *oid_set);

Description The gss_create_empty_oid_set() function creates an object-identifier set containing no object identifiers to which members may be subsequently added using the gss_add_oid_set_member(3GSS) function. These functions can be used to construct sets of mechanism object identifiers for input to gss_acquire_cred(3GSS).

Parameters The parameter descriptions for gss_create_empty_oid_set() follow:

minor_status   Mechanism-specific status code

oid_set   Empty object identifier set. The function will allocate the gss_OID_set_desc object, which the application must free after use with a call to gss_release_oid_set(3GSS).

Errors The gss_create_empty_oid_set() function may return the following status codes:

GSS_S_COMPLETE   Successful completion

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also gss_acquire_cred(3GSS), gss_add_oid_set_member(3GSS), gss_release_oid_set(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_delete_sec_context – delete a GSS-API security context

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

OM_uint32 gss_delete_sec_context(OM_uint32 *minor_status*,
    gss_ctx_id_t *context_handle*,gss_buffer_t *output_token*);

Description Use the gss_delete_sec_context() function to delete a security context. The
gss_delete_sec_context() function will delete the local data structures associated with the
specified security context. You may not obtain further security services that use the context
specified by *context_handle*.

In addition to deleting established security contexts, gss_delete_sec_context() will delete
any half-built security contexts that result from incomplete sequences of calls to
gss_init_sec_context(3GSS) and gss_accept_sec_context(3GSS).

The Solaris implementation of the GSS-API retains the *output_token* parameter for
compatibility with version 1 of the GSS-API. Both peer applications should invoke
gss_delete_sec_context(), passing the value GSS_C_NO_BUFFER to the *output_token*
parameter; this indicates that no token is required. If the application passes a valid buffer to
gss_delete_sec_context(), it will return a zero-length token, indicating that no token
should be transferred by the application.

Parameters The parameter descriptions for gss_delete_sec_context() follow:

minor_status	A mechanism specific status code.
context_handle	Context handle identifying specific context to delete. After deleting the context, the GSS-API will set *context_handle* to GSS_C_NO_CONTEXT.
output_token	A token to be sent to remote applications that instructs them to delete the context.

Errors gss_delete_sec_context() may return the following status codes:

GSS_S_COMPLETE	Successful completion.
GSS_S_NO_CONTEXT	No valid context was supplied.
GSS_S_FAILURE	The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   gss_accept_sec_context(3GSS), gss_init_sec_context(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_display_name – convert internal-form name to text

Synopsis cc [*flag* ...] *file*... -lgss [*library* ...]
#include <gssapi/gssapi.h>

OM_uint32 gss_display_name(OM_uint32 *minor_status*,
    const gss_name_t *input_name*, gss_buffer_t *output_name_buffer*,
    gss_OID *output_name_type*);

Description The gss_display_name() function allows an application to obtain a textual representation of an opaque internal-form name for display purposes.

If *input_name* denotes an anonymous principal, the GSS-API returns the gss_OID value GSS_C_NT_ANONYMOUS as the *output_name_type*, and a textual name that is syntactically distinct from all valid supported printable names in *output_name_buffer*.

If *input_name* was created by a call to gss_import_name(3GSS), specifying GSS_C_NO_OID as the name-type, the GSS-API returns GSS_C_NO_OID by means of the *output_name_type* parameter.

Parameters The parameter descriptions for gss_display_name() follow:

minor_status	Mechanism-specific status code.
input_name	Name in internal form.
output_name_buffer	Buffer to receive textual name string. The application must free storage associated with this name after use with a call to gss_release_buffer(3GSS).
output_name_type	The type of the returned name. The returned gss_OID will be a pointer into static storage and should be treated as read-only by the caller. In particular, the application should not attempt to free it. Specify NULL if this parameter is not required.

Errors The gss_display_name() function may return the following status codes:

GSS_S_COMPLETE	Successful completion.
GSS_S_BAD_NAME	The *input_name* was ill-formed.
GSS_S_FAILURE	The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   gss_import_name(3GSS), gss_release_buffer(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_display_status – convert a GSS-API status code to text

Synopsis cc [*flag...*] *file...* -lgss [*library...*]
#include <gssapi/gssapi.h>

OM_uint32 gss_display_status(OM_uint32 *minor_status,
    OM_uint32 *status value*, int *status type*,
    const gss_OID *mech_type*, OM_uint32 *message_context*,
    gss_buffer_t *status string*);

Description The gss_display_status() function enables an application to obtain a textual
representation of a GSS-API status code for display to the user or for logging purposes.
Because some status values may indicate multiple conditions, applications may need to call
gss_display_status() multiple times, with each call generating a single text string.

The *message_context* parameter is used by gss_acquire_cred() to store state information on
error messages that are extracted from a given *status_value*. The *message_context* parameter
must be initialized to 0 by the application prior to the first call, and gss_display_status()
will return a non-zero value in this parameter if there are further messages to extract.

The *message_context* parameter contains all state information required by
gss_display_status() to extract further messages from the *status_value*. If a non-zero value
is returned in this parameter, the application is not required to call gss_display_status()
again unless subsequent messages are desired.

Parameters The parameter descriptions for gss_display_status() follow:

minor_status	Status code returned by the underlying mechanism.
status_value	Status value to be converted.
status_type	If the value is GSS_C_GSS_CODE, *status_value* is a GSS-API status code. If the value is GSS_C_MECH_CODE, then *status_value* is a mechanism status code.
mech_type	Underlying mechanism that is used to interpret a minor status value. Supply GSS_C_NO_OID to obtain the system default.
message_context	Should be initialized to zero prior to the first call. On return from gss_display_status(), a non-zero *status_value* parameter indicates that additional messages may be extracted from the status code by means of subsequent calls to gss_display_status(), passing the same *status_value*, *status_type*, *mech_type*, and *message_context* parameters.
status_string	Textual representation of the *status_value*. Storage associated with this parameter must be freed by the application after use with a call to gss_release_buffer(3GSS).

Errors The gss_display_status() function may return the following status codes:

GSS_S_COMPLETE   Successful completion.

GSS_S_BAD_MECH   Indicates that translation in accordance with an unsupported
                      mechanism type was requested.

GSS_S_BAD_STATUS   The status value was not recognized, or the status type was neither
                      GSS_C_GSS_CODE nor GSS_C_MECH_CODE.

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific
                      GSS status code is defined. The mechanism-specific status code
                      reported by means of the *minor_status* parameter details the error
                      condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also gss_acquire_cred(3GSS), gss_release_buffer(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_duplicate_name – create a copy of an internal name

Synopsis cc [*flag ...*] *file*... -lgss [*library ...*]
#include <gssapi/gssapi.h>

OM_uint32 gss_duplicate_name(OM_uint32 **minor_status*,
    const gss_name_t *src_name*,gss_name_t **dest_name*);

Description The gss_duplicate_name() function creates an exact duplicate of the existing internal name *src_name*. The new *dest_name* will be independent of the *src_name*. The *src_name* and *dest_name* must both be released, and the release of one does not affect the validity of the other.

Parameters The parameter descriptions for gss_duplicate_name() follow:

minor_status   A mechanism-specific status code.

src_name   Internal name to be duplicated.

dest_name   The resultant copy of *src_name*. Storage associated with this name must be freed by the application after use with a call to gss_release_name(3GSS).

Errors The gss_duplicate_name() function may return the following status codes:

GSS_S_COMPLETE   Successful completion.

GSS_S_BAD_NAME   The *src_name* parameter was ill-formed.

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also gss_release_name(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_export_name – convert a mechanism name to export form

Synopsis cc [*flag ...*] *file*... -lgss [*library ...*]
#include <gssapi/gssapi.h>

OM_uint32 gss_export_name(OM_uint32 *minor_status,
    const gss_name_t *input_name*,gss_buffer_t *exported_name*);

Description The gss_export_name() function allows a GSS-API internal name to be converted into a
mechanism-specific name. The function produces a canonical contiguous string
representation of a mechanism name, suitable for direct comparison, with memory(3C), or for
use in authorization functions, matching entries in an access-control list. The *input_name*
parameter must specify a valid mechanism name, that is, an internal name generated by
gss_accept_sec_context(3GSS) or by gss_canonicalize_name(3GSS).

Parameters The parameter descriptions for gss_export_name() follow:

minor_status   A mechanism-specific status code.

input_name   The mechanism name to be exported.

exported_name   The canonical contiguous string form of *input_name*. Storage associated
                    with this string must freed by the application after use with
                    gss_release_buffer(3GSS).

Errors The gss_export_name() function may return the following status codes:

GSS_S_COMPLETE   Successful completion.

GSS_S_NAME_NOT_MN   The provided internal name was not a mechanism name.

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific
                     GSS status code is defined. The mechanism-specific status code
                     reported by means of the *minor_status* parameter details the error
                     condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also gss_accept_sec_context(3GSS), gss_canonicalize_name(3GSS),
gss_release_buffer(3GSS)memory(3C), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_export_sec_context – transfer a security context to another process

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

OM_uint32 gss_export_sec_context(OM_uint32 *minor_status*,
    gss_ctx_id_t *context_handle*,gss_buffer_t *interprocess_token*);

Description The gss_export_sec_context() function generates an interprocess token for transfer to
another process within an end system. gss_export_sec_context() and
gss_import_sec_context() allow a security context to be transferred between processes on a
single machine.

The gss_export_sec_context() function supports the sharing of work between multiple
processes. This routine is typically used by the context-acceptor, in an application where a
single process receives incoming connection requests and accepts security contexts over them,
then passes the established context to one or more other processes for message exchange.
gss_export_sec_context() deactivates the security context for the calling process and
creates an interprocess token which, when passed to gss_import_sec_context() in another
process, reactivates the context in the second process. Only a single instantiation of a given
context can be active at any one time; a subsequent attempt by a context exporter to access the
exported security context will fail.

The interprocess token may contain security-sensitive information, for example
cryptographic keys. While mechanisms are encouraged to either avoid placing such sensitive
information within interprocess tokens or to encrypt the token before returning it to the
application, in a typical object-library GSS-API implementation, this might not be possible.
Thus, the application must take care to protect the interprocess token and ensure that any
process to which the token is transferred is trustworthy. If creation of the interprocess token is
successful, the GSS-API deallocates all process-wide resources associated with the security
context and sets the context_handle to GSS_C_NO_CONTEXT. In the event of an error that makes
it impossible to complete the export of the security context, the function does not return an
interprocess token and leaves the security context referenced by the *context_handle* parameter
untouched.

Sun's implementation of gss_export_sec_context() does not encrypt the interprocess
token. The interprocess token is serialized before it is transferred to another process.

Parameters The parameter descriptions for gss_export_sec_context() are as follows:

minor_status	A mechanism-specific status code.
context_handle	Context handle identifying the context to transfer.
interprocess_token	Token to be transferred to target process. Storage associated with this token must be freed by the application after use with a call to gss_release_buffer(3GSS).

Errors   gss_export_sec_context() returns one of the following status codes:

GSS_S_COMPLETE   Successful completion.

GSS_S_CONTEXT_EXPIRED   The context has expired.

GSS_S_NO_CONTEXT   The context was invalid.

GSS_S_UNAVAILABLE   The operation is not supported.

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also   gss_accept_sec_context(3GSS), gss_import_sec_context(3GSS), gss_init_sec_context(3GSS), gss_release_buffer(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_get_mic – calculate a cryptographic message

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

OM_uint32 gss_get_mic(OM_uint32 **minor_status*,
    const gss_ctx_id_t *context_handle*, gss_qop_t *qop_req*,
    const gss_buffer_t *message_buffer*, gss_buffer_t *msg_token*);

Description The gss_get_mic() function generates a cryptographic MIC for the supplied message, and places the MIC in a token for transfer to the peer application. The *qop_req* parameter allows a choice between several cryptographic algorithms, if supported by the chosen mechanism.

Since some application-level protocols may wish to use tokens emitted by gss_wrap(3GSS) to provide secure framing, the GSS-API allows MICs to be derived from zero-length messages.

Parameters The parameter descriptions for gss_get_mic() follow:

minor_status   The status code returned by the underlying mechanism.

context_handle Identifies the context on which the message will be sent.

qop_req   Specifies the requested quality of protection. Callers are encouraged, on portability grounds, to accept the default quality of protection offered by the chosen mechanism, which may be requested by specifying GSS_C_QOP_DEFAULT for this parameter. If an unsupported protection strength is requested, gss_get_mic() will return a *major_status* of GSS_S_BAD_QOP.

message_buffer The message to be protected.

msg_token   The buffer to receive the token. Storage associated with this message must be freed by the application after use with a call to gss_release_buffer(3GSS).

Errors gss_get_mic() may return the following status codes:

GSS_S_COMPLETE   Successful completion.

GSS_S_CONTEXT_EXPIRED   The context has already expired.

GSS_S_NO_CONTEXT   The *context_handle* parameter did not identify a valid context.

GSS_S_BAD_QOP   The specified QOP is not supported by the mechanism.

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   gss_release_buffer(3GSS), gss_wrap(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_import_name – convert a contiguous string name to GSS_API internal format

Synopsis cc [*flag ...*] *file...* -lgss [*library ...*]
#include <gssapi/gssapi.h>

OM_uint32 gss_import_name(OM_uint32 * *minor_status*,
    const gss_buffer_t *input_name_buffer*, const gss_OID *input_name_type*,
    gss_name_t *output_name*);

Description The gss_import_name() function converts a contiguous string name to internal form. In general, the internal name returned by means of the *output_name* parameter will not be a mechanism name; the exception to this is if the *input_name_type* indicates that the contiguous string provided by means of the *input_name_buffer* parameter is of type GSS_C_NT_EXPORT_NAME, in which case, the returned internal name will be a mechanism name for the mechanism that exported the name.

Parameters The parameter descriptions for gss_import_name() follow:

minor_status	Status code returned by the underlying mechanism.
input_name_buffer	The gss_buffer_desc structure containing the name to be imported.
input_name_type	A gss_OID that specifies the format that the *input_name_buffer* is in.
output_name	The gss_name_t structure to receive the returned name in internal form. Storage associated with this name must be freed by the application after use with a call to gss_release_name().

Errors The gss_import_name() function may return the following status codes:

GSS_S_COMPLETE	The gss_import_name() function completed successfully.
GSS_S_BAD_NAMETYPE	The *input_name_type* was unrecognized.
GSS_S_BAD_NAME	The *input_name* parameter could not be interpreted as a name of the specified type.
GSS_S_BAD_MECH	The *input_name_type* was GSS_C_NT_EXPORT_NAME, but the mechanism contained within the *input_name* is not supported.
GSS_S_FAILURE	The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   gss_release_buffer(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_import_sec_context – import security context established by another process

Synopsis cc [*flag...*] *file...* -lgss [*library...*]
#include <gssapi/gssapi.h>

OM_uint32 gss_import_sec_context(OM_uint32 *minor_status*,
    const gss_buffer_t *interprocess_token*,gss_ctx_id_t *context_handle*);

Description The gss_import_sec_context() function allows a process to import a security context established by another process. A given interprocess token can be imported only once. See gss_export_sec_context(3GSS).

Parameters The parameter descriptions for gss_import_sec_context() are as follows:

minor_status   A mechanism-specific status code.

interprocess_token   Token received from exporting process.

context_handle   Context handle of newly reactivated context. Resources associated with this context handle must be released by the application after use with a call to gss_delete_sec_context(3GSS).

Errors gss_import_sec_context() returns one of the following status codes:

GSS_S_COMPLETE   Successful completion.

GSS_S_NO_CONTEXT   The token did not contain a valid context reference.

GSS_S_DEFECTIVE_TOKEN   The token was invalid.

GSS_S_UNAVAILABLE   The operation is unavailable.

GSS_S_UNAUTHORIZED   Local policy prevents the import of this context by the current process.

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also gss_accept_sec_context(3GSS), gss_context_time(3GSS), gss_delete_sec_context(3GSS), gss_export_sec_context(3GSS), gss_init_sec_context(3GSS), gss_inquire_context(3GSS), gss_process_context_token(3GSS), gss_wrap_size_limit(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_indicate_mechs – determine available security mechanisms

Synopsis cc [*flag...*] *file...* -lgss [*library...*]
#include <gssapi/gssapi.h>

OM_uint32 gss_indicate_mechs(OM_uint32 **minor_status*,
    gss_OID_set **mech_set*);

Description The gss_indicate_mechs() function enables an application to determine available underlying security mechanisms.

Parameters The parameter descriptions for gss_indicate_mechs() follow:

minor_status   A mechanism-specific status code.

mech_set   Set of supported mechanisms. The returned gss_OID_set value will be a dynamically-allocated OID set that should be released by the caller after use with a call to gss_release_oid_set(3GSS).

Errors The gss_indicate_mechs() function may return the following status codes:

GSS_S_COMPLETE   Successful completion.

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also gss_release_oid_set(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_init_sec_context – initiate a GSS-API security context with a peer application

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

```
OM_uint32 gss_init_sec_context(OM_uint32 *minor_status,
    const gss_cred_id_t initiator_cred_handle,
    gss_ctx_id_t *context_handle, const gss_name_t *target_name,
    const gss_OID mech_type, OM_uint32 req_flags,
    OM_uint32 time_req, const gss_channel_bindings_t input_chan_bindings,
    const gss_buffer_t input_token, gss_OID *actual_mech_type,
    gss_buffer_t output_token, OM_uint32 *ret_flags,
    OM_uint32 *time_rec);
```

Parameters The parameter descriptions for gss_init_sec_context() follow:

minor_status
  A mechanism specific status code.

initiator_cred_handle
  The handle for the credentials claimed. Supply GSS_C_NO_CREDENTIAL to act as a default
  initiator principal. If no default initiator is defined, the function returns GSS_S_NO_CRED.

context_handle
  The context handle for a new context. Supply the value GSS_C_NO_CONTEXT for the first call,
  and use the value returned in any continuation calls. The resources associated with
  context_handle must be released by the application after use by a call to
  gss_delete_sec_context(3GSS).

target_name
  The name of the context acceptor.

mech_type
  The object ID of the desired mechanism. To obtain a specific default, supply the value
  GSS_C_NO_OID.

req_flags
  Contains independent flags, each of which will request that the context support a specific
  service option. A symbolic name is provided for each flag. Logically-OR the symbolic name
  to the corresponding required flag to form the bit-mask value. *req_flags* may contain one of
  the following values:

  GSS_C_DELEG_FLAG
    If true, delegate credentials to a remote peer. Do not delegate the credentials if the value
    is false.

  GSS_C_MUTUAL_FLAG
    If true, request that the peer authenticate itself. If false, authenticate to the remote peer
    only.

GSS_C_REPLAY_FLAG
    If true, enable replay detection for messages protected with gss_wrap(3GSS) or
    gss_get_mic(3GSS). Do not attempt to detect replayed messages if false.

GSS_C_SEQUENCE_FLAG
    If true, enable detection of out-of-sequence protected messages. Do not attempt to
    detect out-of-sequence messages if false.

GSS_C_CONF_FLAG
    If true, request that confidential service be made available by means of gss_wrap(3GSS).
    If false, no per-message confidential service is required.

GSS_C_INTEG_FLAG
    If true, request that integrity service be made available by means of gss_wrap(3GSS) or
    gss_get_mic(3GSS). If false, no per-message integrity service is required.

GSS_C_ANON_FLAG
    If true, do not reveal the initiator's identify to the acceptor. If false, authenticate
    normally.

time_req
    The number of seconds for which the context will remain valid. Supply a zero value to
    time_req to request a default validity period.

input_chan_bindings
    Optional application-specified bindings. Allows application to securely bind channel
    identification information to the security context. Set to GSS_C_NO_CHANNEL_BINDINGS if
    you do not want to use channel bindings.

input_token
    Token received from the peer application. On the initial call, supply GSS_C_NO_BUFFER or a
    pointer to a buffer containing the value GSS_C_EMPTY_BUFFER.

actual_mech_type
    The actual mechanism used. The OID returned by means of this parameter will be pointer
    to static storage that should be treated as read-only. The application should not attempt to
    free it. To obtain a specific default, supply the value GSS_C_NO_OID. Specify NULL if the
    parameter is not required.

output_token
    The token to send to the peer application. If the length field of the returned buffer is zero,
    no token need be sent to the peer application. After use storage associated with this buffer
    must be freed by the application by a call to gss_release_buffer(3GSS).

ret_flags
    Contains various independent flags, each of which indicates that the context supports a
    specific service option. If not needed, specify NULL. Test the returned bit-mask *ret_flags*
    value against its symbolic name to determine if the given option is supported by the
    context. *ret_flags* may contain one of the following values:

GSS_C_DELEG_FLAG
 If true, credentials were delegated to the remote peer. If false, no credentials were
 delegated.

GSS_C_MUTUAL_FLAG
 If true, the remote peer authenticated itself. If false, the remote peer did not authenticate
 itself.

GSS_C_REPLAY_FLAG
 If true, replay of protected messages will be detected. If false, replayed messages will not
 be detected.

GSS_C_SEQUENCE_FLAG
 If true, out of sequence protected messages will be detected. If false, they will not be
 detected.

GSS_C_CONF_FLAG
 If true, confidential service may be invoked by calling the gss_wrap() routine. If false,
 no confidentiality service is available by means of gss_wrap(3GSS). gss_wrap() will
 provide message encapsulation, data-origin authentication and integrity services only.

GSS_C_INTEG_FLAG
 If true, integrity service may be invoked by calling either the gss_wrap(3GSS) or
 gss_get_mic(3GSS) routine. If false, per-message integrity service is not available.

GSS_C_ANON_FLAG
 If true, the initiator's identity has not been revealed; it will not be revealed if any emitted
 token is passed to the acceptor. If false, the initiator has been or will be authenticated
 normally.

GSS_C_PROT_READY_FLAG
 If true, the protection services specified by the states of GSS_C_CONF_FLAG and
 GSS_C_INTEG_FLAG are available if the accompanying major status return value is either
 GSS_S_COMPLETE or GSS_S_CONTINUE_NEEDED. If false, the protection services are
 available only if the accompanying major status return value is GSS_S_COMPLETE.

GSS_C_TRANS_FLAG
 If true, the resultant security context may be transferred to other processes by means of a
 call to gss_export_sec_context(3GSS). If false, the security context cannot be
 transferred.

time_rec
 The number of seconds for which the context will remain valid. Specify NULL if the
 parameter is not required.

Description The gss_init_sec_context() function initiates the establishment of a security context
 between the application and a remote peer. Initially, the *input_token* parameter should be
 specified either as GSS_C_NO_BUFFER, or as a pointer to a gss_buffer_desc object with a
 length field that contains a zero value. The routine may return a *output_token*, which should

be transferred to the peer application, which will present it to
gss_accept_sec_context(3GSS). If no token need be sent, gss_init_sec_context() will
indicate this by setting the length field of the *output_token* argument to zero. To complete
context establishment, one or more reply tokens may be required from the peer application; if
so, gss_init_sec_context() will return a status code that contains the supplementary
information bit GSS_S_CONTINUE_NEEDED. In this case, make another call to
gss_init_sec_context() when the reply token is received from the peer application and pass
the reply token to gss_init_sec_context() by means of the *input_token* parameter.

Construct portable applications to use the token length and return status to determine
whether to send or wait for a token.

Whenever the routine returns a major status that includes the value GSS_S_CONTINUE_NEEDED,
the context is not fully established, and the following restrictions apply to the output
parameters:

- The value returned by means of the *time_rec* parameter is undefined. Unless the
  accompanying *ret_flags* parameter contains the bit GSS_C_PROT_READY_FLAG, which
  indicates that per-message services may be applied in advance of a successful completion
  status, the value returned by means of the *actual_mech_type* parameter is undefined until
  the routine returns a major status value of GSS_S_COMPLETE.

- The values of the GSS_C_DELEG_FLAG, GSS_C_MUTUAL_FLAG, GSS_C_REPLAY_FLAG,
  GSS_C_SEQUENCE_FLAG, GSS_C_CONF_FLAG, GSS_C_INTEG_FLAG and GSS_C_ANON_FLAG bits
  returned by the *ret_flags* parameter contain values that will be valid if context
  establishment succeeds. For example, if the application requests a service such as
  delegation or anonymous authentication by means of the *req_flags* argument, and the
  service is unavailable from the underlying mechanism, gss_init_sec_context()
  generates a token that will not provide the service, and it indicate by means of the *ret_flags*
  argument that the service will not be supported. The application may choose to abort
  context establishment by calling gss_delete_sec_context(3GSS) if it cannot continue
  without the service, or if the service was merely desired but not mandatory, it may transmit
  the token and continue context establishment.

- The values of the GSS_C_PROT_READY_FLAG and GSS_C_TRANS_FLAG bits within *ret_flags*
  indicate the actual state at the time gss_init_sec_context() returns, whether or not the
  context is fully established.

- The GSS-API sets the GSS_C_PROT_READY_FLAG in the final *ret_flags* returned to a caller,
  for example, when accompanied by a GSS_S_COMPLETE status code. However, applications
  should not rely on this behavior, as the flag was not defined in Version 1 of the GSS-API.
  Instead, applications should determine what per-message services are available after a
  successful context establishment according to the GSS_C_INTEG_FLAG and
  GSS_C_CONF_FLAG values.

- All other bits within the *ret_flags* argument are set to zero.

If the initial call of gss_init_sec_context() fails, the GSS-API does not create a context
object; it leaves the value of the *context_handle* parameter set to GSS_C_NO_CONTEXT to indicate

this. In the event of failure on a subsequent call, the GSS-API leaves the security context untouched for the application to delete using gss_delete_sec_context(3GSS).

During context establishment, the informational status bits GSS_S_OLD_TOKEN and GSS_S_DUPLICATE_TOKEN indicate fatal errors, and GSS-API mechanisms should always return them in association with a status code of GSS_S_FAILURE. This pairing requirement was not part of Version 1 of the GSS-API specification, so applications that wish to run on Version 1 implementations must special-case these codes.

Errors   gss_init_sec_context() may return the following status codes:

GSS_S_COMPLETE
   Successful completion.

GSS_S_CONTINUE_NEEDED
   A token from the peer application is required to complete the context, and gss_init_sec_context() must be called again with that token.

GSS_S_DEFECTIVE_TOKEN
   Consistency checks performed on the *input_token* failed.

GSS_S_DEFECTIVE_CREDENTIAL
   Consistency checks performed on the credential failed.

GSS_S_NO_CRED
   The supplied credentials are not valid for context acceptance, or the credential handle does not reference any credentials.

GSS_S_CREDENTIALS_EXPIRED
   The referenced credentials have expired.

GSS_S_BAD_BINDINGS
   The *input_token* contains different channel bindings than those specified by means of the *input_chan_bindings* parameter.

GSS_S_BAD_SIG
   The *input_token* contains an invalid MIC or a MIC that cannot be verified.

GSS_S_OLD_TOKEN
   The *input_token* is too old. This is a fatal error while establishing context.

GSS_S_DUPLICATE_TOKEN
   The *input_token* is valid, but it is a duplicate of a token already processed. This is a fatal error while establishing context.

GSS_S_NO_CONTEXT
   The supplied context handle does not refer to a valid context.

GSS_S_BAD_NAMETYPE
   The provided *target_name* parameter contains an invalid or unsupported *name* type.

GSS_S_BAD_NAME
   The supplied *target_name* parameter is ill-formed.

GSS_S_BAD_MECH
   The token received specifies a mechanism that is not supported by the implementation or
   the provided credential.

GSS_S_FAILURE
   The underlying mechanism detected an error for which no specific GSS status code is
   defined. The mechanism-specific status code reported by means of the *minor_status*
   parameter details the error condition.

Examples   EXAMPLE 1   Invoking gss_init_sec_context() Within a Loop

A typical portable caller should always invoke gss_init_sec_context() within a loop:

```
int context_established = 0;
gss_ctx_id_t context_hdl = GSS_C_NO_CONTEXT;
        ...
input_token->length = 0;

while (!context_established) {
  maj_stat = gss_init_sec_context(&min_stat,
                                  cred_hdl,
                                  &context_hdl,
                                  target_name,
                                  desired_mech,
                                  desired_services,
                                  desired_time,
                                  input_bindings,
                                  input_token,
                                  &actual_mech,
                                  output_token,
                                  &actual_services,
                                  &actual_time);
  if (GSS_ERROR(maj_stat)) {
    report_error(maj_stat, min_stat);
  };

  if (output_token->length != 0) {
    send_token_to_peer(output_token);
    gss_release_buffer(&min_stat, output_token)
  };
  if (GSS_ERROR(maj_stat)) {

    if (context_hdl != GSS_C_NO_CONTEXT)
      gss_delete_sec_context(&min_stat,
                             &context_hdl,
                             GSS_C_NO_BUFFER);
```

EXAMPLE 1 Invoking `gss_init_sec_context()` Within a Loop   *(Continued)*

```
    break;
  };
  if (maj_stat & GSS_S_CONTINUE_NEEDED) {
    receive_token_from_peer(input_token);
  } else {
    context_established = 1;
  };
};
```

Attributes   See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   `gss_delete_sec_context(3GSS)`, `gss_export_sec_context(3GSS)`, `gss_get_mic(3GSS)`, `gss_wrap(3GSS)`, `attributes(5)`

Developer's Guide to Oracle Solaris 11 Security

Name gss_inquire_context – obtain information about a security context

Synopsis cc [*flag...*] *file...* -lgss [*library...*]
#include <gssapi/gssapi.h>

OM_uint32 gss_inquire_context(OM_uint32 *_minor_status_,
    const gss_ctx_id_t *context_handle*,gss_name_t *_src_name_*,
    gss_name_t *_targ_name_*, OM_uint32 *_lifetime_rec_*,
    gss_OID *_mech_type_*, OM_uint32 *_ctx_flags_*,
    int *_locally_initiated_*, int *_open_*);

Description The gss_inquire_context() function obtains information about a security context. The caller must already have obtained a handle that refers to the context, although the context need not be fully established.

Parameters The parameter descriptions for gss_inquire_context() are as follows:

minor_status	A mechanism-specific status code.
context_handle	A handle that refers to the security context.
src_name	The name of the context initiator. If the context was established using anonymous authentication, and if the application invoking gss_inquire_context() is the context acceptor, an anonymous name is returned. Storage associated with this name must be freed by the application after use with a call to gss_release_name(). Specify NULL if the parameter is not required.
targ_name	The name of the context acceptor. Storage associated with this name must be freed by the application after use with a call to gss_release_name(). If the context acceptor did not authenticate itself, and if the initiator did not specify a target name in its call to gss_init_sec_context(), the value GSS_C_NO_NAME is returned. Specify NULL if the parameter is not required.
lifetime_rec	The number of seconds for which the context will remain valid. If the context has expired, this parameter will be set to zero. Specify NULL if the parameter is not required.
mech_type	The security mechanism providing the context. The returned OID is a pointer to static storage that should be treated as read-only by the application; in particular, the application should not attempt to free it. Specify NULL if the parameter is not required.
ctx_flags	Contains various independent flags, each of which indicates that the context supports (or is expected to support, if ctx_open is false) a specific service option. If not needed, specify NULL. Symbolic names are provided for each flag, and the symbolic names corresponding to the required flags

should be logically ANDed with the ret_flags value to test whether a given option is supported by the context. The flags are:

GSS_C_DELEG_FLAG   If true, credentials were delegated from the initiator to the acceptor. If false, no credentials were delegated.

GSS_C_MUTUAL_FLAG   If true, the acceptor was authenticated to the initiator. If false, the acceptor did not authenticate itself.

GSS_C_REPLAY_FLAG   If true, the replay of protected messages will be detected. If false, replayed messages will not be detected.

GSS_C_SEQUENCE_FLAG   If true, out-of-sequence protected messages will be detected. If false, out-of-sequence messages will not be detected.

GSS_C_CONF_FLAG   If true, confidential service may be invoked by calling the gss_wrap(3GSS) routine. If false, no confidential service is available through gss_wrap(). gss_wrap() provides message encapsulation, data-origin authentication, and integrity services only.

GSS_C_INTEG_FLAG   If true, integrity service can be invoked by calling either the gss_get_mic() or the gss_wrap() routine. If false, per-message integrity service is unavailable.

GSS_C_ANON_FLAG   If true, the initiator's identity is not revealed to the acceptor. The *src_name* parameter, if requested, contains an anonymous internal name. If false, the initiator has been authenticated normally.

GSS_C_PROT_READY_FLAG   If true, the protection services, as specified by the states of the GSS_C_CONF_FLAG and GSS_C_INTEG_FLAG, are available for use. If false, they are available only if the context is fully established, that is, if the *open* parameter is non-zero.

GSS_C_TRANS_FLAG   If true, resultant security context can be transferred to other processes through a call to gss_export_sec_context(). If

false, the security context is not
transferable.

locally_initiated   Non-zero if the invoking application is the context initiator. Specify NULL
if the parameter is not required.

open   Non-zero if the context is fully established; zero if a
context-establishment token is expected from the peer application.
Specify NULL if the parameter is not required.

Errors gss_inquire_context() returns one of the following status codes:

GSS_S_COMPLETE   Successful completion.

GSS_S_NO_CONTEXT   The referenced context could not be accessed.

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific
GSS status code is defined. The mechanism-specific status code
reported by means of the *minor_status* parameter details the error
condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also gss_accept_sec_context(3GSS), gss_context_time(3GSS),
gss_delete_sec_context(3GSS), gss_export_sec_context(3GSS),
gss_import_sec_context(3GSS), gss_init_sec_context(3GSS),
gss_process_context_token(3GSS), gss_wrap(3GSS), gss_wrap_size_limit(3GSS),
attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_inquire_cred – obtain information about a credential

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

OM_uint32 gss_inquire_cred(OM_uint32 *minor_status*,
    const gss_cred_id_t *cred_handle*,gss_name_t *name*,
    OM_uint32 *lifetime*, gss_cred_usage_t *cred_usage*,
    gss_OID_set *mechanisms*);

Parameters The parameter descriptions for gss_inquire_cred() follow:

minor_status   Mechanism specific status code.

cred_handle   Handle that refers to the target credential. Specify GSS_C_NO_CREDENTIAL to
                  inquire about the default initiator principal.

name   Name of the identity asserted by the credential. Any storage associated with
                  this name should be freed by the application after use by a call to
                  gss_release_name(3GSS).

lifetime   Number of seconds for which the credential remains valid. If the credential
                  has expired, this parameter will be set to zero. Specify NULL if the parameter
                  is not required.

cred_usage   Flag that indicates how a credential is used. The *cred_usage* parameter may
                  contain one of the following values: GSS_C_INITIATE, GSS_C_ACCEPT, or
                  GSS_C_BOTH. Specify NULL if this parameter is not required.

mechanisms   Set of mechanisms supported by the credential. Storage for the returned
                  OID-set must be freed by the application after use by a call to
                  gss_release_oid_set(3GSS). Specify NULL if this parameter is not
                  required.

Description Use the gss_inquire_cred() function to obtain information about a credential.

Return Values The gss_inquire_cred() function can return the following status codes:

GSS_S_COMPLETE   Successful completion.

GSS_S_NO_CRED   The referenced credentials could not be accessed.

GSS_S_DEFECTIVE_CREDENTIAL   The referenced credentials were invalid.

GSS_S_CREDENTIALS_EXPIRED   The referenced credentials have expired. If the *lifetime*
                             parameter was not passed as NULL, it will be set to 0.

GSS_S_FAILURE   The underlying mechanism detected an error for which
                             no specific GSS status code is defined. The
                             mechanism-specific status code reported by means of the
                             minor_status parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also gss_release_name(3GSS), gss_release_oid_set(3GSS), libgss(3LIB), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_inquire_cred_by_mech – obtain per-mechanism information about a credential

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

OM_uint32 gss_inquire_cred_by_mech(OM_uint32 *minor_status*,
    const gss_cred_id_t *cred_handle*,const gss_OID *mech_type*,
    gss_name_t *name*, OM_uint32 *initiator_lifetime*,
    OM_uint32 *acceptor_lifetime*, gss_cred_usage_t *cred_usage*);

Parameters *acceptor_lifetime*   The number of seconds that the credential is capable of accepting
security contexts under the specified mechanism. If the credential can
no longer be used to accept contexts, or if the credential usage for this
mechanism is GSS_C_INITIATE, this parameter will be set to 0. Specify
NULL if this parameter is not required.

cred_handle   A handle that refers to the target credential. Specify
GSS_C_NO_CREDENTIAL to inquire about the default initiator principal.

cred_usage   How the credential may be used with the specified mechanism. The
cred_usage parameter may contain one of the following values:
GSS_C_INITIATE, GSS_C_ACCEPT, or GSS_C_BOTH. Specify NULL if this
parameter is not required.

initiator_lifetime   The number of seconds that the credential is capable of initiating
security contexts under the specified mechanism. If the credential can
no longer be used to initiate contexts, or if the credential usage for this
mechanism is GSS_C_ACCEPT, this parameter will be set to 0. Specify
NULL if this parameter is not required.

mech_type   The mechanism for which the information should be returned.

minor_status   A mechanism specific status code.

name   The name whose identity the credential asserts. Any storage associated
with this *name* must be freed by the application after use by a call to
gss_release_name(3GSS).

Description The gss_inquire_cred_by_mech() function obtains per-mechanism information about a
credential.

Errors The gss_inquire_cred_by_mech() function can return the following status codes:

GSS_S_COMPLETE   Successful completion.

GSS_S_CREDENTIALS_EXPIRED   The credentials cannot be added because they have
expired.

GSS_S_DEFECTIVE_CREDENTIAL   The referenced credentials are invalid.

GSS_S_FAILURE   The underlying mechanism detected an error for which
                                 no specific GSS status code is defined. The
                                 mechanism-specific status code reported by means of the
                                 minor_status parameter details the error condition.

GSS_S_NO_CRED   The referenced credentials cannot be accessed.

GSS_S_UNAVAILABLE   The gss_inquire_cred_by_mech() function is not
                                 available for the specified mechanism type.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also gss_release_name(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_inquire_mechs_for_name – list mechanisms that support the specified name-type

Synopsis cc [*flag ...*] *file...* -lgss [*library ...*]
#include <gssapi/gssapi.h>

OM_uint32 gss_inquire_mechs_for_name(OM_uint32 **minor_status*,
    const gss_name_t *input_name*,gss_OID_set **mech_types*);

Description The gss_inquire_mechs_for_name() function returns the set of mechanisms supported by
the GSS-API that may be able to process the specified name. Each mechanism returned will
recognize at least one element within the internal name.

Some implementations of the GSS-API may perform this test by checking nametype
information contained within the passed name and registration information provided by
individual mechanisms. This means that the *mech_types* set returned by the function may
indicate that a particular mechanism will understand the name, when in fact the mechanism
would refuse to accept the name as input to gss_canonicalize_name(3GSS),
gss_init_sec_context(3GSS), gss_acquire_cred(3GSS), or gss_add_cred(3GSS), due to
some property of the name itself rather than the name-type. Therefore, this function should be
used only as a pre-filter for a call to a subsequent mechanism-specific function.

Parameters The parameter descriptions for gss_inquire_mechs_for_name() follow in alphabetical
order:

minor_status	Mechanism-specific status code.
input_name	The name to which the inquiry relates.
mech_types	Set of mechanisms that may support the specified name. The returned OID set must be freed by the caller after use with a call to gss_release_oid_set(3GSS).

Errors The gss_inquire_mechs_for_name() function may return the following status codes:

GSS_S_COMPLETE	Successful completion.
GSS_S_BAD_NAME	The *input_name* parameter was ill-formed.
GSS_S_BAD_NAMETYPE	The *input_name* parameter contained an invalid or unsupported type of name.
GSS_S_FAILURE	The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   gss_acquire_cred(3GSS), gss_add_cred(3GSS), gss_canonicalize_name(3GSS), gss_init_sec_context(3GSS), gss_release_oid_set(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name   gss_inquire_names_for_mech – list the name-types supported by the specified mechanism

Synopsis   cc [*flag ...*] *file...* -lgss [*library ...*]
#include <gssapi/gssapi.h>

OM_uint32 gss_inquire_names_for_mech(OM_uint32 *minor_status*,
const gss_OID *mechanism*,gss_OID_set *name_types*);

Description   The gss_inquire_names_for_mech() function returns the set of name-types supported by the specified mechanism.

Parameters   The parameter descriptions for gss_inquire_names_for_mech() follow:

minor_status   A mechanism-specific status code.

mechanism   The mechanism to be interrogated.

name_types   Set of name-types supported by the specified mechanism. The returned OID set must be freed by the application after use with a call to gss_release_oid_set(3GSS).

Errors   The gss_inquire_names_for_mech() function may return the following values:

GSS_S_COMPLETE   Successful completion.

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   gss_release_oid_set(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_oid_to_str – convert an OID to a string

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

gss_oid_to_str(OM_uint32 *minor_status, const gss_OID *oid*,
    gss_buffer_t *oid_str*);

Parameters *minor_status* Status code returned by underlying mechanism.

oid GSS-API OID structure to convert.

oid_str String to receive converted OID.

Description The gss_oid_to_str() function converts a GSS-API OID structure to a string. You can use the function to convert the name of a mechanism from an OID to a simple string. This function is a convenience function, as is its complementary function, gss_str_to_oid(3GSS).

If an OID must be created, use gss_create_empty_oid_set(3GSS) and gss_add_oid_set_member(3GSS) to create it. OIDs created in this way must be released with gss_release_oid_set(3GSS). However, it is strongly suggested that applications use the default GSS-API mechanism instead of creating an OID for a specific mechanism.

Errors The gss_oid_to_str() function returns one of the following status codes:

GSS_S_CALL_INACCESSIBLE_READ
    A required input parameter could not be read.

GSS_S_CALL_INACCESSIBLE_WRITE
    A required output parameter could not be written.

GSS_S_COMPLETE
    Successful completion.

GSS_S_FAILURE
    The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also gss_add_oid_set_member(3GSS), gss_create_empty_oid_set(3GSS), gss_release_oid_set(3GSS), gss_str_to_oid(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Warnings This function is included for compatibility only with programs using earlier versions of the GSS-API and should not be used for new programs. Other implementations of the GSS-API might not support this function, so portable programs should not rely on it. Sun might not continue to support this function.

Name gss_process_context_token – pass asynchronous token to security service

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

OM_uint32 gss_process_context_token(OM_uint32 **minor_status*,
    const gss_ctx_id_t *context_handle*,const gss_buffer_t *token_buffer*);

Description The gss_process_context_token() function provides a way to pass an asynchronous token to the security service. Most context-level tokens are emitted and processed synchronously by gss_init_sec_context() and gss_accept_sec_context(), and the application is informed as to whether further tokens are expected by the GSS_C_CONTINUE_NEEDED major status bit. Occasionally, a mechanism might need to emit a context-level token at a point when the peer entity is not expecting a token. For example, the initiator's final call to gss_init_sec_context() may emit a token and return a status of GSS_S_COMPLETE, but the acceptor's call to gss_accept_sec_context() might fail. The acceptor's mechanism might want to send a token containing an error indication to the initiator, but the initiator is not expecting a token at this point, believing that the context is fully established. gss_process_context_token() provides a way to pass such a token to the mechanism at any time.

This function is provided for compatibility with the GSS-API version 1. Because gss_delete_sec_context() no longer returns a valid *output_token* to be sent to gss_process_context_token(), applications using a newer version of the GSS-API do not need to rely on this function.

Parameters The parameter descriptions for gss_process_context_token() are as follows:

minor_status   A mechanism-specific status code.

context_handle   Context handle of context on which token is to be processed.

token_buffer   Token to process.

Errors gss_process_context_token() returns one of the following status codes:

GSS_S_COMPLETE   Successful completion.

GSS_S_DEFECTIVE_TOKEN   Indicates that consistency checks performed on the token failed.

GSS_S_NO_CONTEXT   The *context_handle* did not refer to a valid context.

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also gss_accept_sec_context(3GSS), gss_delete_sec_context(3GSS), gss_init_sec_context(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_release_buffer – free buffer storage allocated by a GSS-API function

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

OM_uint32 gss_release_buffer(OM_uint32 *minor_status*, gss_buffer_t*buffer*);

Description The gss_release_buffer() function frees buffer storage allocated by a GSS-API function. The gss_release_buffer() function also zeros the length field in the descriptor to which the buffer parameter refers, while the GSS-API function sets the pointer field in the descriptor to NULL. Any buffer object returned by a GSS-API function may be passed to gss_release_buffer(), even if no storage is associated with the buffer.

Parameters The parameter descriptions for gss_release_buffer() follow:

minor_status   Mechanism-specific status code.

buffer   The storage associated with the buffer will be deleted. The gss_buffer_desc() object will not be freed; however, its length field will be zeroed.

Errors The gss_release_buffer() function may return the following status codes:

GSS_S_COMPLETE   Successful completion

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name   gss_release_cred – discard a credential handle

Synopsis   cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

OM_uint32 gss_release_cred(OM_uint32 *minor_status*,
    gss_cred_id_t *cred_handle*);

Description   The gss_release_cred() function informs the GSS-API that the specified credential handle is no longer required by the application and frees the associated resources. The *cred_handle* parameter is set to GSS_C_NO_CREDENTIAL when this call completes successfully.

Parameters   The parameter descriptions for gss_release_cred() follow:

minor_status   A mechanism specific status code.

cred_handle   An opaque handle that identifies the credential to be released. If GSS_C_NO_CREDENTIAL is specified, the gss_release_cred() function will complete successfully, but it will do nothing.

Errors   gss_release_cred() may return the following status codes:

GSS_S_COMPLETE   Successful completion.

GSS_S_NO_CRED   The referenced credentials cannot be accessed.

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_release_name – discard an internal-form name

Synopsis cc [*flag ...*] *file*... -lgss [*library ...*]
#include <gssapi/gssapi.h

OM_uint32 gss_release_name(OM_uint32 *minor_status*, gss_name_t *name*);

Description The gss_release_name() function frees GSS-API-allocated storage associated with an internal-form name. The *name* is set to GSS_C_NO_NAME on successful completion of this call.

Parameters The parameter descriptions for gss_release_name() follow:

minor_status   A mechanism-specific status code.

name   The name to be deleted.

Errors The gss_release_name() function may return the following status codes:

GSS_S_COMPLETE   Successful completion.

GSS_S_BAD_NAME   The *name* parameter did not contain a valid name.

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_release_oid – release an object identifier

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

gss_release_oid(OM_uint32 *minor_status*, const gss_OID *oid*);

Description The gss_release_oid() function deletes an OID. Such an OID might have been created with
gss_str_to_oid().

Since creating and deleting individual OIDs is discouraged, it is preferable to use
gss_release_oid_set() if it is necessary to deallocate a set of OIDs.

Parameters The parameter descriptions for gss_release_oid() are as follows:

minor_status   A mechanism-specific status code.

oid   The object identifier of the mechanism to be deleted.

Errors gss_release_oid() returns one of the following status codes:

GSS_S_COMPLETE   Successful completion.

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific GSS
                  status code is defined. The mechanism-specific status code reported by
                  means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also gss_release_oid_set(3GSS), gss_str_to_oid(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Warnings This function is included for compatibility only with programs using earlier versions of the
GSS-API and should not be used for new programs. Other implementations of the GSS-API
might not support this function, so portable programs should not rely on it. Sun might not
continue to support this function.

Name gss_release_oid_set – free storage associated with a GSS-API-generated gss_OID_set object

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

OM_uint32 gss_release_oid_set(OM_uint32 *minor_status*, gss_OID_set *set*);

Description The gss_release_oid_set() function frees storage associated with a GSS-API-generated
gss_OID_set object. The *set* parameter must refer to an OID-set that was returned from a
GSS-API function. The gss_release_oid_set() function will free the storage associated with
each individual member OID, the OID *set*'s elements array, and gss_OID_set_desc.

gss_OID_set is set to GSS_C_NO_OID_SET on successful completion of this function.

Parameters The parameter descriptions for gss_release_oid_set() follow:

minor_status   A mechanism-specific status code

set   Storage associated with the gss_OID_set will be deleted

Errors The gss_release_oid_set() function may return the following status codes:

GSS_S_COMPLETE   Successful completion

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific GSS
                   status code is defined. The mechanism-specific status code reported by
                   means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name   gss_store_cred – store a credential in the current credential store

Synopsis   cc [*flag...*] *file...* -lgss [*library...*]
#include <gssapi/gssapi.h>

OM_uint32 gss_store_cred(OM_uint32 *minor_status*,
    const gss_cred_id_t *input_cred*, const gss_cred_usage_t *cred_usage*,
    const gss_OID *desired_mech*, OM_uint32 *overwrite_cred*,
    OM_uint32 *default_cred*, gss_OID_set *elements_stored*,
    gss_cred_usage_t *cred_usage_stored*);

Parameters   The parameter descriptions for gss_store_cred() follow:

input_cred	The credential to be stored.
cred_usage	This parameter specifies whether to store an initiator, an acceptor, or both usage components of a credential.
desired_mech	The mechanism-specific component of a credential to be stored. If GSS_C_NULL_OID is specified, the gss_store_cred() function attempts to store all the elements of the given *input_cred_handle*.
	The gss_store_cred() function is not atomic when storing multiple elements of a credential. All delegated credentials, however, contain a single element.
overwrite_cred	A boolean that indicates whether to overwrite existing credentials in the current store for the same principal as that of the *input_cred_handle*. A non-zero value indicates that credentials are overwritten. A zero value indicates that credentials are not overwritten.
default_cred	A boolean that indicates whether to set the principal name of the *input_cred_handle* parameter as the default of the current credential store. A non-zero value indicates that the principal name is set as the default. A zero value indicates that the principal name is not set as the default. The default principal of a credential store matches GSS_C_NO_NAME as the *desired_name* input parameter for gss_store_cred(3GSS).
elements_stored	The set of mechanism OIDs for which *input_cred_handle* elements have been stored.
cred_usage_stored	The stored *input_cred_handle* usage elements: initiator, acceptor, or both.
minor_status	Minor status code that is specific to one of the following: the mechanism identified by the *desired_mech_element* parameter, or the element of a single mechanism in the *input_cred_handle*. In all other cases, *minor_status* has an undefined value on return.

Description The gss_store_cred() function stores a credential in the the current GSS-API credential store for the calling process. Input credentials can be re-acquired through gss_add_cred(3GSS) and gss_acquire_cred(3GSS).

The gss_store_cred() function is specifically intended to make delegated credentials available to a user's login session.

The gss_accept_sec_context() function can return a delegated GSS-API credential to its caller. The function does not store delegated credentials to be acquired through gss_add_cred(3GSS). Delegated credentials can be used only by a receiving process unless they are made available for acquisition by calling the gss_store_cred() function.

The Solaris Operating System supports a single GSS-API credential store per user. The current GSS-API credential store of a process is determined by its effective UID.

In general, acceptor applications should switch the current credential store by changing the effective UID before storing a delegated credential.

Return Values The gss_store_cred() can return the following status codes:

GSS_S_COMPLETE	Successful completion.
GSS_S_CREDENTIALS_EXPIRED	The credentials could not be stored because they have expired.
GSS_S_CALL_INACCESSIBLE_READ	No input credentials were given.
GSS_S_UNAVAILABLE	The credential store is unavailable.
GSS_S_DUPLICATE_ELEMENT	The credentials could not be stored because the *overwrite_cred* input parameter was set to false (0) and the *input_cred* parameter conflicts with a credential in the current credential store.
GSS_S_FAILURE	The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Uncommitted
MT-Level	Safe

See Also `gss_accept_sec_context(3GSS)`, `gss_acquire_cred(3GSS)`, `gss_add_cred(3GSS)`, `gss_init_sec_context(3GSS)`, `gss_inquire_cred(3GSS)`, `gss_release_cred(3GSS)`, `gss_release_oid_set(3GSS)`, `attributes(5)`

Developer's Guide to Oracle Solaris 11 Security

Name   gss_str_to_oid – convert a string to an OID

Synopsis   cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

OM_uint32 gss_str_to_oid(OM_uint32 *minor_status*,
    const gss_buffer_t *oid_str*,gss_OID *oid*);

Description   The gss_str_to_oid() function converts a string to a GSS-API OID structure. You can use
the function to convert a simple string to an OID to . This function is a convenience function,
as is its complementary function, gss_oid_to_str(3GSS).

OIDs created with gss_str_to_oid() must be deallocated through gss_release_oid(3GSS),
if available. If an OID must be created, use gss_create_empty_oid_set(3GSS) and
gss_add_oid_set_member(3GSS) to create it. OIDs created in this way must be released with
gss_release_oid_set(3GSS). However, it is strongly suggested that applications use the
default GSS-API mechanism instead of creating an OID for a specific mechanism.

Parameters   The parameter descriptions for gss_str_to_oid() are as follows:

minor_status   Status code returned by underlying mechanism.

oid   GSS-API OID structure to receive converted string.

oid_str   String to convert.

Errors   gss_str_to_oid() returns one of the following status codes:

GSS_S_CALL_INACCESSIBLE_READ   A required input parameter could not be read.

GSS_S_CALL_INACCESSIBLE_WRITE   A required output parameter could not be written.

GSS_S_COMPLETE   Successful completion.

GSS_S_FAILURE   The underlying mechanism detected an error for
                                 which no specific GSS status code is defined. The
                                 mechanism-specific status code reported by means of
                                 the *minor_status* parameter details the error
                                 condition.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also   gss_add_oid_set_member(3GSS), gss_create_empty_oid_set(3GSS),
gss_oid_to_str(3GSS), gss_release_oid_set(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Warnings This function is included for compatibility only with programs using earlier versions of the GSS-API and should not be used for new programs. Other implementations of the GSS-API might not support this function, so portable programs should not rely on it. Sun might not continue to support this function.

Name gss_test_oid_set_member – interrogate an object identifier set

Synopsis cc [*flag...*] *file...* -lgss [*library...*]
#include <gssapi/gssapi.h>

OM_uint32 gss_test_oid_set_member(OM_uint32 **minor_status*,
    const gss_OID *member*,const gss_OID_set *set*,
    int **present*);

Description The gss_test_oid_set_member() function interrogates an object identifier set to determine
if a specified object identifier is a member. This function should be used with OID sets
returned by gss_indicate_mechs(3GSS), gss_acquire_cred(3GSS), and
gss_inquire_cred(3GSS), but it will also work with user-generated sets.

Parameters The parameter descriptions for gss_test_oid_set_member() follow:

minor_status   A mechanism-specific status code

member   An object identifier whose presence is to be tested

set   An object identifier set.

present   The value of *present* is non-zero if the specified OID is a member of the set; if
not, the value of *present* is zero.

Errors The gss_test_oid_set_member() function may return the following status codes:

GSS_S_COMPLETE   Successful completion

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific GSS
status code is defined. The mechanism-specific status code reported by
means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also gss_acquire_cred(3GSS), gss_indicate_mechs(3GSS), gss_inquire_cred(3GSS),
attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name   gss_unwrap – verify a message with attached cryptographic message

Synopsis   cc [*flag...*] *file...* -lgss [*library...*]
#include <gssapi/gssapi.h>

OM_uint32 gss_unwrap(OM_uint32 *minor_status*,
    const gss_ctx_id_t *context_handle*,
    const gss_buffer_t *input_message_buffer*,
    gss_buffer_t *output_message_buffer*, int *conf_state*,
    gss_qop_t *qop_state*);

Description   The gss_unwrap() function converts a message previously protected by gss_wrap(3GSS) back to a usable form, verifying the embedded MIC. The *conf_state* parameter indicates whether the message was encrypted; the *qop_state* parameter indicates the strength of protection that was used to provide the confidentiality and integrity services.

Since some application-level protocols may wish to use tokens emitted by gss_wrap(3GSS) to provide secure framing, the GSS-API supports the wrapping and unwrapping of zero-length messages.

Parameters   The parameter descriptions for gss_unwrap() follow:

minor_status	The status code returned by the underlying mechanism.
context_handle	Identifies the context on which the message arrived.
input_message_buffer	The message to be protected.
output_message_buffer	The buffer to receive the unwrapped message. Storage associated with this buffer must be freed by the application after use with a call to gss_release_buffer(3GSS).
conf_state	If the value of *conf_state* is non-zero, then confidentiality and integrity protection were used. If the value is zero, only integrity service was used. Specify NULL if this parameter is not required.
qop_state	Specifies the quality of protection provided. Specify NULL if this parameter is not required.

Errors   gss_unwrap() may return the following status codes:

GSS_S_COMPLETE	Successful completion.
GSS_S_DEFECTIVE_TOKEN	The token failed consistency checks.
GSS_S_BAD_SIG	The MIC was incorrect.
GSS_S_DUPLICATE_TOKEN	The token was valid, and contained a correct MIC for the message, but it had already been processed.
GSS_S_OLD_TOKEN	The token was valid, and contained a correct MIC for the message, but it is too old to check for duplication.

GSS_S_UNSEQ_TOKEN	The token was valid, and contained a correct MIC for the message, but has been verified out of sequence; a later token has already been received.
GSS_S_GAP_TOKEN	The token was valid, and contained a correct MIC for the message, but has been verified out of sequence; an earlier expected token has not yet been received.
GSS_S_CONTEXT_EXPIRED	The context has already expired.
GSS_S_NO_CONTEXT	The *context_handle* parameter did not identify a valid context.
GSS_S_FAILURE	The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   gss_release_buffer(3GSS), gss_wrap(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_verify_mic – verify integrity of a received message

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

OM_uint32 gss_verify_mic(OM_uint32 *minor_status*,
    const gss_ctx_id_t *context_handle*, const gss_buffer_t *message_buffer*,
    const gss_buffer_t *token_buffer*, gss_qop_t *qop_state*);

Description The gss_verify_mic() function verifies that a cryptographic MIC, contained in the token parameter, fits the supplied message. The *qop_state* parameter allows a message recipient to determine the strength of protection that was applied to the message.

Since some application-level protocols may wish to use tokens emitted by gss_wrap(3GSS) to provide secure framing, the GSS-API supports the calculation and verification of MICs over zero-length messages.

Parameters The parameter descriptions for gss_verify_mic() follow:

minor_status	The status code returned by the underlying mechanism.
context_handle	Identifies the context on which the message arrived.
message_buffer	The message to be verified.
token_buffer	The token associated with the message.
qop_state	Specifies the quality of protection gained from the MIC. Specify NULL if this parameter is not required.

Errors gss_verify_mic() may return the following status codes:

GSS_S_COMPLETE	Successful completion.
GSS_S_DEFECTIVE_TOKEN	The token failed consistency checks.
GSS_S_BAD_SIG	The MIC was incorrect.
GSS_S_DUPLICATE_TOKEN	The token was valid and contained a correct MIC for the message, but it had already been processed.
GSS_S_OLD_TOKEN	The token was valid and contained a correct MIC for the message, but it is too old to check for duplication.
GSS_S_UNSEQ_TOKEN	The token was valid and contained a correct MIC for the message, but it has been verified out of sequence; a later token has already been received.
GSS_S_GAP_TOKEN	The token was valid and contained a correct MIC for the message, but it has been verified out of sequence; an earlier expected token has not yet been received.
GSS_S_CONTEXT_EXPIRED	The context has already expired.

GSS_S_NO_CONTEXT   The *context_handle* parameter did not identify a valid context.

GSS_S_FAILURE   The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   gss_wrap(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name gss_wrap – attach a cryptographic message

Synopsis cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

OM_uint32 gss_wrap(OM_uint32 *minor_status*,
    const gss_ctx_id_t *context_handle*, int *conf_req_flag*,
    gss_qop_t *qop_req*, const gss_buffer_t *input_message_buffer*,
    int *conf_state*, gss_buffer_t *output_message_buffer*);

Description The gss_wrap() function attaches a cryptographic MIC and optionally encrypts the specified *input_message*. The *output_message* contains both the MIC and the message. The *qop_req* parameter allows a choice between several cryptographic algorithms, if supported by the chosen mechanism.

Since some application-level protocols may wish to use tokens emitted by gss_wrap() to provide secure framing, the GSS-API supports the wrapping of zero-length messages.

Parameters The parameter descriptions for gss_wrap() follow:

minor_status	The status code returned by the underlying mechanism.
context_handle	Identifies the context on which the message will be sent.
conf_req_flag	If the value of *conf_req_flag* is non-zero, both confidentiality and integrity services are requested. If the value is zero, then only integrity service is requested.
qop_req	Specifies the required quality of protection. A mechanism-specific default may be requested by setting *qop_req* to GSS_C_QOP_DEFAULT. If an unsupported protection strength is requested, gss_wrap() will return a *major_status* of GSS_S_BAD_QOP.
input_message_buffer	The message to be protected.
conf_state	If the value of *conf_state* is non-zero, confidentiality, data origin authentication, and integrity services have been applied. If the value is zero, then integrity services have been applied. Specify NULL if this parameter is not required.
output_message_buffer	The buffer to receive the protected message. Storage associated with this message must be freed by the application after use with a call to gss_release_buffer(3GSS).

Errors gss_wrap() may return the following status codes:

GSS_S_COMPLETE	Successful completion.
GSS_S_CONTEXT_EXPIRED	The context has already expired.

GSS_S_NO_CONTEXT	The *context_handle* parameter did not identify a valid context.
GSS_S_BAD_QOP	The specified QOP is not supported by the mechanism.
GSS_S_FAILURE	The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also gss_release_buffer(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name   gss_wrap_size_limit – allow application to determine maximum message size with resulting output token of a specified maximum size

Synopsis   cc [*flag*...] *file*... -lgss [*library*...]
#include <gssapi/gssapi.h>

OM_uint32 gss_process_context_token(OM_uint32 *minor_status*,
    const gss_ctx_id_t *context_handle*, int *conf_req_flag*,
    gss_qop_t *qop_req*, OM_uint32 *req_output_size*,
    OM_uint32 *max_input_size*);

Description   The gss_wrap_size_limit() function allows an application to determine the maximum message size that, if presented to gss_wrap() with the same *conf_req_flag* and *qop_req* parameters, results in an output token containing no more than *req_output_size* bytes. This call is intended for use by applications that communicate over protocols that impose a maximum message size. It enables the application to fragment messages prior to applying protection. The GSS-API detects invalid QOP values when gss_wrap_size_limit() is called. This routine guarantees only a maximum message size, not the availability of specific QOP values for message protection.

Successful completion of gss_wrap_size_limit() does not guarantee that gss_wrap() will be able to protect a message of length *max_input_size* bytes, since this ability might depend on the availability of system resources at the time that gss_wrap() is called.

Parameters   The parameter descriptions for gss_wrap_size_limit() are as follows:

   minor_status   A mechanism-specific status code.

   context_handle   A handle that refers to the security over which the messages will be sent.

   conf_req_flag   Indicates whether gss_wrap() will be asked to apply confidential protection in addition to integrity protection. See gss_wrap(3GSS) for more details.

   qop_req   Indicates the level of protection that gss_wrap() will be asked to provide. See gss_wrap(3GSS) for more details.

   req_output_size   The desired maximum size for tokens emitted by gss_wrap().

   max_input_size   The maximum input message size that can be presented to gss_wrap() to guarantee that the emitted token will be no larger than *req_output_size* bytes.

Errors   gss_wrap_size_limit() returns one of the following status codes:

   GSS_S_COMPLETE   Successful completion.

   GSS_S_NO_CONTEXT   The referenced context could not be accessed.

   GSS_S_CONTEXT_EXPIRED   The context has expired.

GSS_S_BAD_QOP	The specified QOP is not supported by the mechanism.
GSS_S_FAILURE	The underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism-specific status code reported by means of the *minor_status* parameter details the error condition.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also gss_wrap(3GSS), attributes(5)

Developer's Guide to Oracle Solaris 11 Security

Name htonl, htons, ntohl, ntohs – convert values between host and network byte order

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <arpa/inet.h>

uint32_t htonl(uint32_t *hostlong*);

uint16_t htons(uint16_t *hostshort*);

uint32_t ntohl(uint32_t *netlong*);

uint16_t ntohs(uint16_t *netshort*);

Description These functions convert 16-bit and 32-bit quantities between network byte order and host byte order.

The uint32_t and uint16_t types are made available by inclusion of <inttypes.h>.

Usage These functions are most often used in conjunction with Internet addresses and ports as returned by gethostent(3XNET) and getservent(3XNET).

On some architectures these functions are defined as macros that expand to the value of their argument.

Return Values The htonl() and htons() functions return the argument value converted from host to network byte order.

The ntohl() and ntohs() functions return the argument value converted from network to host byte order.

Errors No errors are defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also endhostent(3XNET), endservent(3XNET), attributes(5), standards(5)

Name   icmp6_filter – Variable allocation datatype

Synopsis   void ICMP6_FILTER_SETPASSALL (struct icmp6_filter *);

void ICMP6_FILTER_SETBLOCKALL (struct icmp6_filter *);

void ICMP6_FILTER_SETPASS (int, *struct icmp6_filter* *);

void ICMP6_FILTER_SETBLOCK (int, *struct icmp6_filter* *);

int ICMP6_FILTER_WILLPASS (int, *const struct icmp6_filter* *);

int ICMP6_FILTER_WILLBLOCK (int, *const struct icmp6_filter* *);

Description   The icmp6_filter structure is similar to the fd_set datatype used with the select()
function in the sockets API. The icmp6_filter structure is an opaque datatype and the
application should not care how it is implemented. The application allocates a variable of this
type, then passes a pointer to it. Next it passes a pointer to a variable of this type to
getsockopt() and setsockopt() and operates on a variable of this type using the six macros
defined below.

The SETPASSALL and SETBLOCKALL functions enable you to specify that all ICMPv6 messages
are passed to the application or that all ICMPv6 messages are blocked from being passed.

The SETPASS and SETBLOCKALL functions enable you to specify that messages of a given
ICMPv6 type should be passed to the application or not passed to the application (blocked).

The WILLPASS and WILLBLOCK return true or false depending whether the specified message
type is passed to the application or blocked from being passed to the application by the filter
pointed to by the second argument.

The pointer argument to all six icmp6_filter macros is a pointer to a filter that is modified by
the first four macros and is examined by ICMP6_FILTER_SETBLOCK and
ICMP6_FILTER_WILLBLOCK. The first argument, (an integer), to the ICMP6_FILTER_BLOCKALL,
ICMP6_FILTER_SETPASS, ICMP6_FILTER_SETBLOCK, and ICMP6_FILTER_WILLBLOCK macros is
an ICMPv6 message type, between 0 and 255.

The current filter is fetched and stored using getsockopt() and setsockopt() with a level of
IPPROTO_ICMPV6 and an option name of ICMP6_FILTER.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe
Interface Stability	Committed
Standard	See standards(5).

Name   if_nametoindex, if_indextoname, if_nameindex, if_freenameindex – routines to map Internet Protocol network interface names and interface indexes

Synopsis   cc [*flag*...] *file*... -lsocket [*library*...]
#include <net/if.h>

unsigned int if_nametoindex(const char *ifname);

char *if_indextoname(unsigned int *ifindex*, char *ifname);

struct if_nameindex *if_nameindex(void)

void if_freenameindex(struct if_nameindex *ptr);

Parameters   *ifname*   interface name

ifindex   interface index

ptr   pointer returned by if_nameindex()

Description   This API defines two functions that map between an Internet Protocol network interface name and index, a third function that returns all the interface names and indexes, and a fourth function to return the dynamic memory allocated by the previous function.

Network interfaces are normally known by names such as eri0, sl1, ppp2, and the like. The *ifname* argument must point to a buffer of at least IF_NAMESIZE bytes into which the interface name corresponding to the specified index is returned. IF_NAMESIZE is defined in <net/if.h> and its value includes a terminating null byte at the end of the interface name.

if_nametoindex()   The if_nametoindex() function returns the interface index corresponding to the interface name pointed to by the *ifname* pointer. If the specified interface name does not exist, the return value is 0, and errno is set to ENXIO. If there was a system error, such as running out of memory, the return value is 0 and errno is set to the proper value, for example, ENOMEM.

if_indextoname()   The if_indextoname() function maps an interface index into its corresponding name. This pointer is also the return value of the function. If there is no interface corresponding to the specified index, NULL is returned, and errno is set to ENXIO, if there was a system error, such as running out of memory, if_indextoname() returns NULL and errno would be set to the proper value, for example, ENOMEM.

if_nameindex()   The if_nameindex() function returns an array of if_nameindex structures, one structure per interface. The if_nameindex structure holds the information about a single interface and is defined when the <net/if.h> header is included:

```
struct if_nameindex
   unsigned int   if_index; /* 1, 2, ... */
   char          *if_name;  /* "net0", ... */
};
```

While any IPMP IP interfaces are returned by if_nameindex(), the underlying IP interfaces that comprise each IPMP group are not returned.

The end of the array of structures is indicated by a structure with an if_index of 0 and an if_name of NULL. The function returns a null pointer upon an error and sets errno to the appropriate value. The memory used for this array of structures along with the interface names pointed to by the if_name members is obtained dynamically. This memory is freed by the if_freenameindex() function.

if_freenameindex()   The if_freenameindex() function frees the dynamic memory that was allocated by if_nameindex(). The argument to this function must be a pointer that was returned by if_nameindex().

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also ifconfig(1M), if_nametoindex(3XNET), attributes(5), if(7P)

Name if_nametoindex, if_indextoname, if_nameindex, if_freenameindex – functions to map Internet Protocol network interface names and interface indexes

Synopsis cc [*flag*...] *file*... -lxnet [*library*...]
#include <net/if.h>

unsigned int if_nametoindex(const char *ifname*);

char *if_indextoname(unsigned int *ifindex*, char *ifname*);

struct if_nameindex *if_nameindex(void)

void if_freenameindex(struct if_nameindex *ptr*);

Parameters These functions support the following parameters:

ifname interface name

ifindex interface index

ptr pointer returned by if_nameindex()

Description This API defines two functions that map between an Internet Protocol network interface name and index, a third function that returns all the interface names and indexes, and a fourth function to return the dynamic memory allocated by the previous function.

Network interfaces are normally known by names such as eri0, sl1, ppp2, and the like. The *ifname* argument must point to a buffer of at least IF_NAMESIZE bytes into which the interface name corresponding to the specified index is returned. IF_NAMESIZE is defined in <net/if.h> and its value includes a terminating null byte at the end of the interface name.

if_nametoindex() The if_nametoindex() function returns the interface index corresponding to the interface name pointed to by the *ifname* pointer. If the specified interface name does not exist, the return value is 0, and errno is set to ENXIO. If there was a system error, such as running out of memory, the return value is 0 and errno is set to the proper value, for example, ENOMEM.

if_indextoname() The if_indextoname() function maps an interface index into its corresponding name. This pointer is also the return value of the function. If there is no interface corresponding to the specified index, NULL is returned, and errno is set to ENXIO, if there was a system error, such as running out of memory, if_indextoname() returns NULL and errno would be set to the proper value, for example, ENOMEM.

*if_nameindex() The if_nameindex() function returns an array of if_nameindex structures, one structure per interface. The if_nameindex structure holds the information about a single interface and is defined when the <net/if.h> header is included:

```
struct if_nameindex {
    unsigned int  if_index; /* 1, 2, ... */
    char         *if_name;  /* null terminated name: "eri0", ... */
};
```

The end of the array of structures is indicated by a structure with an if_index of 0 and an if_name of NULL. The function returns a null pointer upon an error and sets errno to the appropriate value. The memory used for this array of structures along with the interface names pointed to by the if_name members is obtained dynamically. This memory is freed by the if_freenameindex() function.

if_freenameindex()   The if_freenameindex() function frees the dynamic memory that was allocated by if_nameindex(). The argument to this function must be a pointer that was returned by if_nameindex().

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library (32-bit)
	SUNWcslx (64-bit)
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also ifconfig(1M), if_nametoindex(3SOCKET), attributes(5), standards(5), if(7P)

Name   inet, inet6, inet_ntop, inet_pton, inet_aton, inet_addr, inet_network, inet_makeaddr, inet_lnaof, inet_netof, inet_ntoa – Internet address manipulation

Synopsis   cc [*flag*...] *file*... -lsocket -lnsl [*library*...]
```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

const char *inet_ntop(int af, const void *addr, char *cp,
    socklen_t size);

int inet_pton(int af, const char *cp, void *addr);

int inet_aton(const char *cp, struct in_addr *addr);

in_addr_t inet_addr(const char *cp);

in_addr_t inet_network(const char *cp);

struct in_addr inet_makeaddr(const int net, in_addr_t lna);

in_addr_t inet_lnaof(struct in_addr in);

in_addr_t inet_netof(struct in_addr in);

char *inet_ntoa(struct in_addr in);
```

Description   The inet_ntop() and inet_pton() functions can manipulate both IPv4 and IPv6 addresses. The inet_aton(), inet_addr(), inet_network(), inet_makeaddr(), inet_lnaof(), inet_netof(), and inet_ntoa() functions can only manipulate IPv4 addresses.

The inet_ntop() function converts a numeric address into a string suitable for presentation. The *af* argument specifies the family of the address which can be AF_INET or AF_INET6. The *addr* argument points to a buffer that holds an IPv4 address if the *af* argument is AF_INET. The *addr* argument points to a buffer that holds an IPv6 address if the *af* argument is AF_INET6. The address must be in network byte order. The *cp* argument points to a buffer where the function stores the resulting string. The application must specify a non-NULL *cp* argument. The *size* argument specifies the size of this buffer. For IPv6 addresses, the buffer must be at least 46-octets. For IPv4 addresses, the buffer must be at least 16-octets. To allow applications to easily declare buffers of the proper size to store IPv4 and IPv6 addresses in string form, the following two constants are defined in <netinet/in.h>:

```
#define INET_ADDRSTRLEN     16
#define INET6_ADDRSTRLEN    46
```

The inet_pton() function converts the standard text presentation form of an address into its numeric binary form. The *af* argument specifies the family of the address. Currently, the AF_INET and AF_INET6 address families are supported. The *cp* argument points to the string being passed in. The *addr* argument points to a buffer where the function stores the numeric address. The calling application must ensure that the buffer referred to by *addr* is large enough to hold the numeric address, at least 4 bytes for AF_INET or 16 bytes for AF_INET6.

The inet_aton(), inet_addr(), and inet_network() functions interpret character strings that represent numbers expressed in the IPv4 standard '.' notation, returning numbers suitable for use as IPv4 addresses and IPv4 network numbers, respectively. The inet_makeaddr() function uses an IPv4 network number and a local network address to construct an IPv4 address. The inet_netof() and inet_lnaof() functions break apart IPv4 host addresses, then return the network number and local network address, respectively.

The inet_ntoa() function returns a pointer to a string in the base 256 notation d.d.d.d. See the following section on IPv4 addresses.

Internet addresses are returned in network order, bytes ordered from left to right. Network numbers and local address parts are returned as machine format integer values.

IPv6 Addresses   There are three conventional forms for representing IPv6 addresses as strings:

1. The preferred form is x:x:x:x:x:x:x:x, where the 'x's are the hexadecimal values of the eight 16-bit pieces of the address. For example:

   1080:0:0:0:8:800:200C:417A

   It is not necessary to write the leading zeros in an individual field. There must be at least one numeral in every field, except when the special syntax described in the following is used.

2. It is common for addresses to contain long strings of zero bits in some methods used to allocate certain IPv6 address styles. A special syntax is available to compress the zeros. The use of "::" indicates multiple groups of 16 bits of zeros. The :: may only appear once in an address. The :: can also be used to compress the leading and trailing zeros in an address. For example:

   1080::8:800:200C:417A

3. The alternative form x:x:x:x:x:x:d.d.d.d is sometimes more convenient when dealing with a mixed environment of IPv4 and IPv6 nodes. The x's in this form represent the hexadecimal values of the six high-order 16-bit pieces of the address. The d's represent the decimal values of the four low-order 8-bit pieces of the standard IPv4 address. For example:

   ::FFFF:129.144.52.38
   ::129.144.52.38

   The ::FFFF:d.d.d.d and ::d.d.d.d pieces are the general forms of an IPv4–mapped IPv6 address and an IPv4–compatible IPv6 address.

   The IPv4 portion must be in the d.d.d.d form. The following forms are invalid:

   ::FFFF:d.d.d
   ::FFFF:d.d
   ::d.d.d
   ::d.d

The `::FFFF:d` form is a valid but unconventional representation of the IPv4–compatible IPv6 address `::255.255.0.d`.

The `::d` form corresponds to the general IPv6 address `0:0:0:0:0:0:0:d`.

IPv4 Addresses   Values specified using '`.`' notation take one of the following forms:

```
d.d.d.d
d.d.d
d.d
d
```

When four parts are specified, each part is interpreted as a byte of data and assigned from left to right to the four bytes of an IPv4 address.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right most two bytes of the network address. The three part address format is convenient for specifying Class B network addresses such as `128.net.host`.

When a two-part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. The two part address format is convenient for specifying Class A network addresses such as `net.host`.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

With the exception of `inet_pton()`, numbers supplied as *parts* in '`.`' notation may be decimal, octal, or hexadecimal, as specified in C language. For example, a leading `0x` or `0X` implies hexadecimal. A leading `0` implies octal. Otherwise, the number is interpreted as decimal.

For IPv4 addresses, `inet_pton()` accepts only a string in standard IPv4 dot notation:

```
d.d.d.d
```

Each number has one to three digits with a decimal value between 0 and 255.

The `inet_addr()` function has been obsoleted by `inet_aton()`.

Return Values   The `inet_aton()` function returns nonzero if the address is valid, `0` if the address is invalid.

The `inet_ntop()` function returns a pointer to the buffer that contains a string if the conversion succeeds. Otherwise, `NULL` is returned. Upon failure, `errno` is set to `EAFNOSUPPORT` if the *af* argument is invalid or `ENOSPC` if the size of the result buffer is inadequate.

The `inet_pton()` function returns 1 if the conversion succeeds, 0 if the input is not a valid IPv4 dotted-decimal string or a valid IPv6 address string. The function returns −1 with `errno` set to `EAFNOSUPPORT` if the af argument is unknown.

The value `INADDR_NONE`, which is equivalent to `(in_addr_t)(-1)`, is returned by `inet_addr()` and `inet_network()` for malformed requests.

The functions inet_netof() and inet_lnaof() break apart IPv4 host addresses, returning the network number and local network address part, respectively.

The function inet_ntoa() returns a pointer to a string in the base 256 notation d.d.d.d, described in the section on IPv4 addresses.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	See below.
MT-Level	Safe

The inet_ntop(), inet_pton(), inet_aton(), inet_addr(), and inet_network() functions are Committed. The inet_lnaof(), inet_makeaddr(), inet_netof(), and inet_network() functions are Committed (Obsolete).

See Also gethostbyname(3NSL), getipnodebyname(3SOCKET), getnetbyname(3SOCKET), inet.h(3HEAD), hosts(4), networks(4), attributes(5)

Notes The return value from inet_ntoa() points to a buffer which is overwritten on each call. This buffer is implemented as thread-specific data in multithreaded applications.

IPv4-mapped addresses are not recommended.

Bugs The problem of host byte ordering versus network byte ordering is confusing. A simple way to specify Class C network addresses in a manner similar to that for Class B and Class A is needed.

Name   inet6_opt, inet6_opt_init, inet6_opt_append, inet6_opt_finish, inet6_opt_set_val,
          inet6_opt_next, inet6_opt_find, inet6_opt_get_val – Option manipulation mechanism

Synopsis   cc [*flag* ...] *file* ... -lsocket [*library...*]
          #include <netinet/in.h>

          int inet6_opt_init(void **extbuf*, socklen_t *extlen*);

          int inet6_opt_append(void **extbuf*, socklen_t *extlen*,
              int *offset*, uint8_t *type*, socklen_t *len*, uint_t *align*,
              void **databufp*);

          int inet6_opt_finish(void **extbuf*, socklen_t *extlen*,
              int *offset*);

          int inet6_opt_set_val(void **databuf*, int *offset*,
              void *val*, socklen_t *vallen*);

          int inet6_opt_next(void **extbuf*, socklen_t *extlen*,
              int *offset*, uint8_t *typep*, socklen_t *lenp*,
              void **databufp*);

          int inet6_opt_find(void **extbuf*, socklen_t *extlen*, int *offset*,
              uint8_t *type*, socklen_t *lenp*, void **databufp*);

          intinet6_opt_get_val(void **databuf*, int *offset*,
              void *val*, socklen_t *vallen*);

Description   The inet6_opt functions enable users to manipulate options without having to know the
          structure of the option header.

          The inet6_opt_init() function returns the number of bytes needed for the empty extension
          header, that is, without any options. If *extbuf* is not NULL, it also initializes the extension header
          to the correct length field. If the *extlen* value is not a positive non-zero multiple of 8, the
          function fails and returns −1.

          The inet6_opt_append() function returns the updated total length while adding an option
          with length len and alignment align. If *extbuf* is not NULL, then, in addition to returning the
          length, the function inserts any needed Pad option, initializes the option setting the type and
          length fields, and returns a pointer to the location for the option content in *databufp*. If the
          option does not fit in the extension header buffer, the function returns −1. The *type* is the 8–bit
          option type. The *len* is the length of the option data, excluding the option type and option
          length fields. Once inet6_opt_append() is called, the application can use the *databuf* directly,
          or inet6_opt_set_val() can be used to specify the content of the option. The option type
          must have a value from 2 to 255, inclusive. The values 0 and 1 are reserved for the Pad1 and
          PadN options, respectively. The option data length must have a value between 0 and 255,
          inclusive, and it is the length of the option data that follows. The align parameter must have a
          value of 1, 2, 4, or 8. The align value cannot exceed the value of len.

          The inet6_opt_finish() function returns the updated total length the takes into account the
          final padding of the extension header to make it a multiple of 8 bytes. If *extbuf* is not NULL, the

function also initializes the option by inserting a Pad1 or PadN option of the proper length. If the necessary pad does not fit in the extension header buffer, the function returns −1.

The inet6_opt_set_val() function inserts data items of various sizes in the data portion of the option. The *val* parameter should point to the data to be inserted. The *offset* specifies the data portion of the option in which the value should be inserted. The first byte after the option type and length is accessed by specifying an *offset* of zero.

The inet6_opt_next() function parses the received option extension headers which return the next option. The *extbuf* and *extlen* parameters specify the extension header. The *offset* should be zero for the first option or the length returned by a previous call to either inet6_opt_next() or inet6_opt_find(). The *offset* specifies where to continue scanning the extension buffer. The subsequent option is returned by updating *typep*, *lenp*, and *databufp*. The *typep* argument stores the option type. The *lenp* argument stores the length of the option data, excluding the option type and option length fields. The *databufp* argument points to the data field of the option.

The inet6_opt_find() function is similar to the inet6_opt_next() function. Unlike inet6_opt_next(), the inet6_opt_find() function enables the caller to specify the option type to be searched for, rather than returning the next option in the extension header.

The inet6_opt_get_val() function extracts data items of various sizes in the portion of the option. The *val* argument should point to the destination for the extracted data. The *offset* specifies at which point in the option's data portion the value should be extracted. The first byte following the option type and length is accessed by specifying an *offset* of zero.

Return Values The inet6_opt_init() function returns the number of bytes needed for the empty extension header. If the *extlen* value is not a positive non-zero multiple of 8, the function fails and returns −1.

The inet6_opt_append() function returns the updated total length.

The inet6_opt_finish() function returns the updated total length.

The inet6_opt_set_val() function returns the *offset* for the subsequent field.

The inet6_opt_next() function returns the updated "previous" length computed by advancing past the option that was returned. When there are no additional options or if the option extension header is malformed, the return value is −1.

The inet6_opt_find() function returns the updated "previous" total length. If an option of the specified type is not located, the return value is −1. If the option extension header is malformed, the return value is −1.

The inet6_opt_get_val() function returns the *offset* for the next field (that is, *offset* + *vallen*) which can be used when extracting option content with multiple fields.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See standards(5).

See Also RFC 3542 – *Advanced Sockets Application Programming Interface (API) for IPv6*, The Internet Society. May 2003

Name   inet6_rth, inet6_rth_space, inet6_rth_init, inet6_rth_add, inet6_rth_reverse, inet6_rth_segments, inet6_rth_getaddr – Routing header manipulation

Synopsis   cc [*flag* ...] *file* ... -lsocket [*library*]
#include <netinet/in.h>

socklen_t inet6_rth_space(int *type*, int *segments*);

void *inet6_rth_init(void *bp*, socklen_t *bp_len*, int *type*, int *segments*);

int inet6_rth_add(void *bp*, const *struct*, in6_addr *addr*);

int inet6_rth_reverse(const void *in*, void *out*);

int inet6_rth_segments(const void *bp*);

struct in6_addr *inet6_rth_getaddr(const void *bp*, int *index*);

Description   The inet6_rth functions enable users to manipulate routing headers without having knowledge of their structure.

The iet6_rth_init() function initializes the buffer pointed to by *bp* to contain a routing header of the specified type and sets ip6r_len based on the segments parameter. The *bp_len* argument is used only to verify that the buffer is large enough. The ip6r_segleft field is set to zero and inet6_rth_add() increments it. The caller allocates the buffer and its size can be determined by calling inet6_rth_space().

The inet6_rth_add() function adds the IPv6 address pointed to by *addr* to the end of the routing header that is being constructed.

The inet6_rth_reverse() function takes a routing header extension header pointed to by the first argument and writes a new routing header that sends datagrams along the reverse of the route. The function reverses the order of the addresses and sets the segleft member in the new routing header to the number of segments. Both arguments can point to the same buffer (that is, the reversal can occur in place).

The inet6_rth_segments() function returns the number of segments (addresses) contained in the routing header described by *bp*.

The inet6_rth_getaddr() function returns a pointer to the IPv6 address specified by index, which must have a value between 0 and one less than the value returned by inet6_rth_segments() in the routing header described by *bp*. Applications should first call inet6_rth_segments() to obtain the number of segments in the routing header.

The inet6_rth_space() function returns the size, but the function does not allocate the space required for the ancillary data routing header.

Routing Headers   To receive a routing header, the application must enable the IPV6_RECVRTHDR socket option:

```
int  on = 1;
setsockopt (fd, IPPROTO_IPV6, IPV6_RECVRTHDR, &on, sizeof(on));
```

Each received routing header is returned as one ancillary data object described by a cmsghdr structure with cmsg_type set to IPV6_RTHDR.

To send a routing header, the application specifies it either as ancillary data in a call to sendmsg() or by using setsockopt(). For the sending side, this API assumes the number of occurrences of the routing header as described in *RFC-2460*. Applications can specify no more than one outgoing routing header.

The application can remove any sticky routing header by calling setsockopt() for IPV6_RTHDR with a zero option length.

When using ancillary data, a routing header is passed between the application and the kernel as follows: The cmsg_level member has a value of IPPROTO_IPV6 and the cmsg_type member has a value of IPV6_RTHDR. The contents of the cmsg_data member is implementation-dependent and should not be accessed directly by the application, but should be accessed using the inet6_rth functions.

The following constant is defined as a result of including the <netinet/in.h>:

```
#define IPV6_RTHDR_TYPE_0   0 /* IPv6 Routing header type 0 */
```

ROUTING HEADER OPTION
Source routing in IPv6 is accomplished by specifying a routing header as an extension header. There are a number of different routing headers, but IPv6 currently defines only the Type 0 header. See *RFC-2460*. The Type 0 header supports up to 127 intermediate nodes, limited by the length field in the extension header. With this maximum number of intermediate nodes, a source, and a destination, there are 128 hops.

Return Values
The inet6_rth_init() function returns a pointer to the buffer (*bp*) upon success.

For the inet6_rth_add() function, the segleft member of the routing header is updated to account for the new address in the routing header. The function returns 0 upon success and −1 upon failure.

The inet6_rth_reverse() function returns 0 upon success or −1 upon an error.

The inet6_rth_segments() function returns 0 or greater upon success and −1 upon an error.

The inet6_rth_getaddr() function returns NULL upon an error.

The inet6_rth_space() function returns the size of the buffer needed for the routing header.

Attributes
See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Standard	See standards(5).

See Also RFC 3542– *Advanced Sockets Application Programming Interface (API) for IPv6*, The Internet Society. May 2003

Name   inet_addr, inet_network, inet_makeaddr, inet_lnaof, inet_netof, inet_ntoa – Internet address manipulation

Synopsis   cc [*flag* ...] *file* ... -lxnet [*library* ...]
```
#include <arpa/inet.h>

in_addr_t inet_addr(const char *cp);

in_addr_t inet_lnaof(struct in_addr in);

struct in_addr inet_makeaddr(in_addr_t net, in_addr_t lna);

in_addr_t inet_netof(struct in_addr in);

in_addr_t inet_network(const char *cp);

char *inet_ntoa(struct in_addr in);
```

Description   The inet_addr() function converts the string pointed to by *cp*, in the Internet standard dot notation, to an integer value suitable for use as an Internet address.

The inet_lnaof() function takes an Internet host address specified by *in* and extracts the local network address part, in host byte order.

The inet_makeaddr() function takes the Internet network number specified by *net* and the local network address specified by *lna*, both in host byte order, and constructs an Internet address from them.

The inet_netof() function takes an Internet host address specified by *in* and extracts the network number part, in host byte order.

The inet_network() function converts the string pointed to by *cp*, in the Internet standard dot notation, to an integer value suitable for use as an Internet network number.

The inet_ntoa() function converts the Internet host address specified by *in* to a string in the Internet standard dot notation.

All Internet addresses are returned in network order (bytes ordered from left to right).

Values specified using dot notation take one of the following forms:

a.b.c.d   When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

a.b.c   When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the rightmost two bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as 128.*net.host*.

a.b   When a two-part address is supplied, the last part is interpreted as a 24-bit
            quantity and placed in the rightmost three bytes of the network address. This
            makes the two-part address format convenient for specifying Class A network
            addresses as *net.host*.

a   When only one part is given, the value is stored directly in the network address
            without any byte rearrangement.

All numbers supplied as parts in dot notation may be decimal, octal, or hexadecimal, that is, a
leading 0x or 0X implies hexadecimal, as specified in the *ISO C* standard; otherwise, a leading
0 implies octal; otherwise, the number is interpreted as decimal.

Usage   The return value of inet_ntoa() may point to static data that may be overwritten by
subsequent calls to inet_ntoa().

Return Values   Upon successful completion, inet_addr() returns the Internet address. Otherwise, it returns
(in_addr_t)(−1).

Upon successful completion, inet_network() returns the converted Internet network
number. Otherwise, it returns (in_addr_t)(−1).

The inet_makeaddr() function returns the constructed Internet address.

The inet_lnaof() function returns the local network address part.

The inet_netof() function returns the network number.

The inet_ntoa() function returns a pointer to the network address in Internet-standard dot
notation.

Errors   No errors are defined.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also   endhostent(3XNET), endnetent(3XNET), attributes(5), standards(5)

Name inet_cidr_ntop, inet_cidr_pton – network translation routines

Synopsis cc [*flag*...] *file*... -lresolv -lsocket -lnsl [*library*...]
```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char *inet_cidr_ntop(int af, const void *src, int bits, char *dst,
    size_t size);

int inet_cidr_pton(int af, const char *src, void *dst, int *bits);
```

Description These routines are used for converting addresses to and from network and presentation forms with CIDR (Classless Inter-Domain Routing) representation, embedded net mask.

The inet_cidr_ntop() function converts an address from network to presentation format.

The *af* parameter describes the type of address that is being passed in *src*. Currently only AF_INET is supported.

The *src* parameter is an address in network byte order, its length is determined from *af*.

The *bits* parameter specifies the number of bits in the netmask unless it is -1 in which case the CIDR representation is omitted.

The *dst* parameter is a caller supplied buffer of at least *size* bytes.

The inet_cidr_ntop() function returns *dst* on success or NULL. Check errno for reason.

The inet_cidr_pton() function converts and address from presentation format, with optional CIDR representation, to network format. The resulting address is zero filled if there were insufficient bits in *src*.

The *af* parameter describes the type of address that is being passed in via *src* and determines the size of *dst*.

The *src* parameter is an address in presentation format.

The *bits* parameter returns the number of bits in the netmask or -1 if a CIDR representation was not supplied.

The inet_cidr_pton() function returns 0 on success or -1 on error. Check errno for reason. ENOENT indicates an invalid netmask.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	**ATTRIBUTE VALUE**
Interface Stability	Committed
MT-Level	MT-Safe

See Also   Intro(2), attributes(5)

Name inet_ntop, inet_pton – convert IPv4 and IPv6 addresses between binary and text form

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <arpa/inet.h>

const char *inet_ntop(int *af*, const void *restrict *src*,
    char *restrict *dst*, socklen_t *size*);

int inet_pton(int *af*, const char *restrict *src*, *dst*);

Description The inet_ntop() function converts a numeric address into a text string suitable for presentation. The *af* argument specifies the family of the address. This can be AF_INET or AF_INET6. The *src* argument points to a buffer holding an IPv4 address if the *af* argument is AF_INET, or an IPv6 address if the *af* argument is AF_INET6. The *dst* argument points to a buffer where the function stores the resulting text string; it cannot be NULL. The *size* argument specifies the size of this buffer, which must be large enough to hold the text string (INET_ADDRSTRLEN characters for IPv4, INET6_ADDRSTRLEN characters for IPv6).

The inet_pton() function converts an address in its standard text presentation form into its numeric binary form. The *af* argument specifies the family of the address. The AF_INET and AF_INET6 address families are supported. The *src* argument points to the string being passed in. The *dst* argument points to a buffer into which the function stores the numeric address; this must be large enough to hold the numeric address (32 bits for AF_INET, 128 bits for AF_INET6).

If the *af* argument of inet_pton() is AF_INET, the *src* string is in the standard IPv4 dotted-decimal form:

ddd.ddd.ddd.ddd

where "ddd" is a one to three digit decimal number between 0 and 255 (see inet_addr(3XNET)). The inet_pton() function does not accept other formats (such as the octal numbers, hexadecimal numbers, and fewer than four numbers that inet_addr() accepts).

If the *af* argument of inet_pton() is AF_INET6, the *src* string is in one of the following standard IPv6 text forms:

1. The preferred form is "x:x:x:x:x:x:x:x", where the 'x's are the hexadecimal values of the eight 16-bit pieces of the address. Leading zeros in individual fields can be omitted, but there must be at least one numeral in every field.

2. A string of contiguous zero fields in the preferred form can be shown as "::". The "::" can only appear once in an address. Unspecified addresses ("0:0:0:0:0:0:0:0") can be represented simply as "::".

3. A third form that is sometimes more convenient when dealing with a mixed environment of IPv4 and IPv6 nodes is "x:x:x:x:x:x:d.d.d.d", where the 'x's are the hexadecimal values of the six high-order 16-bit pieces of the address, and the 'd's are the decimal values of the four low-order 8-bit pieces of the address (standard IPv4 representation).

A more extensive description of the standard representations of IPv6 addresses can be found in RFC 2373.

Return Values The inet_ntop() function returns a pointer to the buffer containing the text string if the conversion succeeds. Otherwise it returns NULL and sets errno to indicate the error.

The inet_pton() function returns 1 if the conversion succeeds, with the address pointed to by *dst* in network byte order. It returns 0 if the input is not a valid IPv4 dotted-decimal string or a valid IPv6 address string. It returns −1 and sets errno to EAFNOSUPPORT if the *af* argument is unknown.

Errors The inet_ntop() and inet_pton() functions will fail if:

EAFNOSUPPORT The *af* argument is invalid.

ENOSPC The size of the inet_ntop() result buffer is inadequate.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also inet_addr(3XNET), attributes(5)

Name ldap – Lightweight Directory Access Protocol package

Synopsis cc[*flag*...] *file*... -lldap[*library*...]
#include <lber.h>
#include <ldap.h>

Description The Lightweight Directory Access Protocol ("LDAP") package (SUNWlldap) includes various command line LDAP clients and a LDAP client library to provide programmatic access to the LDAP protocol. This man page gives an overview of the LDAP client library functions.

An application might use the LDAP client library functions as follows. The application would initialize a LDAP session with a LDAP server by calling ldap_init(3LDAP). Next, it authenticates to the LDAP server by calling ldap_sasl_bind(3LDAP) and friends. It may perform some LDAP operations and obtain results by calling ldap_search(3LDAP) and friends. To parse the results returned from these functions, it calls ldap_parse_result(3LDAP),ldap_next_entry(3LDAP), and ldap_first_entry(3LDAP) and others. It closes the LDAP session by calling ldap_unbind(3LDAP).

LDAP operations can be either synchronous or asynchronous. By convention, the names of the sychronous functions end with "_s." For example, a synchronous binding to the LDAP server can be performed by calling ldap_sasl_bind_s(3LDAP). Complete an asynchronous binding with ldap_sasl_bind(3LDAP). All synchronous functions return the actual outcome of the operation, either LDAP_SUCCESS or an error code. Asynchronous routines provide an invocation identifier which can be used to obtain the result of a specific operation by passing it to theldap_result(3LDAP) function.

Initializing a LDAP session Initializing a LDAP session involves calling the ldap_init(3LDAP) function. However, the call does not actually open a connection to the LDAP server. It merely initializes a LDAP structure that represents the session. The connection is opened when the first operation is attempted. Unlike ldap_init(), ldap_open(3LDAP) attempts to open a connection with the LDAP server. However, the use of ldap_open() is deprecated.

Authenticating to a LDAP server The ldap_sasl_bind(3LDAP) and ldap_sasl_bind_s(3LDAP) functions provide general and extensible authenticaton for an LDAP client to a LDAP server. Both use the Simple Authentication Security Layer (SASL). Simplified routines ldap_simple_bind(3LDAP) and ldap_simple_bind_s(3LDAP) use cleartext passwords to bind to the LDAP server. Use of ldap_bind(3LDAP) and ldap_bind_s(3LDAP)(3LDAP) is deprecated.

Searching a LDAP directory Search for an entry in a LDAP directory by calling the ldap_search_ext(3LDAP) or the ldap_search_ext_s(3LDAP) functions. These functions support LDAPv3 server controls, client controls and variable size and time limits as arguments for each search operation. ldap_search(3LDAP) and ldap_search_s(3LDAP) are identical functions but do not support the controls and limits as arguments to the call.

Adding or Deleting an entry
Use ldap_add_ext(3LDAP) and ldap_delete_ext(3LDAP) to add or delete entries in a LDAP directory server. The synchronous counterparts to these functions are ldap_add_ext_s(3LDAP) and ldap_delete_ext_s(3LDAP). The ldap_add(3LDAP), ldap_add_s(3LDAP), ldap_delete(3LDAP), and ldap_delete_s(3LDAP) provide identical functionality to add and to delete entries, but they do not support LDAP v3 server and client controls.

Modifying Entries
Use ldap_modify_ext(3LDAP) and ldap_modify_ext_s(3LDAP) to modify an existing entry in a LDAP server that supports for LDAPv3 server and client controls. Similarly, use ldap_rename(3LDAP) and ldap_rename_s(3LDAP) to change the name of an LDAP entry. The ldap_modrdn(3LDAP), ldap_modrdn_s(3LDAP), ldap_modrdn2(3LDAP) and ldap_modrdn2_s(3LDAP) interfaces are deprecated.

Obtaining Results
Use ldap_result(3LDAP) to obtain the results of a previous asynchronous operation. For all LDAP operations other than search, only one message is returned. For the search operation, a list of result messages can be returned.

Handling Errors and Parsing Results
Use the ldap_parse_result(3LDAP), ldap_parse_sasl_bind_result(3LDAP), and the ldap_parse_extended_result(3LDAP) functions to extract required information from results and and to handle the returned errors. To covert a numeric error code into a null-terminated character string message describing the error, use ldap_err2string(3LDAP). The ldap_result2error(3LDAP) and ldap_perror(3LDAP) functions are deprecated. To step through the list of messages in a result returned by ldap_result(), use ldap_first_message(3LDAP) and ldap_next_message(3LDAP). ldap_count_messages(3LDAP) returns the number of messages contained in the list.

You can use ldap_first_entry(3LDAP) and ldap_next_entry(3LDAP) to step through and obtain a list of entries from a list of messages returned by a search result. ldap_count_entries(3LDAP) returns the number of entries contained in a list of messages. Call either ldap_first_attribute(3LDAP) and ldap_next_attribute(3LDAP) to step through a list of attributes associated with an entry. Retrieve the values of a given attribute by calling ldap_get_values(3LDAP) and ldap_get_values_len(3LDAP). Count the number of values returned by using ldap_count_values(3LDAP) and ldap_count_values_len(3LDAP).

Use the ldap_get_lang_values(3LDAP) and ldap_get_lang_values_len(3LDAP) to return an attribute's values that matches a specified language subtype. The ldap_get_lang_values() function returns an array of an attribute's string values that matches a specified language subtype. To retrieve the binary data from an attribute, call the ldap_get_lang_values_len() function instead.

Uniform Resource Locators (URLS)
You can use the ldap_url(3LDAP)functions to test a URL to verify that it is an LDAP URL, to parse LDAP URLs into their component pieces, to initiate searches directly using an LDAP URL, and to retrieve the URL associated with a DNS domain name or a distinguished name.

User Friendly Naming   The ldap_ufn(3LDAP) functions implement a user friendly naming scheme by means of
LDAP. This scheme allows you to look up entries using fuzzy, untyped names like "mark
smith, umich, us".

Caching   The ldap_memcache(3LDAP) functions provide an in-memory client side cache to store
search requests. Caching improves performance and reduces network bandwidth when a
client makes repeated requests.

Utility Functions   There are also various utility functions. You can use the ldap_sort(3LDAP) functions are
used to sort the entries and values returned by means of the ldap search functions. The
ldap_friendly(3LDAP) functions will map from short two letter country codes or other
strings to longer "friendlier" names. Use the ldap_charset(3LDAP) functions to translate to
and from the T.61 character set that is used for many character strings in the LDAP protocol.

Generating Filters   Make calls to ldap_init_getfilter(3LDAP) and ldap_search(3LDAP) to generate filters to
be used in ldap_search(3LDAP) and ldap_search_s(3LDAP). ldap_init_getfilter()
reads ldapfilter.conf(4), the LDAP configuration file, while ldap_init_getfilter_buf()
reads the configuration information from *buf* of length *buflen*.
ldap_getfilter_free(3LDAP) frees memory that has been allocated by means of
ldap_init_getfilter().

BER Library   The LDAP package includes a set of lightweight Basic Encoding Rules ("BER") functions. The
LDAP library functions use the BER functions to encode and decode LDAP protocol elements
through the slightly simplified BER defined by LDAP. They are not normally used directly by
an LDAP application program will not normally use the BER functions directly. Instead, these
functions provide a printf() and scanf()-like interface, as well as lower-level access.

List Of Interfaces

ldap_open(3LDAP)	Deprecated. Use ldap_init(3LDAP).
ldap_init(3LDAP)	Initialize a session with a LDAP server without opening a connection to a server.
ldap_result(3LDAP)	Obtain the result from a previous asynchronous operation.
ldap_abandon(3LDAP)	Abandon or abort an asynchronous operation.
ldap_add(3LDAP)	Asynchronously add an entry
ldap_add_s(3LDAP)	Synchronously add an entry.
ldap_add_ext(3LDAP)	Asynchronously add an entry with support for LDAPv3 controls.
ldap_add_ext_s(3LDAP)	Synchronously add an entry with support for LDAPv3 controls.
ldap_bind(3LDAP)	Deprecated. Use ldap_sasl_bind(3LDAP) or ldap_simple_bind(3LDAP).

ldap_sasl_bind(3LDAP)	Asynchronously bind to the directory using SASL authentication
ldap_sasl_bind_s(3LDAP)	Synchronously bind to the directory using SASL authentication
ldap_bind_s(3LDAP)	Deprecated. Use ldap_sasl_bind_s(3LDAP) or ldap_simple_bind_s(3LDAP).
ldap_simple_bind(3LDAP)	Asynchronously bind to the directory using simple authentication.
ldap_simple_bind_s(3LDAP)	Synchronously bind to the directory using simple authentication.
ldap_unbind(3LDAP)	Synchronously unbind from the LDAP server, close the connection, and dispose the session handle.
ldap_unbind_ext(3LDAP)	Synchronously unbind from the LDAP server and close the connection. ldap_unbind_ext() allows you to explicitly include both server and client controls in the unbind request.
ldap_set_rebind_proc(3LDAP)	Set callback function for obtaining credentials from a referral.
ldap_memcache_init(3LDAP)	Create the in-memory client side cache.
ldap_memcache_set(3LDAP)	Associate an in-memory cache that has been already created by calling the ldap_memcache_init(3LDAP) function with an LDAP connection handle.
ldap_memcache_get(3LDAP)	Get the cache associated with the specified LDAP structure.
ldap_memcache_flush(3LDAP)	Flushes search requests from the cache.
ldap_memcache_destroy(3LDAP)	Frees the specified LDAPMemCache structure pointed to by cache from memory.
ldap_memcache_update(3LDAP)	Checks the cache for items that have expired and removes them.
ldap_compare(3LDAP)	Asynchronous compare with a directory entry.
ldap_compare_s(3LDAP)	Synchronous compare with a directory entry.
ldap_compare_ext(3LDAP)	Asynchronous compare with a directory entry, with support for LDAPv3 controls.

ldap_compare_ext_s(3LDAP)	Synchronous compare with a directory entry, with support for LDAPv3 controls.
ldap_control_free(3LDAP)	Dispose of an LDAP control.
ldap_controls_free(3LDAP)	Dispose of an array of LDAP controls.
ldap_delete(3LDAP)	Asynchronously delete an entry.
ldap_delete_s(3LDAP)	Synchronously delete an entry.
ldap_delete_ext(3LDAP)	Asynchronously delete an entry, with support for LDAPv3 controls.
ldap_delete_ext_s(3LDAP)	Synchronously delete an entry, with support for LDAPv3 controls.
ldap_init_templates(3LDAP)	Read a sequence of templates from a LDAP template configuration file.
ldap_init_templates_buf(3LDAP)	Read a sequence of templates from a buffer.
ldap_free_templates(3LDAP)	Dispose of the templates allocated.
ldap_first_reference(3LDAP)	Step through a list of continuation references from a search result.
ldap_next_reference(3LDAP)	Step through a list of continuation references from a search result.
ldap_count_references(3LDAP)	Count the number of messages in a search result.
ldap_first_message(3LDAP)	Step through a list of messages in a search result.
ldap_count_messages(3LDAP)	Count the messages in a list of messages in a search result.
ldap_next_message(3LDAP)	Step through a list of messages in a search result.
ldap_msgtype(3LDAP)	Return the type of LDAP message.
ldap_first_disptmpl(3LDAP)	Get first display template in a list.
ldap_next_disptmpl(3LDAP)	Get next display template in a list.
ldap_oc2template(3LDAP)	Return template appropriate for the objectclass.
ldap_name2template(3LDAP)	Return named template
ldap_tmplattrs(3LDAP)	Return attributes needed by the template.

ldap_first_tmplrow(3LDAP)	Return first row of displayable items in a template.
ldap_next_tmplrow(3LDAP)	Return next row of displayable items in a template.
ldap_first_tmplcol(3LDAP)	Return first column of displayable items in a template.
ldap_next_tmplcol(3LDAP)	Return next column of displayable items in a template.
ldap_entry2text(3LDAP)	Display an entry as text by using a display template.
ldap_entry2text_search(3LDAP)	Search for and display an entry as text by using a display template.
ldap_vals2text(3LDAP)	Display values as text.
ldap_entry2html(3LDAP)	Display an entry as HTML (HyperText Markup Language) by using a display template.
ldap_entry2html_search(3LDAP)	Search for and display an entry as HTML by using a display template.
ldap_vals2html(3LDAP)	Display values as HTML.
ldap_perror(3LDAP)	Deprecated. Use ldap_parse_result(3LDAP).
ldap_result2error(3LDAP)	Deprecated. Use ldap_parse_result(3LDAP).
ldap_err2string(3LDAP)	Convert LDAP error indication to a string.
ldap_first_attribute(3LDAP)	Return first attribute name in an entry.
ldap_next_attribute(3LDAP)	Return next attribute name in an entry.
ldap_first_entry(3LDAP)	Return first entry in a chain of search results.
ldap_next_entry(3LDAP)	Return next entry in a chain of search results.
ldap_count_entries(3LDAP)	Return number of entries in a search result.
ldap_friendly_name(3LDAP)	Map from unfriendly to friendly names.
ldap_free_friendlymap(3LDAP)	Free resources used by ldap_friendly(3LDAP).
ldap_get_dn(3LDAP)	Extract the DN from an entry.
ldap_explode_dn(3LDAP)	Convert a DN into its component parts.

ldap_explode_dns(3LDAP)	Convert a DNS-style DN into its component parts (experimental).
ldap_is_dns_dn(3LDAP)	Check to see if a DN is a DNS-style DN (experimental).
ldap_dns_to_dn(3LDAP)	Convert a DNS domain name into an X.500 distinguished name.
ldap_dn2ufn(3LDAP)	Convert a DN into user friendly form.
ldap_get_values(3LDAP)	Return an attribute's values.
ldap_get_values_len(3LDAP)	Return an attribute's values with lengths.
ldap_value_free(3LDAP)	Free memory allocated by ldap_get_values(3LDAP).
ldap_value_free_len(3LDAP)	Free memory allocated by ldap_get_values_len(3LDAP).
ldap_count_values(3LDAP)	Return number of values.
ldap_count_values_len(3LDAP)	Return number of values.
ldap_init_getfilter(3LDAP)	Initialize getfilter functions from a file.
ldap_init_getfilter_buf(3LDAP)	Initialize getfilter functions from a buffer.
ldap_getfilter_free(3LDAP)	Free resources allocated by ldap_init_getfilter(3LDAP).
ldap_getfirstfilter(3LDAP)	Return first search filter.
ldap_getnextfilter(3LDAP)	Return next search filter.
ldap_build_filter(3LDAP)	Construct an LDAP search filter from a pattern.
ldap_setfilteraffixes(3LDAP)	Set prefix and suffix for search filters.
ldap_modify(3LDAP)	Asynchronously modify an entry.
ldap_modify_s(3LDAP)	Synchronously modify an entry.
ldap_modify_ext(3LDAP)	Asynchronously modify an entry, return value, and place message.
ldap_modify_ext_s(3LDAP)	Synchronously modify an entry, return value, and place message.
ldap_mods_free(3LDAP)	Free array of pointers to mod structures used by ldap_modify(3LDAP).

ldap_modrdn2(3LDAP)	Deprecated. Use ldap_rename(3LDAP) instead.
ldap_modrdn2_s(3LDAP)	Deprecated. Use ldap_rename_s(3LDAP) instead.
ldap_modrdn(3LDAP)	Deprecated. Use ldap_rename(3LDAP) instead.
ldap_modrdn_s(3LDAP)	Depreciated. Use ldap_rename_s(3LDAP) instead.
ldap_rename(3LDAP)	Asynchronously modify the name of an LDAP entry.
ldap_rename_s(3LDAP)	Synchronously modify the name of an LDAP entry.
ldap_msgfree(3LDAP)	Free result messages.
ldap_parse_result(3LDAP)	Search for a message to parse.
ldap_parse_extended_result(3LDAP)	Search for a message to parse.
ldap_parse_sasl_bind_result(3LDAP)	Search for a message to parse.
ldap_search(3LDAP)	Asynchronously search the directory.
ldap_search_s(3LDAP)	Synchronously search the directory.
ldap_search_ext(3LDAP)	Asynchronously search the directory with support for LDAPv3 controls.
ldap_search_ext_s(3LDAP)	Synchronously search the directory with support for LDAPv3 controls.
ldap_search_st(3LDAP)	Synchronously search the directory with support for a local timeout value.
ldap_ufn_search_s(3LDAP)	User friendly search the directory.
ldap_ufn_search_c(3LDAP)	User friendly search the directory with cancel.
ldap_ufn_search_ct(3LDAP)	User friendly search the directory with cancel and timeout.
ldap_ufn_setfilter(3LDAP)	Set filter file used by ldap_ufn(3LDAP) functions.
ldap_ufn_setprefix(3LDAP)	Set prefix used by ldap_ufn(3LDAP) functions.

ldap_ufn_timeout(3LDAP)	Set timeout used by ldap_ufn(3LDAP) functions.
ldap_is_ldap_url(3LDAP)	Check a URL string to see if it is an LDAP URL.
ldap_url_parse(3LDAP)	Break up an LDAP URL string into its components.
ldap_free_urldesc(3LDAP)	Free an LDAP URL structure.
ldap_url_search(3LDAP)	Asynchronously search by using an LDAP URL.
ldap_url_search_s(3LDAP)	Synchronously search by using an LDAP URL.
ldap_url_search_st(3LDAP)	Asynchronously search by using an LDAP URL, with support for a local timeout value.
ldap_dns_to_url(3LDAP)	Locate the LDAP URL associated with a DNS domain name.
ldap_dn_to_url(3LDAP)	Locate the LDAP URL associated with a distinguished name.
ldap_init_searchprefs(3LDAP)	Initialize searchprefs functions from a file.
ldap_init_searchprefs_buf(3LDAP)	Initialize searchprefs functions from a buffer.
ldap_free_searchprefs(3LDAP)	Free memory allocated by searchprefs functions.
ldap_first_searchobj(3LDAP)	Return first searchpref object.
ldap_next_searchobj(3LDAP)	Return next searchpref object.
ldap_sort_entries(3LDAP)	Sort a list of search results.
ldap_sort_values(3LDAP)	Sort a list of attribute values.
ldap_sort_strcasecmp(3LDAP)	Case insensitive string comparison.
ldap_set_string_translators(3LDAP)	Set character set translation functions used by LDAP library.
ldap_translate_from_t61(3LDAP)	Translate from the T.61 character set to another character set.
ldap_translate_to_t61(3LDAP)	Translate to the T.61 character set from another character set.
ldap_enable_translation(3LDAP)	Enable or disable character translation for an LDAP entry result.

ldap_version(3LDAP)	Get version information about the LDAP SDK for C.
ldap_get_lang_values(3LDAP)	Return an attribute's value that matches a specified language subtype.
ldap_get_lang_values_len(3LDAP)	Return an attribute's value that matches a specified language subtype along with lengths.
ldap_get_entry_controls(3LDAP)	Get the LDAP controls included with a directory entry in a set of search results.
ldap_get_option(3LDAP)	Get session preferences in an LDAP structure.
ldap_set_option(3LDAP)	Set session preferences in an LDAP structure.
ldap_memfree(3LDAP)	Free memory allocated by LDAP API functions.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also attributes(5)

Name   ldap_abandon – abandon an LDAP operation in progress

Synopsis   cc[*flag*...] *file*... -lldap[*library*...]
#include <lber.h>
#include <ldap.h>

int ldap_abandon(LDAP *ld, int *msgid*);

Description   The ldap_abandon() function is used to abandon or cancel an LDAP operation in progress.
The *msgid* passed should be the message id of an outstanding LDAP operation, as returned by
ldap_search(3LDAP), ldap_modify(3LDAP), etc.

ldap_abandon( ) checks to see if the result of the operation has already come in. If it has, it
deletes it from the queue of pending messages. If not, it sends an LDAP abandon operation to
the the LDAP server.

The caller can expect that the result of an abandoned operation will not be returned from a
future call to ldap_result(3LDAP).

Errors   ldap_abandon() returns 0 if successful or −1otherwise and setting *ld_errno* appropriately. See
ldap_error(3LDAP) for details.

Attributes   See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also   ldap(3LDAP), ldap_result(3LDAP), ldap_error(3LDAP), attributes(5)

Name ldap_add, ldap_add_s, ldap_add_ext, ldap_add_ext_s – perform an LDAP add operation

Synopsis cc [*flag*...] *file*... -lldap [*library*...]
#include <lber.h>
#include <ldap.h>

int ldap_add(LDAP *ld*, char *dn*, LDAPMod *attrs*[]);

int ldap_add_s(LDAP *ld*, char *dn*, LDAPMod *attrs*[]);

int ldap_add_ext(LDAP *ld*, char *dn*, LDAPMod **attrs*,
    LDAPControl **serverctrls*, int * *msgidp*);

int ldap_add_ext_s(LDAP *ld*, char *dn*, LDAPMod **attrs*,
    LDAPControl **serverctrls*, LDAPControl **clientctrls*);

Description The ldap_add_s() function is used to perform an LDAP add operation. It takes *dn*, the DN of
the entry to add, and *attrs*, a null-terminated array of the entry's attributes. The LDAPMod
structure is used to represent attributes, with the *mod_type* and *mod_values* fields being used
as described under ldap_modify(3LDAP), and the *ldap_op* field being used only if you need to
specify the LDAP_MOD_BVALUES option. Otherwise, it should be set to zero.

Note that all entries except that specified by the last component in the given DN must already
exist. ldap_add_s() returns an LDAP error code indicating success or failure of the operation.
See ldap_error(3LDAP) for more details.

The ldap_add() function works just like ldap_add_s(), but it is asynchronous. It returns the
message id of the request it initiated. The result of this operation can be obtained by calling
ldap_result(3LDAP).

The ldap_add_ext() function initiates an asynchronous add operation and returns
LDAP_SUCCESS if the request was successfully sent to the server, or else it returns a LDAP error
code if not (see ldap_error(3LDAP)). If successful, ldap_add_ext() places the message id
of *msgidp*. A subsequent call to ldap_result(), can be used to obtain the result of the add
request.

The ldap_add_ext_s() function initiates a synchronous add operation and returns the result
of the operation itself.

Errors ldap_add() returns −1 in case of error initiating the request, and will set the *ld_errno* field in
the *ld* parameter to indicate the error. ldap_add_s() will return an LDAP error code directly.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also   ldap(3LDAP), ldap_error(3LDAP), ldap_modify(3LDAP), attributes(5)

Name ldap_ber_free – free a BerElement structure from memory

Synopsis cc *flag... file... -lldap [library ...]*
#include <ldap.h>

void ldap_ber_free(BerElement *ber, int *freebuf*);

Description You can make a call to the ldap_ber_free() function to free BerElement structures allocated by ldap_first_attribute() and by ldap_next_attribute() function calls. When freeing structures allocated by these functions, specify 0 for the *freebuf* argument. The ldap_first_attribute() and by ldap_next_attribute() functions do not allocate the extra buffer in the BerElement structure.

For example, to retrieve attributes from a search result entry, you need to call the ldap_first_attribute() function. A call to this function allocates a BerElement structure, which is used to help track the current attribute. When you are done working with the attributes, this structure should be freed from memory, if it still exists.

This function is deprecated . Use the ber_free() function instead.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ber_free(3LDAP), ldap_first_attribute(3LDAP), ldap_next_attribute(3LDAP), attributes(5)

Name ldap_bind, ldap_bind_s, ldap_sasl_bind, ldap_sasl_bind_s, ldap_simple_bind, ldap_simple_bind_s, ldap_unbind, ldap_unbind_s, ldap_unbind_ext, ldap_set_rebind_proc, ldap_sasl_interactive_bind_s – LDAP bind functions

Synopsis cc [*flag*...] *file*... -lldap [*library*...]
```
#include <lber.h>
#include <ldap.h>
```

```
int ldap_bind(LDAP *ld, char *who, char *cred, int method);
```

```
int ldap_bind_s(LDAP *ld, char *who, char *cred, int method);
```

```
int ldap_simple_bind(LDAP *ld, char *who, char *passwd);
```

```
int ldap_simple_bind_s(LDAP *ld, char *who, char *passwd);
```

```
int ldap_unbind(LDAP *ld);
```

```
int ldap_unbind_s(LDAP *ld);
```

```
int ldap_unbind_ext(LDAP *ld, LDAPControl **serverctrls,
    LDAPControl **clientctrls);
```

```
void ldap_set_rebind_proc(LDAP *ld, int (*rebindproc));
```

```
int ldap_sasl_bind(LDAP *ld, char *dn, char *mechanism,
    struct berval **serverctrls, LDAPControl **clientctrls,
    int *msgidp);
```

```
int ldap_sasl_bind_s(LDAP *ld, char *dn, char *mechanism,
    struct berval *cred, LDAPControl **serverctrls,
    LDAPControl **clientctrls);
```

```
int ldap_sasl_interactive_bind_s(LDAP *ld, char *dn,
    char *saslMechanism, LDAPControl **sctrl,
    LDAPControl **cctrl, unsigned flags,
    LDAP_SASL_INTERACT_PROC *callback, void *defaults);
```

Description These functions provide various interfaces to the LDAP bind operation. After a connection is made to an LDAP server, the ldap_bind() function returns the message ID of the request initiated. The ldap_bind_s() function returns an LDAP error code.

Simple Authentication The simplest form of the bind call is ldap_simple_bind_s(). The function takes the DN (Distinguished Name) of the *dn* parameter and the userPassword associated with the entry in *passwd* to return an LDAP error code. See ldap_error(3LDAP).

The ldap_simple_bind() call is asynchronous. The function takes the same parameters as ldap_simple_bind_s() but initiates the bind operation and returns the message ID of the request sent. The result of the operation can be obtained by a subsequent call to ldap_result(3LDAP).

General Authentication   The ldap_bind() and ldap_bind_s() functions are used to select the authentication method at runtime. Both functions take an extra *method* parameter to set the authentication method. For simple authentication, the *method* parameter is set to LDAP_AUTH_SIMPLE. The ldap_bind() function returns the message id of the request initiated. The ldap_bind_s() function returns an LDAP error code.

SASL Authentication   The ldap_sasl_bind() and ldap_sasl_bind_s() functions are used for general and extensible authentication over LDAP through the use of the Simple Authentication Security Layer. The routines both take the DN to bind as the authentication method. A dotted-string representation of an OID identifies the method, and the berval structure holds the credentials. The special constant value LDAP_SASL_SIMPLE (" ") can be passed to request simple authentication. Otherwise, the ldap_simple_bind() function or the ldap_simple_bind_s() function can be used.

The ldap_sasl_interactive_bind_s() helper function takes its data and performs the necessary ldap_sasl_bind() and associated SASL library authentication sequencing with the LDAP server that uses the provided connection (*ld*).

Upon a successful bind, the ldap_sasl_bind() function will, if negotiated by the SASL interface, install the necessary internal libldap plumbing to enable SASL integrity and privacy (over the wire encryption) with the LDAP server.

The LDAP_SASL_INTERACTIVE option flag is passed to the libldap API through the flags argument of the API. The flag tells the API to use the SASL interactive mode and to have the API request SASL authentication data through the LDAP_SASL_INTERACTIVE_PROC callback as needed. The callback provided is in the form:

```
typedef int (LDAP_SASL_INTERACT_PROC)
    (LDAP *ld, unsigned flags, void* defaults, void *interact);
```

The user-provided SASL callback is passed to the current LDAP connection pointer, the current flags field, an optional pointer to user-defined data, and the list of sasl_interact_t authentication values requested by libsasl(3LIB) to complete authentication.

The user-defined callback collects and returns the authentication information in the sasl_interact_t array according to libsasl rules. The authentication information can include user IDs, passwords, realms, or other information defined by SASL. The SASL library uses this date during sequencing to complete authentication.

Unbinding   The ldap_unbind() call is used to unbind from a directory, to terminate the current association, and to free the resources contained in the *ld* structure. Once the function is called, the connection to the LDAP server is closed and the *ld* structure is invalid. The ldap_unbind_s() and ldap_unbind() calls are identical and synchronous in nature.

The ldap_unbind_ext() function is used to unbind from a directory, to terminate the current association, and to free the resources contained in the LDAP structure. Unlike ldap_unbind() and ldap_unbind_s(), both server and client controls can be explicitly included with

ldap_unbind_ext() requests. No server response is made to an unbind request and responses should not be expected from server controls included with unbind requests.

Rebinding While Following Referral The ldap_set_rebind_proc() call is used to set a function called back to obtain bind credentials. The credentials are used when a new server is contacted after an LDAP referral. If ldap_set_rebind_proc() is never called, or if it is called with a NULL *rebindproc* parameter, an unauthenticated simple LDAP bind is always done when chasing referrals.

The rebindproc() function is declared as shown below:

```
int rebindproc(LDAP *ld, char **whop, char **credp,
    int *methodp, int freeit);
```

The LDAP library first calls the rebindproc() to obtain the referral bind credentials. The *freeit* parameter is zero. The *whop*, *credp*, and *methodp* parameters should be set as appropriate. If rebindproc() returns LDAP_SUCCESS, referral processing continues. The rebindproc() is called a second time with a non-zero *freeit* value to give the application a chance to free any memory allocated in the previous call.

If anything but LDAP_SUCCESS is returned by the first call to rebindproc(), referral processing is stopped and the error code is returned for the original LDAP operation.

Return Values Make a call to ldap_result(3LDAP) to obtain the result of a bind operation.

Errors Asynchronous functions will return −1 in case of error. See ldap_error(3LDAP) for more information on error codes returned. If no credentials are returned, the result parameter is set to NULL.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also ldap(3LDAP), ldap_error(3LDAP), ldap_open(3LDAP), ldap_result(3LDAP), libsasl(3LIB), attributes(5)

Name ldap_charset, ldap_set_string_translators, ldap_t61_to_8859, ldap_8859_to_t61, ldap_translate_from_t61, ldap_translate_to_t61, ldap_enable_translation – LDAP character set translation functions

Synopsis cc[*flag...*] *file...* -lldap[*library...*]
#include <lber.h>
#include <ldap.h>

void ldap_set_string_translators(LDAP *ld*,
    BERTranslateProc *encode_proc*, BERTranslateProc *decodeproc*);

typedef int(*BERTranslateProc)(char **bufp*, unsigned long *buflenp*,
    int *free_input*);

int ldap_t61_to_8859(char **bufp*, unsigned long *buflenp*,
    int *free_input*);

int ldap_8859_to_t61(char **bufp*, unsigned long *buflenp*,
    int *free_input*);

int ldap_translate_from_t61(LDAP *ld*, char **bufp*,
    unsigned long **lenp*, int *free_input*);

int ldap_translate_to_t61(LDAP *ld*, char **bufp*, unsigned long **lenp*,
    int *free_input*);

void ldap_enable_translation(LDAP *ld*, LDAPMessage *entry*, int *enable*);

Description These functions are used to used to enable translation of character strings used in the LDAP library to and from the T.61 character set used in the LDAP protocol. These functions are only available if the LDAP and LBER libraries are compiled with STR_TRANSLATION defined. It is also possible to turn on character translation by default so that all LDAP library callers will experience translation; see the LDAP Make-common source file for details.

ldap_set_string_translators() sets the translation functions that will be used by the LDAP library. They are not actually used until the *ld_lberoptions* field of the LDAP structure is set to include the LBER_TRANSLATE_STRINGS option.

ldap_t61_to_8859() and ldap_8859_to_t61() are translation functions for converting between T.61 characters and ISO-8859 characters. The specific 8859 character set used is determined at compile time.

ldap_translate_from_t61() is used to translate a string of characters from the T.61 character set to a different character set. The actual translation is done using the *decode_proc* that was passed to a previous call to ldap_set_string_translators(). On entry, *bufp* should point to the start of the T.61 characters to be translated and *lenp* should contain the number of bytes to translate. If *free_input* is non-zero, the input buffer will be freed if translation is a success. If the translation is a success, LDAP_SUCCESS will be returned, *bufp* will point to a newly malloc'd buffer that contains the translated characters, and *lenp* will contain the length of the result. If translation fails, an LDAP error code will be returned.

ldap_translate_to_t61() is used to translate a string of characters to the T.61 character set from a different character set. The actual translation is done using the *encode_proc* that was passed to a previous call to ldap_set_string_translators(). This function is called just like ldap_translate_from_t61().

ldap_enable_translation() is used to turn on or off string translation for the LDAP entry *entry* (typically obtained by calling ldap_first_entry() or ldap_next_entry() after a successful LDAP search operation). If enable is zero, translation is disabled; if non-zero, translation is enabled. This function is useful if you need to ensure that a particular attribute is not translated when it is extracted using ldap_get_values() or ldap_get_values_len(). For example, you would not want to translate a binary attributes such as jpegPhoto.

Attributes   See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also   ldap(3LDAP), attributes(5)

Name ldap_compare, ldap_compare_s, ldap_compare_ext, ldap_compare_ext_s – LDAP compare operation

Synopsis cc[*flag*...] *file*... -lldap[*library*...]
```
#include <lber.h>
#include <ldap.h>
```

int ldap_compare(LDAP *ld*, char *dn*, char *attr*, char *value*);

int ldap_compare_s(LDAP *ld*, char *dn*, char *attr*, char *value*);

int ldap_compare_ext(LDAP *ld*, char *dn*, char *attr*,
    struct berval *bvalue*, LDAPControl **serverctrls*,
    LDAPControl **clientctrls*,int *msgidp*);

int ldap_compare_ext_s(LDAP *ld*, char *dn*, char *attr*,
    struct berval *bvalue*, LDAPControl **serverctrls*,
    LDAPControl **clientctrls*);

Description The ldap_compare_s() function is used to perform an LDAP compare operation synchronously. It takes *dn*, the DN of the entry upon which to perform the compare, and *attr* and *value*, the attribute type and value to compare to those found in the entry. It returns an LDAP error code, which will be LDAP_COMPARE_TRUE if the entry contains the attribute value and LDAP_COMPARE_FALSE if it does not. Otherwise, some error code is returned.

The ldap_compare() function is used to perform an LDAP compare operation asynchronously. It takes the same parameters as ldap_compare_s(), but returns the message id of the request it initiated. The result of the compare can be obtained by a subsequent call to ldap_result(3LDAP).

The ldap_compare_ext() function initiates an asynchronous compare operation and returns LDAP_SUCCESS if the request was successfully sent to the server, or else it returns a LDAP error code if not (see ldap_error(3LDAP). If successful, ldap_compare_ext() places the message id of the request in *msgidp*. A subsequent call to ldap_result(), can be used to obtain the result of the add request.

The ldap_compare_ext_s() function initiates a synchronous compare operation and as such returns the result of the operation itself.

Errors ldap_compare_s() returns an LDAP error code which can be interpreted by calling one of ldap_perror(3LDAP) and friends. ldap_compare() returns −1 if something went wrong initiating the request. It returns the non-negative message id of the request if it was successful.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also ldap(3LDAP), ldap_error(3LDAP), attributes(5)

Bugs There is no way to compare binary values using ldap_compare().

Name ldap_control_free, ldap_controls_free – LDAP control disposal

Synopsis cc[*flag*...] *file*... -lldap[*library*...]
#include <lber.h>
#include <ldap.h>

void ldap_control_free(LDAPControl *ctrl*);

void ldap_controls_free(LDAPControl *ctrls*);

Description ldap_controls_free() and ldap_control_free() are routines which can be used to dispose of a single control or an array of controls allocated by other LDAP APIs.

Return Values None.

Errors No errors are defined for these functions.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap_error(3LDAP), ldap_result(3LDAP), attributes(5)

Name ldap_delete, ldap_delete_s, ldap_delete_ext, ldap_delete_ext_s – LDAP delete operation

Synopsis cc[*flag*...] *file*... -lldap[*library*...]
#include <lber.h>
#include <ldap.h>

int ldap_delete(LDAP *ld*, char *dn*);

int ldap_delete_s(LDAP *ld*, char *dn*);

int ldap_delete_ext(LDAP *ld*, char *dn*, LDAPControl **serverctrls*,
    LDAPControl **clientctrls*, int *msgidp*);

int ldap_delete_ext_s(LDAP *ld*, char *dn*, LDAPControl **serverctrls*,
    LDAPControl **clientctrls*);

Description The ldap_delete_s() function is used to perform an LDAP delete operation synchronously.
It takes *dn*, the DN of the entry to be deleted. It returns an LDAP error code, indicating the
success or failure of the operation.

The ldap_delete() function is used to perform an LDAP delete operation asynchronously. It
takes the same parameters as ldap_delete_s(), but returns the message id of the request it
initiated. The result of the delete can be obtained by a subsequent call to
ldap_result(3LDAP).

The ldap_delete_ext() function initiates an asynchronous delete operation and returns
LDAP_SUCCESS if the request was successfully sent to the server, or else it returns a LDAP error
code if not (see ldap_error(3LDAP)). If successful, ldap_delete_ext() places the message
id of the request in *msgidp*. A subsequent call to ldap_result(), can be used to obtain the
result of the add request.

The ldap_delete_ext_s() function initiates a synchronous delete operation and as such
returns the result of the operation itself.

Errors ldap_delete_s() returns an LDAP error code which can be interpreted by calling one of
ldap_perror(3LDAP) functions. ldap_delete() returns −1 if something went wrong
initiating the request. It returns the non-negative message id of the request if things were
successful.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also   ldap(3LDAP), ldap_error(3LDAP), attributes(5)

Name ldap_disptmpl, ldap_init_templates, ldap_init_templates_buf, ldap_free_templates, ldap_first_disptmpl, ldap_next_disptmpl, ldap_oc2template, ldap_name2template, ldap_tmplattrs, ldap_first_tmplrow, ldap_next_tmplrow, ldap_first_tmplcol, ldap_next_tmplcol – LDAP display template functions

Synopsis cc [*flag...*] *file...* -lldap [*library...*]
```
#include <lber.h>
#include <ldap.h>

int ldap_init_templates(char *file, struct ldap_disptmpl **tmpllistp);

int ldap_init_templates_buf(char *buf, unsigned long len,
    struct ldap_disptmpl **tmpllistp);

void ldap_free_templates(struct ldap_disptmpl *tmpllist);

struct ldap_disptmpl *ldap_first_disptmpl
    (struct ldap_disptmpl *tmpllist);

struct ldap_disptmpl *ldap_next_disptmpl
    (struct ldap_disptmpl *tmpllist,struct ldap_disptmpl *tmpl);

struct ldap_disptmpl *ldap_oc2template (char **oclist,
    struct ldap_disptmpl *tmpllist);

struct ldap_disptmpl *ldap_name2template (char *name,
    struct ldap_disptmpl *tmpllist);

char **ldap_tmplattrs(struct ldap_disptmpl *tmpl, char **includeattrs,
    int exclude, unsigned long syntaxmask);

struct ldap_tmplitem *ldap_first_tmplrow(struct ldap_disptmpl *tmpl);

struct ldap_tmplitem *ldap_next_tmplrow(struct ldap_disptmpl *tmpl,
    struct ldap_tmplitem *row);

struct ldap_tmplitem *ldap_first_tmplcol(struct ldap_disptmpl *tmpl,
    struct ldap_tmplitem *row, struct ldap_tmplitem *col);

struct ldap_tmplitem *ldap_next_tmplcol(struct ldap_disptmpl *tmpl,
    struct ldap_tmplitem *row, struct ldap_tmplitem *col);
```

Description These functions provide a standard way to access LDAP entry display templates. Entry display templates provide a standard way for LDAP applications to display directory entries. The general idea is that it is possible to map the list of object class values present in an entry to an appropriate display template. Display templates are defined in a configuration file. See ldaptemplates.conf(4). Each display template contains a pre-determined list of items, where each item generally corresponds to an attribute to be displayed. The items contain information and flags that the caller can use to display the attribute and values in a reasonable fashion. Each item has a syntaxid, which are described in the SYNTAX IDS section below. The ldap_entry2text(3LDAP) functions use the display template functions and produce text output.

ldap_init_templates() reads a sequence of templates from a valid LDAP template configuration file (see ldaptemplates.conf(4)). Upon success, 0 is returned, and *tmpllistp* is set to point to a list of templates. Each member of the list is an ldap_disptmpl structure (defined below in the DISPTMPL Structure Elements section).

ldap_init_templates_buf() reads a sequence of templates from *buf* (whose size is *buflen)*. *buf* should point to the data in the format defined for an LDAP template configuration file (see ldaptemplates.conf(4)). Upon success, 0 is returned, and *tmpllistp* is set to point to a list of templates.

The LDAP_SET_DISPTMPL_APPDATA() macro is used to set the value of the dt_appdata field in an ldap_disptmpl structure. This field is reserved for the calling application to use; it is not used internally.

The LDAP_GET_DISPTMPL_APPDATA() macro is used to retrieve the value in the dt_appdata field.

The LDAP_IS_DISPTMPL_OPTION_SET() macro is used to test a ldap_disptmpl structure for the existence of a template option. The options currently defined are: LDAP_DTMPL_OPT_ADDABLE (it is appropriate to allow entries of this type to be added), LDAP_DTMPL_OPT_ALLOWMODRDN (it is appropriate to offer the "modify rdn"operation), LDAP_DTMPL_OPT_ALTVIEW (this template is merely an alternate view of another template, typically used for templates pointed to be an LDAP_SYN_LINKACTION item).

ldap_free_templates() disposes of the templates allocated by ldap_init_templates().

ldap_first_disptmpl() returns the first template in the list *tmpllist.* The *tmpllist* is typically obtained by calling ldap_init_templates() .

ldap_next_disptmpl() returns the template after *tmpl* in the template list *tmpllist. A* NULL pointer is returned if *tmpl* is the last template in the list.

ldap_oc2template() searches *tmpllist* for the best template to use to display an entry that has a specific set of objectClass values. *oclist* should be a null-terminated array of strings that contains the values of the objectClass attribute of the entry. A pointer to the first template where all of the object classes listed in one of the template's dt_oclist elements are contained in *oclist* is returned. A NULL pointer is returned if no appropriate template is found.

ldap_tmplattrs() returns a null-terminated array that contains the names of attributes that need to be retrieved if the template *tmpl* is to be used to display an entry. The attribute list should be freed using ldap_value_free( ). The *includeattrs* parameter contains a null-terminated array of attributes that should always be included (it may be NULL if no extra attributes are required). If *syntaxmask* is non-zero, it is used to restrict the attribute set returned. If *exclude* is zero, only attributes where the logical AND of the template item syntax id and the *syntaxmask* is non-zero are included. If *exclude* is non-zero, attributes where the logical AND of the template item syntax id and the *syntaxmask* is non-zero are excluded.

ldap_first_tmplrow() returns a pointer to the first row of items in template *tmpl.*

ldap_next_tmplrow() returns a pointer to the row that follows *row* in template *tmpl*.

ldap_first_tmplcol() returns a pointer to the first item (in the first column) of row *row* within template *tmpl*. A pointer to an ldap_tmplitem structure (defined below in the TMPLITEM Structure Elements section) is returned.

The LDAP_SET_TMPLITEM_APPDATA() macro is used to set the value of the ti_appdata field in a ldap_tmplitem structure. This field is reserved for the calling application to use; it is not used internally.

The LDAP_GET_TMPLITEM_APPDATA() macro is used to retrieve the value of the ti_appdata field.

The LDAP_IS_TMPLITEM_OPTION_SET() macro is used to test a ldap_tmplitem structure for the existence of an item option. The options currently defined are: LDAP_DITEM_OPT_READONLY (this attribute should not be modified), LDAP_DITEM_OPT_SORTVALUES (it makes sense to sort the values), LDAP_DITEM_OPT_SINGLEVALUED (this attribute can only hold a single value), LDAP_DITEM_OPT_VALUEREQUIRED (this attribute must contain at least one value), LDAP_DITEM_OPT_HIDEIFEMPTY (do not show this item if there are no values), and LDAP_DITEM_OPT_HIDEIFFALSE (for boolean attributes only: hide this item if the value is FALSE).

ldap_next_tmplcol() returns a pointer to the item (column) that follows column col within row *row* of template *tmpl*.

DISPTMPL Structure Elements

The ldap_disptmpl structure is defined as:

```
struct ldap_disptmpl {
char                    *dt_name;
char                    *dt_pluralname;
char                    *dt_iconname;
unsigned long           dt_options;
char                    *dt_authattrname;
char                    *dt_defrdnattrname;
char                    *dt_defaddlocation;
struct ldap_oclist      *dt_oclist;
struct ldap_adddeflist  *dt_adddeflist;
struct ldap_tmplitem    *dt_items;
void                    *dt_appdata;
struct ldap_disptmpl    *dt_next;
};
```

The dt_name member is the singular name of the template. The dt_pluralname is the plural name. The dt_iconname member will contain the name of an icon or other graphical element that can be used to depict entries that correspond to this display template. The dt_options contains options which may be tested using the LDAP_IS_TMPLITEM_OPTION_SET() macro.

The dt_authattrname contains the name of the DN-syntax attribute whose value(s) should be used to authenticate to make changes to an entry. If dt_authattrname is NULL, then authenticating as the entry itself is appropriate. The dt_defrdnattrname is the name of the attribute that is normally used to name entries of this type, for example, "cn" for person entries. The dt_defaddlocation is the distinguished name of an entry below which new entries of this type are typically created (its value is site-dependent).

dt_oclist is a pointer to a linked list of object class arrays, defined as:

```
struct ldap_oclist {
char               **oc_objclasses;
struct ldap_oclist *oc_next;
};
```

These are used by the ldap_oc2template() function.

dt_adddeflist is a pointer to a linked list of rules for defaulting the values of attributes when new entries are created. The ldap_adddeflist structure is defined as:

```
struct ldap_adddeflist {
int                 ad_source;
char                *ad_attrname;
char                *ad_value;
struct ldap_adddeflist *ad_next;
};
```

The ad_attrname member contains the name of the attribute whose value this rule sets. If ad_source is LDAP_ADSRC_CONSTANTVALUE then the ad_value member contains the (constant) value to use. If ad_source is LDAP_ADSRC_ADDERSDN then ad_value is ignored and the distinguished name of the person who is adding the new entry is used as the default value for ad_attrname.

TMPLITEM Structure Elements   The ldap_tmplitem structure is defined as:

```
struct ldap_tmplitem {
unsigned long       ti_syntaxid;
unsigned long       ti_options;
char                *ti_attrname;
char                *ti_label;
char                **ti_args;
struct ldap_tmplitem *ti_next_in_row;
struct ldap_tmplitem *ti_next_in_col;
void                *ti_appdata;
};
```

Syntax IDs   Syntax ids are found in the ldap_tmplitem structure element ti_syntaxid, and they can be used to determine how to display the values for the attribute associated with an item. The LDAP_GET_SYN_TYPE() macro can be used to return a general type from a syntax id. The five general types currently defined are: LDAP_SYN_TYPE_TEXT (for attributes that are most

appropriately shown as text), `LDAP_SYN_TYPE_IMAGE` (for JPEG or FAX format images), `LDAP_SYN_TYPE_BOOLEAN` (for boolean attributes), `LDAP_SYN_TYPE_BUTTON` (for attributes whose values are to be retrieved and display only upon request, for example, in response to the press of a button, a JPEG image is retrieved, decoded, and displayed), and `LDAP_SYN_TYPE_ACTION` (for special purpose actions such as "search for the entries where this entry is listed in the seeAlso attribute").

The `LDAP_GET_SYN_OPTIONS` macro can be used to retrieve an unsigned long bitmap that defines options. The only currently defined option is `LDAP_SYN_OPT_DEFER,` which (if set) implies that the values for the attribute should not be retrieved until requested.

There are sixteen distinct syntax ids currently defined. These generally correspond to one or more X.500 syntaxes.

`LDAP_SYN_CASEIGNORESTR` is used for text attributes which are simple strings whose case is ignored for comparison purposes.

`LDAP_SYN_MULTILINESTR` is used for text attributes which consist of multiple lines, for example, `postalAddress`, `homePostalAddress`, `multilineDescription`, or any attributes of syntax `caseIgnoreList`.

`LDAP_SYN_RFC822ADDR` is used for case ignore string attributes that are RFC-822 conformant mail addresses, for example, mail.

`LDAP_SYN_DN` is used for attributes with a Distinguished Name syntax, for example, `seeAlso`.

`LDAP_SYN_BOOLEAN` is used for attributes with a boolean syntax.

`LDAP_SYN_JPEGIMAGE` is used for attributes with a jpeg syntax, for example, jpegPhoto.

`LDAP_SYN_JPEGBUTTON` is used to provide a button (or equivalent interface element) that can be used to retrieve, decode, and display an attribute of jpeg syntax.

`LDAP_SYN_FAXIMAGE` is used for attributes with a photo syntax, for example, Photo. These are actually Group 3 Fax (T.4) format images.

`LDAP_SYN_FAXBUTTON` is used to provide a button (or equivalent interface element) that can be used to retrieve, decode, and display an attribute of photo syntax.

`LDAP_SYN_AUDIOBUTTON` is used to provide a button (or equivalent interface element) that can be used to retrieve and play an attribute of audio syntax. Audio values are in the "mu law" format, also known as "au" format.

`LDAP_SYN_TIME` is used for attributes with the UTCTime syntax, for example, `lastModifiedTime`. The value(s) should be displayed in complete date and time fashion.

`LDAP_SYN_DATE` is used for attributes with the UTCTime syntax, for example, `lastModifiedTime`. Only the date portion of the value(s) should be displayed.

LDAP_SYN_LABELEDURL is used for labeledURL attributes.

LDAP_SYN_SEARCHACTION is used to define a search that is used to retrieve related information. If ti_attrname is not NULL, it is assumed to be a boolean attribute which will cause no search to be performed if its value is FALSE. The ti_args structure member will have four strings in it: ti_args[0] should be the name of an attribute whose values are used to help construct a search filter or "-dn" is the distinguished name of the entry being displayed should be used, ti_args[1] should be a filter pattern where any occurrences of "%v" are replaced with the value derived from ti_args[0], ti_args[2] should be the name of an additional attribute to retrieve when performing the search, and ti_args[3] should be a human-consumable name for that attribute. The ti_args[2] attribute is typically displayed along with a list of distinguished names when multiple entries are returned by the search.

LDAP_SYN_LINKACTION is used to define a link to another template by name. ti_args[0] will contain the name of the display template to use. The ldap_name2template() function can be used to obtain a pointer to the correct ldap_disptmpl structure.

LDAP_SYN_ADDDNACTION and LDAP_SYN_VERIFYDNACTION are reserved as actions but currently undefined.

Errors The init template functions return LDAP_TMPL_ERR_VERSION if *buf* points to data that is newer than can be handled, LDAP_TMPL_ERR_MEM if there is a memory allocation problem, LDAP_TMPL_ERR_SYNTAX if there is a problem with the format of the templates buffer or file. LDAP_TMPL_ERR_FILE is returned by ldap_init_templates if the file cannot be read. Other functions generally return NULL upon error.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap(3LDAP), ldap_entry2text(3LDAP), ldaptemplates.conf(4), attributes(5)

Name ldap_entry2text, ldap_entry2text_search, ldap_entry2html, ldap_entry2html_search, ldap_vals2html, ldap_vals2text – LDAP entry display functions

Synopsis cc[*flag...*] *file...* -lldap[*library...*]
#include <lber.h>
#include <ldap.h>

```
int ldap_entry2text(LDAP *ld, char *buf, LDAPMessage *entry,
     struct ldap_disptmpl *tmpl, char **defattrs, char ***defvals,
     int (*writeproc)(), void *writeparm, char *eol, int rdncount,
     unsigned long opts);

int ldap_entry2text_search(LDAP *ld, char *dn, char *base,
     LDAPMessage *entry, struct ldap_disptmpl *tmpllist,
     char **defattrs, char ***defvals, int (*writeproc)(),
     void *writeparm, char *eol,int rdncount,
     unsigned long opts);

int ldap_vals2text(LDAP *ld, char *buf, char **vals, char *label,
     int labelwidth, unsigned longsyntaxid, int (*writeproc)(),
     void *writeparm, char *eol, int rdncount);

int ldap_entry2html(LDAP *ld, char *buf, LDAPMessage *entry,
     struct ldap_disptmpl *tmpl, char **defattrs, char ***defvals,
     int (*writeproc)(),void *writeparm, char *eol, int rdncount,
     unsigned long opts, char *urlprefix, char *base);

int ldap_entry2html_search(LDAP *ld, char *dn, LDAPMessage *entry,
     struct ldap_disptmpl *tmpllist, char **defattrs, char ***defvals,
     int (*writeproc)(), void *writeparm, char *eol, int rdncount,
     unsigned long opts, char *urlprefix);

int ldap_vals2html(LDAP *ld, char *buf, char **vals,
     char *label, int labelwidth, unsigned long syntaxid,
     int (*writeproc)(), void *writeparm, char *eol, int rdncount,
     char *urlprefix);
```

#define LDAP_DISP_OPT_AUTOLABELWIDTH 0x00000001

#define LDAP_DISP_OPT_HTMLBODYONLY   0x00000002

#define LDAP_DTMPL_BUFSIZ 2048

Description These functions use the LDAP display template functions (see ldap_disptmpl(3LDAP) and ldap_templates.conf(4)) to produce a plain text or an HyperText Markup Language (HTML) display of an entry or a set of values. Typical plain text output produced for an entry might look like:

```
"Barbara J Jensen, Information Technology Division"
Also Known As:
Babs Jensen
Barbara Jensen
Barbara J Jensen
```

```
          E-Mail Address:
          bjensen@terminator.rs.itd.umich.edu
          Work Address:
          535 W. William
          Ann Arbor, MI 48103
          Title:
          Mythical Manager, Research Systems
          ...
```

The exact output produced will depend on the display template configuration. HTML output is similar to the plain text output, but more richly formatted.

`ldap_entry2text()` produces a text representation of *entry* and writes the text by calling the *writeproc* function. All of the attributes values to be displayed must be present in *entry*; no interaction with the LDAP server will be performed within ldap_entry2text. ld is the LDAP pointer obtained by a previous call to ldap_open. *writeproc* should be declared as:

```
int writeproc( writeparm, p, len )
 void  *writeparm;
 char  *p;
 int   len;
```

where *p* is a pointer to text to be written and *len* is the length of the text. *p* is guaranteed to be zero-terminated. Lines of text are terminated with the string *eol*. *buf* is a pointer to a buffer of size `LDAP_DTMPL_BUFSIZ` or larger. If *buf is* NULL then a buffer is allocated and freed internally. *tmpl* is a pointer to the display template to be used (usually obtained by calling `ldap_oc2template`). If *tmpl* is NULL, no template is used and a generic display is produced. *defattrs* is a NULL-terminated array of LDAP attribute names which you wish to provide default values for (only used if *entry* contains no values for the attribute). An array of NULL-terminated arrays of default values corresponding to the attributes should be passed in *defvals. The rdncount* parameter is used to limit the number of Distinguished Name (DN) components that are actually displayed for DN attributes. If *rdncount* is zero, all components are shown. *opts* is used to specify output options. The only values currently allowed are zero (default output), `LDAP_DISP_OPT_AUTOLABELWIDTH` which causes the width for labels to be determined based on the longest label in *tmpl, and* `LDAP_DISP_OPT_HTMLBODYONLY`. The `LDAP_DISP_OPT_HTMLBODYONLY` option instructs the library not to include <HTML>, <HEAD>, <TITLE>, and <BODY> tags. In other words, an HTML fragment is generated, and the caller is responsible for prepending and appending the appropriate HTML tags to construct a correct HTML document.

`ldap_entry2text_search()` is similar to ldap_entry2text, and all of the like-named parameters have the same meaning except as noted below. If *base* is not NULL, it is the search base to use when executing search actions. If it is NULL, search action template items are ignored. If *entry* is not NULL, it should contain the *objectClass* attribute values for the entry to be displayed. If *entry* is NULL, *dn* must not be NULL, and ldap_entry2text_search will retrieve the objectClass values itself by calling ldap_search_s. ldap_entry2text_search will determine the appropriate display template to use by calling ldap_oc2template, and will call

ldap_search_s to retrieve any attribute values to be displayed. The *tmpllist* parameter is a pointer to the entire list of templates available (usually obtained by calling ldap_init_templates or ldap_init_templates_buf). If *tmpllist* is NULL, ldap_entry2text_search will attempt to read a load templates from the default template configuration file ETCDIR/ldaptemplates.conf

ldap_vals2text produces a text representation of a single set of LDAP attribute values. The *ld, buf, writeproc, writeparm, eol,* and *rdncount* parameters are the same as the like-named parameters for ldap_entry2text. *vals* is a NULL-terminated list of values, usually obtained by a call to ldap_get_values. *label* is a string shown next to the values (usually a friendly form of an LDAP attribute name). *labelwidth* specifies the label margin, which is the number of blank spaces displayed to the left of the values. If zero is passed, a default label width is used. *syntaxid* is a display template attribute syntax identifier (see ldap_disptmpl(3LDAP) for a list of the pre-defined LDAP_SYN_... values).

ldap_entry2html produces an HTML representation of *entry*. It behaves exactly like ldap_entry2text(3LDAP), except for the formatted output and the addition of two parameters. *urlprefix* is the starting text to use when constructing an LDAP URL. The default is the string *ldap:///* The second additional parameter, *base,* the search base to use when executing search actions. If it is NULL, search action template items are ignored.

ldap_entry2html_search behaves exactly like ldap_entry2text_search(3LDAP), except HTML output is produced and one additional parameter is required. *urlprefix* is the starting text to use when constructing an LDAP URL. The default is the string *ldap:///*

ldap_vals2html behaves exactly like ldap_vals2text, except HTML output is and one additional parameter is required. *urlprefix* is the starting text to use when constructing an LDAP URL. The default is the string *ldap:///*

Errors These functions all return an LDAP error code. LDAP_SUCCESS is returned if no error occurs. See ldap_error(3LDAP) for details. The *ld_errno* field of the *ld* parameter is also set to indicate the error.

Files ETCDIR/ldaptemplates.conf

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap(3LDAP), ldap_disptmpl(3LDAP), ldaptemplates.conf(4), attributes(5)

Name ldap_error, ldap_err2string, ldap_perror, ldap_result2error – LDAP protocol error handling functions

Synopsis cc[*flag*...] *file*... -lldap[*library*...]
#include <lber.h>
#include <ldap.h>

char *ldap_err2string(int *err*);

void ldap_perror(LDAP *ld*, const char *s*);

int ldap_result2error(LDAP *ld*, LDAPMessage *res*, int *freeit*);

Description These functions interpret the error codes that are returned by the LDAP API routines. The ldap_perror() and ldap_result2error() functions are deprecated for all new development. Use ldap_err2string() instead.

You can also use ldap_parse_sasl_bind_result(3LDAP), ldap_parse_extended_result(3LDAP), and ldap_parse_result(3LDAP) to provide error handling and interpret error codes returned by LDAP API functions.

The ldap_err2string() function takes *err*, a numeric LDAP error code, returned either by ldap_parse_result(3LDAP) or another LDAP API call. It returns an informative, null-terminated, character string that describes the error.

The ldap_result2error() function takes *res*, a result produced by ldap_result(3LDAP) or other synchronous LDAP calls, and returns the corresponding error code. If the *freeit* parameter is non-zero, it indicates that the *res* parameter should be freed by a call to ldap_result(3LDAP) after the error code has been extracted.

Similar to the way perror(3C) works, the ldap_perror() function can be called to print an indication of the error to standard error.

Errors The possible values for an LDAP error code are:

LDAP_SUCCESS	The request was successful.
LDAP_OPERATIONS_ERROR	An operations error occurred.
LDAP_PROTOCOL_ERROR	A protocol violation was detected.
LDAP_TIMELIMIT_EXCEEDED	An LDAP time limit was exceeded.
LDAP_SIZELIMIT_EXCEEDED	An LDAP size limit was exceeded.
LDAP_COMPARE_FALSE	A compare operation returned false.
LDAP_COMPARE_TRUE	A compare operation returned true.
LDAP_STRONG_AUTH_NOT_SUPPORTED	The LDAP server does not support strong authentication.
LDAP_STRONG_AUTH_REQUIRED	Strong authentication is required for the operation.

LDAP_PARTIAL_RESULTS	Only partial results are returned.
LDAP_NO_SUCH_ATTRIBUTE	The attribute type specified does not exist in the entry.
LDAP_UNDEFINED_TYPE	The attribute type specified is invalid.
LDAP_INAPPROPRIATE_MATCHING	The filter type is not supported for the specified attribute.
LDAP_CONSTRAINT_VIOLATION	An attribute value specified violates some constraint. For example, a postalAddress has too many lines, or a line that is too long.
LDAP_TYPE_OR_VALUE_EXISTS	An attribute type or attribute value specified already exists in the entry.
LDAP_INVALID_SYNTAX	An invalid attribute value was specified.
LDAP_NO_SUCH_OBJECT	The specified object does not exist in the directory.
LDAP_ALIAS_PROBLEM	An alias in the directory points to a nonexistent entry.
LDAP_INVALID_DN_SYNTAX	A syntactically invalid DN was specified.
LDAP_IS_LEAF	The object specified is a leaf.
LDAP_ALIAS_DEREF_PROBLEM	A problem was encountered when dereferencing an alias.
LDAP_INAPPROPRIATE_AUTH	Inappropriate authentication was specified. For example, LDAP_AUTH_SIMPLE was specified and the entry does not have a userPassword attribute.
LDAP_INVALID_CREDENTIALS	Invalid credentials were presented, for example, the wrong password.
LDAP_INSUFFICIENT_ACCESS	The user has insufficient access to perform the operation.
LDAP_BUSY	The DSA is busy.
LDAP_UNAVAILABLE	The DSA is unavailable.
LDAP_UNWILLING_TO_PERFORM	The DSA is unwilling to perform the operation.
LDAP_LOOP_DETECT	A loop was detected.
LDAP_NAMING_VIOLATION	A naming violation occurred.
LDAP_OBJECT_CLASS_VIOLATION	An object class violation occurred. For example, a must attribute was missing from the entry.

LDAP_NOT_ALLOWED_ON_NONLEAF	The operation is not allowed on a nonleaf object.
LDAP_NOT_ALLOWED_ON_RDN	The operation is not allowed on an RDN.
LDAP_ALREADY_EXISTS	The entry already exists.
LDAP_NO_OBJECT_CLASS_MODS	Object class modifications are not allowed.
LDAP_OTHER	An unknown error occurred.
LDAP_SERVER_DOWN	The LDAP library cannot contact the LDAP server.
LDAP_LOCAL_ERROR	Some local error occurred. This is usually the result of a failed malloc(3C) call or a failure to fflush(3C) the stdio stream to files, even when the LDAP requests were processed successfully by the remote server.
LDAP_ENCODING_ERROR	An error was encountered encoding parameters to send to the LDAP server.
LDAP_DECODING_ERROR	An error was encountered decoding a result from the LDAP server.
LDAP_TIMEOUT	A time limit was exceeded while waiting for a result.
LDAP_AUTH_UNKNOWN	The authentication method specified to ldap_bind(3LDAP) is not known.
LDAP_FILTER_ERROR	An invalid filter was supplied to ldap_search(3LDAP), for example, unbalanced parentheses.
LDAP_PARAM_ERROR	An LDAP function was called with a bad parameter, for example, a NULL *ld* pointer, and the like.
LDAP_NO_MEMORY	A memory allocation call failed in an LDAP library function, for example, malloc(3C).
LDAP_CONNECT_ERROR	The LDAP client has either lost its connetion to an LDAP server or it cannot establish a connection.
LDAP_NOT_SUPPORTED	The requested functionality is not supported., for example, when an LDAPv2 client requests some LDAPv3 functionality.
LDAP_CONTROL_NOT_FOUND	An LDAP client requested a control not found in the list of supported controls sent by the server.
LDAP_NO_RESULTS_RETURNED	The LDAP server sent no results.
LDAP_MORE_RESULTS_TO_RETURN	More results are chained in the message chain.

LDAP_CLIENT_LOOP   A loop has been detected, for example, when following referrals.

LDAP_REFERRAL_LIMIT_EXCEEDED   The referral exceeds the hop limit. The hop limit determines the number of servers that the client can hop through to retrieve data.

Attributes   See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also   fflush(3C), ldap(3LDAP), ldap_bind(3LDAP), ldap_result(3LDAP), ldap_parse_extended_result(3LDAP), ldap_parse_result(3LDAP), ldap_parse_sasl_bind_result(3LDAP), ldap_search(3LDAP), malloc(3C), perror(3C) , attributes(5)

Name ldap_first_attribute, ldap_next_attribute – step through LDAP entry attributes

Synopsis cc [*flag*...] *file*... -lldap[*library*...]
#include <lber.h>
#include <ldap.h>

char *ldap_first_attribute(LDAP *_ld_, LDAPMessage *_entry_,
    BerElement **_berptr_);

char *ldap_next_attribute(LDAP *_ld_, LDAPMessage *_entry_,
    BerElement *_ber_);

Description The ldap_first_attribute() function gets the value of the first attribute in an entry.

The ldap_first_attribute() function returns the name of the first attribute in the entry. To get the value of the first attribute, pass the attribute name to the ldap_get_values() function or to the ldap_get_values_len() function.

The ldap_next_attribute() function gets the value of the next attribute in an entry.

After stepping through the attributes, the application should call ber_free() to free the BerElement structure allocated by the ldap_first_attribute() function if the structure is other than NULL.

Errors If an error occurs, NULL is returned and the ld_errno field in the *ld* parameter is set to indicate the error. See ldap_error(3LDAP) for a description of possible error codes.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap(3LDAP), ldap_first_entry(3LDAP), ldap_get_values(3LDAP), ldap_error(3LDAP), attributes(5)

Notes The ldap_first_attribute() function alllocates memory that might need to be freed by the caller by means of ber_free(3LDAP).

Name ldap_first_entry, ldap_next_entry, ldap_count_entries, ldap_count_references, ldap_first_reference, ldap_next_reference – LDAP entry parsing and counting functions

Synopsis cc [*flag*...] *file*... -lldap [*library*...]
```
#include <lber.h>
#include <ldap.h>
```

LDAPMessage *ldap_first_entry(LDAP*ld, LDAPMessage *result);

LDAPMessage *ldap_next_entry(LDAP *ld, LDAPMessage *entry);

int ldap_count_entries(LDAP *ld, LDAPMessage *result);

LDAPMessage *ldap_first_reference(LDAP *ld, LDAPMessage *res);

LDAPMessage *ldap_next_reference(LDAP *ld, LDAPMessage *res);

int ldap_count_references(LDAP *ld, LDAPMessage *res);

Description These functions are used to parse results received from ldap_result(3LDAP) or the synchronous LDAP search operation functions ldap_search_s(3LDAP) and ldap_search_st(3LDAP).

The ldap_first_entry() function is used to retrieve the first entry in a chain of search results. It takes the *result* as returned by a call to ldap_result(3LDAP) or ldap_search_s(3LDAP) or ldap_search_st(3LDAP) and returns a pointer to the first entry in the result.

This pointer should be supplied on a subsequent call to ldap_next_entry() to get the next entry, the result of which should be supplied to the next call to ldap_next_entry(), etc. ldap_next_entry() will return NULL when there are no more entries. The entries returned from these calls are used in calls to the functions described in ldap_get_dn(3LDAP), ldap_first_attribute(3LDAP), ldap_get_values(3LDAP), etc.

A count of the number of entries in the search result can be obtained by calling ldap_count_entries().

ldap_first_reference() and ldap_next_reference() are used to step through and retrieve the list of continuation references from a search result chain.

The ldap_count_references() function is used to count the number of references that are contained in and remain in a search result chain.

Errors If an error occurs in ldap_first_entry() or ldap_next_entry(), NULL is returned and the ld_errno field in the *ld* parameter is set to indicate the error. If an error occurs in ldap_count_entries(), −1 is returned, and ld_errno is set appropriately. See ldap_error(3LDAP) for a description of possible error codes.

Attributes   See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also   ldap(3LDAP), ldap_result(3LDAP), ldap_search(3LDAP),
ldap_first_attribute(3LDAP), ldap_get_values(3LDAP), ldap_get_dn(3LDAP),
attributes(5)

Name ldap_first_message, ldap_count_messages, ldap_next_message, ldap_msgtype – LDAP message processing functions

Synopsis cc[*flag...*] *file...* -lldap[*library...*]
```
#include <lber.h>
#include <ldap.h>
```

```
int ldap_count_messages(LDAP *ld, LDAPMessage *res);
```

```
LDAPMessage *ldap_first_message(LDAP *ld, LDAPMessage *res);
```

```
LDAPMessage *ldap_next_message(LDAP *ld, LDAPMessage *msg);
```

```
int ldap_msgtype(LDAPMessage *res);
```

Description ldap_count_messages() is used to count the number of messages that remain in a chain of results if called with a message, entry, or reference returned by ldap_first_message(), ldap_next_message(), ldap_first_entry(), ldap_next_entry(), ldap_first_reference(), and ldap_next_reference()

ldap_first_message() and ldap_next_message() functions are used to step through the list of messages in a result chain returned by ldap_result().

ldap_msgtype() function returns the type of an LDAP message.

Return Values ldap_first_message() and ldap_next_message() return LDAPMessage which can include referral messages, entry messages and result messages.

ldap_count_messages() returns the number of messages contained in a chain of results.

Errors ldap_first_message() and ldap_next_message() return NULL when no more messages exist. NULL is also returned if an error occurs while stepping through the entries, in which case the error parameters in the session handle *ld* will be set to indicate the error.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap_error(3LDAP), ldap_result(3LDAP), attributes(5)

Name ldap_friendly, ldap_friendly_name, ldap_free_friendlymap – LDAP attribute remapping functions

Synopsis cc[*flag*...] *file*... -lldap[*library*...]
#include <lber.h>
#include <ldap.h>

char *ldap_friendly_name(char **filename*, char **name*,
    FriendlyMap **map*);

void ldap_free_friendlymap(FriendlyMap **map*);

Description This function is used to map one set of strings to another. Typically, this is done for country names, to map from the two-letter country codes to longer more readable names. The mechanism is general enough to be used with other things, though.

filename is the name of a file containing the unfriendly to friendly mapping, *name* is the unfriendly name to map to a friendly name, and *map* is a result-parameter that should be set to NULL on the first call. It is then used to hold the mapping in core so that the file need not be read on subsequent calls.

For example:

```
FriendlyMap *map = NULL;
printf( "unfriendly %s => friendly %s\n", name,
    ldap_friendly_name( "ETCDIR/ldapfriendly", name, &map ) );
```

The mapping file should contain lines like this: unfriendlyname\tfriendlyname. Lines that begin with a '#' character are comments and are ignored.

The ldap_free_friendlymap() call is used to free structures allocated by ldap_friendly_name() when no more calls to ldap_friendly_name() are to be made.

Errors NULL is returned by ldap_friendly_name() if there is an error opening *filename*, or if the file has a bad format, or if the *map* parameter is NULL.

Files ETCDIR/ldapfriendly.conf

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap(3LDAP), attributes(5)

Name ldap_get_dn, ldap_explode_dn, ldap_dn2ufn, ldap_is_dns_dn, ldap_explode_dns, ldap_dns_to_dn – LDAP DN handling functions

Synopsis cc[*flag*...] *file*... -lldap[*library*...]
#include <lber.h>
#include <ldap.h>

char *ldap_get_dn(LDAP *ld*, LDAPMessage *entry*);

char **ldap_explode_dn(char *dn*, int *notypes*);

char *ldap_dn2ufn(char *dn*);

int ldap_is_dns_dn(char *dn*);

char **ldap_explode_dns(char *dn*);

char *ldap_dns_to_dn(char *dns_name*, int *nameparts*);

Description These functions allow LDAP entry names (Distinguished Names, or DNs) to be obtained, parsed, converted to a user-friendly form, and tested. A DN has the form described in RFC 1779 *A String Representation of Distinguished Names*, unless it is an experimental DNS-style DN which takes the form of an *RFC 822* mail address.

The ldap_get_dn() function takes an *entry* as returned by ldap_first_entry(3LDAP) or ldap_next_entry(3LDAP) and returns a copy of the entry's DN. Space for the DN will have been obtained by means of malloc(3C), and should be freed by the caller by a call to free(3C).

The ldap_explode_dn() function takes a DN as returned by ldap_get_dn() and breaks it up into its component parts. Each part is known as a Relative Distinguished Name, or RDN. ldap_explode_dn() returns a null-terminated array, each component of which contains an RDN from the DN. The *notypes* parameter is used to request that only the RDN values be returned, not their types. For example, the DN "cn=Bob, c=US" would return as either { "cn=Bob", "c=US", NULL } or { "Bob", "US", NULL }, depending on whether notypes was 0 or 1, respectively. The result can be freed by calling ldap_value_free(3LDAP).

ldap_dn2ufn() is used to turn a DN as returned by ldap_get_dn() into a more user-friendly form, stripping off type names. See *RFC 1781* "Using the Directory to Achieve User Friendly Naming" for more details on the UFN format. The space for the UFN returned is obtained by a call to malloc(3C), and the user is responsible for freeing it by means of a call to free(3C).

ldap_is_dns_dn() returns non-zero if the dn string is an experimental DNS-style DN (generally in the form of an *RFC 822* e-mail address). It returns zero if the dn appears to be an *RFC 1779* format DN.

ldap_explode_dns() takes a DNS-style DN and breaks it up into its component parts. ldap_explode_dns() returns a null-terminated array. For example, the DN "mcs.umich.edu" will return { "mcs", "umich", "edu", NULL }. The result can be freed by calling ldap_value_free(3LDAP).

ldap_dns_to_dn() converts a DNS domain name into an X.500 distinguished name. A string distinguished name and the number of nameparts is returned.

Errors If an error occurs in ldap_get_dn(), NULL is returned and the ld_errno field in the *ld* parameter is set to indicate the error. See ldap_error(3LDAP) for a description of possible error codes. ldap_explode_dn(), ldap_explode_dns() and ldap_dn2ufn() will return NULL with errno(3C) set appropriately in case of trouble.

If an error in ldap_dns_to_dn() is encountered zero is returned. The caller should free the returned string if it is non-zero.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap(3LDAP), ldap_first_entry(3LDAP), ldap_error(3LDAP), ldap_value_free(3LDAP)

Notes These functions allocate memory that the caller must free.

Name   ldap_get_entry_controls – get the LDAP controls included with a directory entry in a set of search results

Synopsis   cc [*flag ...*] *file...* -lldap [*library ...*]
#include <ldap.h>

int ldap_get_entry_controls(LDAP *ld, LDAPMessage *entry,
    LDAPControl ***serverctrlsp);

Description   The ldap_get_entry_controls() function retrieves the LDAP v3 controls included in a directory entry in a chain of search results. The LDAP controls are specified in an array of LDAPControl structures. Each LDAPControl structure represents an LDAP control. The function takes *entry* as a parameter, which points to an LDAPMessage structure that represents an entry in a chain of search results.

The entry notification controls that are used with persistent search controls are the only controls that are returned with individual entries. Other controls are returned with results sent from the server. You can call ldap_parse_result() to retrieve those controls.

Errors   ldap_get_entry_controls() returns the following error codes.

LDAP_SUCCESS   LDAP controls were successfully retrieved.

LDAP_DECODING_ERROR   An error occurred when decoding the BER-encoded message.

LDAP_PARAM_ERROR   An invalid parameter was passed to the function.

LDAP_NO_MEMORY   Memory cannot be allocated.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also   ldap_error(3LDAP), ldap_parse_result(3LDAP), attributes(5)

Name ldap_getfilter, ldap_init_getfilter, ldap_init_getfilter_buf, ldap_getfilter_free, ldap_getfirstfilter, ldap_getnextfilter, ldap_setfilteraffixes, ldap_build_filter – LDAP filter generating functions

Synopsis cc [*flag*...] *file*... -lldap [*library*...]
#include <lber.h>
#include <ldap.h>
#define LDAP_FILT_MAXSIZ   1024

LDAPFiltDesc *ldap_init_getfilter(char *file);

LDAPFiltDesc *ldap_init_getfilter_buf(char *buf, long buflen);

void ldap_getfilter_free(LDAPFiltDesc *lfdp);

LDAPFiltInfo *ldap_getfirstfilter(LDAPFiltDesc *lfdp, char *tagpat,
     char *value);

LDAPFiltInfo *ldap_getnextfilter(LDAPFiltDesc *lfdp);

void ldap_setfilteraffixes(LDAPFiltDesc *lfdp, char *prefix,
     char *suffix);

void ldap_build_filter(char *buf, unsigned long buflen, char *pattern,
     char *prefix, char *suffix, char *attr, char *value,
     char **valwords);

Description These functions are used to generate filters to be used in ldap_search(3LDAP) or ldap_search_s(3LDAP). Either ldap_init_getfilter or ldap_init_getfilter_buf must be called prior to calling any of the other functions except ldap_build_filter.

ldap_init_getfilter() takes a file name as its only argument. The contents of the file must be a valid LDAP filter configuration file (see ldapfilter.conf(4)). If the file is successfully read, a pointer to an LDAPFiltDesc is returned. This is an opaque object that is passed in subsequent get filter calls.

ldap_init_getfilter_buf() reads from *buf*, whose length is *buflen*, the LDAP filter configuration information. *buf* must point to the contents of a valid LDAP filter configuration file. See ldapfilter.conf(4). If the filter configuration information is successfully read, a pointer to an LDAPFiltDesc is returned. This is an opaque object that is passed in subsequent get filter calls.

ldap_getfilter_free() deallocates the memory consumed by ldap_init_getfilter. Once it is called, the LDAPFiltDesc is no longer valid and cannot be used again.

ldap_getfirstfilter() retrieves the first filter that is appropriate for *value*. Only filter sets that have tags that match the regular expression *tagpat* are considered. ldap_getfirstfilter returns a pointer to an LDAPFiltInfo structure, which contains a filter with *value* inserted as appropriate in lfi_filter, a text match description in lfi_desc, lfi_scope set to indicate the search scope, and lfi_isexact set to indicate the type of filter. NULL is returned if no matching filters are found. lfi_scope will be one of LDAP_SCOPE_BASE,

LDAP_SCOPE_ONELEVEL, or LDAP_SCOPE_SUBTREE. lfi_isexact will be zero if the filter has any '~' or '*' characters in it and non-zero otherwise.

ldap_getnextfilter() retrieves the next appropriate filter in the filter set that was determined when ldap_getfirstfilter was called. It returns NULL when the list has been exhausted.

ldap_setfilteraffixes() sets a *prefix* to be prepended and a *suffix* to be appended to all filters returned in the future.

ldap_build_filter() constructs an LDAP search filter in *buf. buflen* is the size, in bytes, of the largest filter *buf* can hold. A pattern for the desired filter is passed in *pattern.* Where the string %a appears in the pattern it is replaced with *attr. prefix* is pre-pended to the resulting filter, and *suffix* is appended. Either can be NULL , in which case they are not used. *value* and *valwords* are used when the string %v appears in *pattern.* See ldapfilter.conf(4) for a description of how %v is handled.

Errors NULL is returned by ldap_init_getfilter if there is an error reading *file.* NULL is returned by ldap_getfirstfilter and ldap_getnextfilter when there are no more appropriate filters to return.

Files ETCDIR/ldapfilter.conf   LDAP filtering routine configuration file.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap(3LDAP), ldapfilter.conf(4), attributes(5)

Notes The return values for all of these functions are declared in the <ldap.h> header file. Some functions may allocate memory which must be freed by the calling application.

Name ldap_get_lang_values, ldap_get_lang_values_len – return an attribute's values that matches a specified language subtype

Synopsis cc [*flag ...*] *file...* -lldap [*library ...*]
#include <ldap.h>

char **ldap_get_lang_values(LDAP *ld, LDAPMessage *entry,
    const char *target, char **type);

struct berval **ldap_get_lang_values_len(LDAP *ld, LDAPMessage *entry,
    const char *target, char **type);

Description The ldap_get_lang_values() function returns an array of an attribute's string values that matches a specified language subtype. To retrieve the binary data from an attribute, call the ldap_get_lang_values_len() function instead.

ldap_get_lang_values() should be called to retrieve a null-terminated array of an attribute's string values that match a specified language subtype. The *entry* parameter is the entry retrieved from the directory. The *target* parameter should contain the attribute type the values that are required, including the optional language subtype. The *type* parameter points to a buffer that returns the attribute type retrieved by this function. Unlike the ldap_get_values() function, if a language subtype is specified, this function first attempts to find and return values that match that subtype, for example, cn;lang-en.

ldap_get_lang_values_len() returns a null–terminated array of pointers to berval structures, each containing the length and pointer to a binary value of an attribute for a given entry. The *entry* parameter is the result returned by ldap_result() or ldap_search_s() functions. The *target* parameter is the attribute returned by the call to ldap_first_attribute() or ldap_next_attribute(), or the attribute as a literal string, such as jpegPhoto or audio.

These functions are deprecated. Use ldap_get_values() or ldap_get_values_len() instead.

Return Values If successful, ldap_get_lang_values() returns a null-terminated array of the attribute's values. If the call is unsuccessful, or if no such attribute exists in the *entry*, it returns a NULL and sets the appropriate error code in the LDAP structure.

The ldap_get_lang_values_len() function returns a null-terminated array of pointers to berval structures, which in turn, if successful, contain pointers to the attribute's binary values. If the call is unsuccessful, or if no such attribute exists in the *entry*, it returns a NULL and sets the appropriate error code in the LDAP structure.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

See Also   ldap_first_attribute(3LDAP), ldap_first_attribute(3LDAP),
ldap_get_values(3LDAP), ldap_result(3LDAP), ldap_search(3LDAP), attributes(5)

Name ldap_get_option, ldap_set_option – get or set session preferences in the ldap structure.

Synopsis cc [*flag...*] *file...* -lldap [*library...*]
#include <lber.h>
#include <ldap.h>

LDAP ldap_set_option(LDAP *ld*, int *option*, void *optdata*[]);

LDAP ldap_get_option(LDAP *ld*, int *option*, void *optdata*[]);

Description These functions provide an LDAP structure with access to session preferences. The
ldap_get_option() function gets session preferences from the LDAP structure. The
ldap_set_option() function sets session preferences in the LDAP structure.

The *ld* parameter specifies the connection handle, a pointer to an LDAP structure that contains
information about the LDAP server connection. The *option* parameter specifies the name of
the option to be read or modified. The *optdata* parameter serves as a pointer to the value of the
option that you set or get.

Parameters The following values can be specified for the *option* parameter:

LDAP_OPT_API_INFO	Retrieves basic information about the LDAP API implementation at execution time. The data type for the *optdata* parameter is (LDAPAPIInfo *). This option is READ-ONLY and cannot be set.	
LDAP_OPT_DEREF	Determines how aliases are handled during a search. The data type for the *optdata* parameter is (int *). The following values can be specified for the *optdata* parameter:	
	LDAP_DEREF_NEVER	Specifies that aliases are never dereferenced.
	LDAP_DEREF_SEARCHING	Specifies that aliases are dereferenced when searching under the base object, but not when finding the base object.
	LDAP_DEREF_FINDING	Specifies that aliases are dereferenced when finding the base object, but not when searching under the base object.
	LDAP_DEREF_ALWAYS	Specifies that aliases are always dereferenced when finding the base object

	and searching under the base object.
LDAP_OPT_SIZELIMIT	Specifies the maximum number of entries returned by the server in search results. The data type for the *optdata* parameter is (int *). Setting the *optdata* parameter to LDAP_NO_LIMIT removes any size limit enforced by the client.
LDAP_OPT_TIMELIMIT	Specifies the maximum number of seconds spent by the server when answering a search request. The data type for the *optdata* parameter is (int *). Setting the *optdata* parameter to LDAP_NO_LIMIT removes any time limit enforced by the client.
LDAP_OPT_REFERRALS	Determines whether the client should follow referrals. The data type for the *optdata* parameter is (int *). The following values can be specified for the *optdata* parameter:

	LDAP_OPT_ON	Specifies that the client should follow referrals.
	LDAP_OPT_OFF	Specifies that the client should not follow referrals.

	By default, the client follows referrals.
LDAP_OPT_RESTART	Determines whether LDAP I/O operations are automatically restarted if aborted prematurely. It can be set to one of the constants LDAP_OPT_ON or LDAP_OPT_OFF.
LDAP_OPT_PROTOCOL_VERSION	Specifies the version of the protocol supported by the client. The data type for the *optdata* parameter is (int *). The version LDAP_VERSION2 or LDAP_VERSION3 can be specified. If no version is set, the default version LDAP_VERSION2 is set. To use LDAP v3 features, set the protocol version to LDAP_VERSION3.
LDAP_OPT_SERVER_CONTROLS	Specifies a pointer to an array of LDAPControl structures that represent the LDAP v3 server controls sent by default with every request. The data type for the *optdata* parameter for ldap_set_option() is (LDAPControl **). For ldap_get_option(), the data type is (LDAPControl ***).

LDAP_OPT_CLIENT_CONTROLS	Specifies a pointer to an array of LDAPControl structures that represent the LDAP v3 client controls sent by default with every request. The data type for the *optdata* parameter for ldap_set_option() is (LDAPControl **). For ldap_get_option(), the data type is (LDAPControl ***).
LDAP_OPT_API_FEATURE_INFO	Retrieves version information at execution time about extended features of the LDAP API. The data type for the *optdata* parameter is (LDAPAPIFeatureInfo *). This option is READ-ONLY and cannot be set.
LDAP_OPT_HOST_NAME	Sets the host name or a list of hosts for the primary LDAP server. The data type for the *optdata* parameter for ldap_set_option() is (char *). For ldap_get_option(), the data type is (char **).
LDAP_OPT_ERROR_NUMBER	Specifies the code of the most recent LDAP error that occurred for this session. The data type for the *optdata* parameter is (int *).
LDAP_OPT_ERROR_STRING	Specifies the message returned with the most recent LDAP error that occurred for this session. The data type for the *optdata* parameter for ldap_set_option() is (char *) and for ldap_get_option() is (char **).
LDAP_OPT_MATCHED_DN	Specifies the matched DN value returned with the most recent LDAP error that occurred for this session. The data type for the *optdata* parameter for ldap_set_option() is (char *) and for ldap_get_option() is (char **).
LDAP_OPT_REBIND_ARG	Sets the last argument passed to the routine specified by LDAP_OPT_REBIND_FN. This option can also be set by calling the ldap_set_rebind_proc() function. The data type for the *optdata* parameter is (void *).
LDAP_OPT_REBIND_FN	Sets the routine to be called to authenticate a connection with another LDAP server. For example, the option is used to set the routine called during the course of a referral. This option can also be by calling the ldap_set_rebind_proc() function. The data type for the *optdata* parameter is (LDAP_REBINDPROC_CALLBACK *).

LDAP_OPT_X_SASL_MECH	Sets the default SASL mechanism to call `ldap_interactive_bind_s()`. The data type for the *optdata* parameter is (char *).
LDAP_OPT_X_SASL_REALM	Sets the default SASL_REALM. The default SASL_REALM should be used during a SASL challenge in response to a SASL_CB_GETREALM request when using the `ldap_interactive_bind_s()` function. The data type for the *optdata* parameter is (char *).
LDAP_OPT_X_SASL_AUTHCID	Sets the default SASL_AUTHNAME used during a SASL challenge in response to a SASL_CB_AUTHNAME request when using the `ldap_interactive_bind_s()` function. The data type for the *optdata* parameter is (char *).
LDAP_OPT_X_SASL_AUTHZID	Sets the default SASL_USER that should be used during a SASL challenge in response to a SASL_CB_USER request when using the `ldap_interactive_bind_s` function. The data type for the *optdata* parameter is (char *).
LDAP_OPT_X_SASL_SSF	A read-only option used exclusively with the `ldap_get_option()` function. The `ldap_get_option()` function performs a `sasl_getprop()` operation that gets the SASL_SSF value for the current connection. The data type for the *optdata* parameter is (sasl_ssf_t *).
LDAP_OPT_X_SASL_SSF_EXTERNAL	A write-only option used exclusively with the `ldap_set_option()` function. The `ldap_set_option()` function performs a `sasl_setprop()` operation to set the SASL_SSF_EXTERNAL value for the current connection. The data type for the *optdata* parameter is (sasl_ssf_t *).
LDAP_OPT_X_SASL_SECPROPS	A write-only option used exclusively with the `ldap_set_option()`. This function performs a [sasl_setprop(3SASL)](#) operation for the SASL_SEC_PROPS value for the current connection during an `ldap_interactive_bind_s()` operation. The data type for the *optdata* parameter is (char *), a comma delimited string containing text values for any of the SASL_SEC_PROPS that should be set. The text values are:

	noanonymous	Sets the SASL_SEC_NOANONYMOUS flag
	nodict	Sets the SASL_SEC_NODICTIONARY flag
	noplain	Sets the SASL_SEC_NOPLAINTEXT flag
	forwardsec	Sets the SASL_SEC_FORWARD_SECRECY flag
	passcred	Sets the SASL_SEC_PASS_CREDENTIALS flag
	minssf=N	Sets minssf to the integer value N
	maxssf=N	Sets maxssf to the integer value N
	maxbufsize=N	Sets maxbufsize to the integer value N

LDAP_OPT_X_SASL_SSF_MIN   Sets the default SSF_MIN value used during a ldap_interactive_bind_s() operation. The data type for the *optdata* parameter is (char *) numeric string.

LDAP_OPT_X_SASL_SSF_MAX   Sets the default SSF_MAX value used during a ldap_interactive_bind_s() operation. The data type for the *optdata* parameter is (char *) numeric string.

LDAP_OPT_X_SASL_MAXBUFSIZE   Sets the default SSF_MAXBUFSIZE value used during a ldap_interactive_bind_s() operation. The data type for the *optdata* parameter is (char *) numeric string.

Return Values The ldap_set_option() and ldap_get_option() functions return:

LDAP_SUCCESS   If successful

-1   If unsuccessful

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also ldap_init(3LDAP), sasl_setprop(3SASL), attributes(5)

Notes There are other elements in the LDAP structure that should not be changed. No assumptions should be made about the order of elements in the LDAP structure.

Name ldap_get_values, ldap_get_values_len, ldap_count_values, ldap_count_values_len, ldap_value_free, ldap_value_free_len – LDAP attribute value handling functions

Synopsis cc [*flag*...] *file*... -lldap [*library*...]
```
#include <lber.h>
#include <ldap.h>
```

```
char **ldap_get_values(LDAP *ld, LDAPMessage *entry, char *attr);
```

```
struct berval **ldap_get_values_len(LDAP * ld, LDAPMessage *entry,
    char *attr);
```

```
int ldap_count_values(char **vals);
```

```
int ldap_count_values_len(struct berval **vals);
```

```
void ldap_value_free(char **vals);
```

```
void ldap_value_free_len(struct berval vals);
```

Description These functions are used to retrieve and manipulate attribute values from an LDAP entry as returned by ldap_first_entry(3LDAP) or ldap_next_entry(3LDAP). ldap_get_values() takes the *entry* and the attribute *attr* whose values are desired and returns a null-terminated array of the attribute's values. *attr* may be an attribute type as returned from ldap_first_attribute(3LDAP) or ldap_next_attribute(3LDAP), or if the attribute type is known it can simply be given.

The number of values in the array can be counted by calling ldap_count_values(). The array of values returned can be freed by calling ldap_value_free().

If the attribute values are binary in nature, and thus not suitable to be returned as an array of char *'s, the ldap_get_values_len() function can be used instead. It takes the same parameters as ldap_get_values(), but returns a null-terminated array of pointers to berval structures, each containing the length of and a pointer to a value.

The number of values in the array can be counted by calling ldap_count_values_len(). The array of values returned can be freed by calling ldap_value_free_len().

Errors If an error occurs in ldap_get_values() or ldap_get_values_len(), NULL returned and the ld_errno field in the ld parameter is set to indicate the error. See ldap_error(3LDAP) for a description of possible error codes.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also   ldap(3LDAP), ldap_first_entry(3LDAP), ldap_first_attribute(3LDAP), ldap_error(3LDAP), attributes(5)

Notes   These functions allocates memory that the caller must free.

Name ldap_memcache, ldap_memcache_init, ldap_memcache_set, ldap_memcache_get, ldap_memcache_flush, ldap_memcache_destroy, ldap_memcache_update – LDAP client caching functions

Synopsis cc [*flag ...*] *file...* -lldap [*library ...*]
#include <ldap.h>

int ldap_memcache_init(unsigned long *ttl*, unsigned long *size*,
    char **baseDNs*,struct ldap_thread_fns *thread_fns*,
    LDAPMemCache **cachep*);

int ldap_memcache_set(LDAP *ld*, LDAPMemCache **cache*);

int ldap_memcache_get(LDAP *ld*, LDAPMemCache **cachep*);

void ldap_memcache_flush(LDAPMemCache *cache*, char *dn*, int *scope*

void ldap_memcache_destroy(LDAPMemCache *cache*);

void ldap_memcache_update(LDAPMemCache *cache*);

Description Use the ldap_memcache functions to maintain an in-memory client side cache to store search requests. Caching improves performance and reduces network bandwidth when a client makes repeated requests. The *cache* uses search criteria as the key to the cached items. When you send a search request, the *cache* checks the search criteria to determine if that request has been previously stored . If the request was stored, the search results are read from the *cache*.

Make a call to ldap_memcache_init() to create the in-memory client side *cache*. The function passes back a pointer to an LDAPMemCache structure, which represents the *cache*. Make a call to the ldap_memcache_set() function to associate this *cache* with an LDAP connection handle, an LDAP structure. *ttl* is the the maximum amount of time (in seconds) that an item can be cached. If a *ttl* value of 0 is passed, there is no limit to the amount of time that an item can be cached. *size* is the maximum amount of memory (in bytes) that the cache will consume. A zero value of *size* means the cache has no size limit. *baseDNS* is an array of the base DN strings representing the base DNs of the search requests you want cached. If *baseDNS* is not NULL, only the search requests with the specified base DNs will be cached. If *baseDNS* is NULL, all search requests are cached. The *thread_fns* parameter takes an ldap_thread_fns structure specifying the functions that you want used to ensure that the cache is thread-safe. You should specify this if you have multiple threads that are using the same connection handle and cache. If you are not using multiple threads, pass NULL for this parameter.

ldap_memcache_set() associates an in-memory *cache* that you have already created by calling the ldap_memcache_init() function with an LDAP connection handle. The *ld* parameter should be the result of a successful call to ldap_open(3LDAP). The *cache* parameter should be the result of a *cache* created by the ldap_memcache_init() call. After you call this function, search requests made over the specified LDAP connection will use this cache. To disassociate the cache from the LDAP connection handle, make a call to the ldap_bind(3LDAP) or ldap_bind(3LDAP) function. Make a call to ldap_memcache_set() if you want to associate a

cache with multiple LDAP connection handles. For example, call the ldap_memcache_get() function to get the *cache* associated with one connection, then you can call this function and associate the *cache* with another connection.

The ldap_memcache_get() function gets the *cache* associated with the specified connection handle (LDAP structure). This *cache* is used by all search requests made through that connection. When you call this function, the function sets the *cachep* parameter as a pointer to the LDAPMemCache structure that is associated with the connection handle.

ldap_memcache_flush() flushes search requests from the *cache*. If the base DN of a search request is within the scope specified by the *dn* and *scope* arguments, the search request is flushed from the *cache*. If no DN is specified, the entire cache is flushed. The *scope* parameter, along with the *dn* parameter, identifies the search requests that you want flushed from the *cache*. This argument can have one of the following values:

```
LDAP_SCOPE_BASE
LDAP_SCOPE_ONELEVEL
LDAP_SCOPE_SUBTREE
```

ldap_memcache_destroy() frees the specified LDAPMemCache structure pointed to by *cache* from memory. Call this function after you are done working with a *cache*.

ldap_memcache_update() checks the cache for items that have expired and removes them. This check is typically done as part of the way the *cache* normally works. You do not need to call this function unless you want to update the *cache* at this point in time. This function is only useful in a multithreaded application, since it will not return until the *cache* is destroyed.

Parameters

ttl	The maximum amount of time (in seconds) that an item can be cached
size	The maximum amount of memory (in bytes) that the cache will consume.
baseDNs	An array of the base DN strings representing the base DNs of the search requests you want cached
thread_fns	A pointer to the ldap_thread_fns structure structure.
cachep	A pointer to the LDAPMemCache structure
cache	The result of a *cache* created by the ldap_memcache_init() call
ld	The result of a successful call to ldap_open(3LDAP)
dn	The search requests that you want flushed from the *cache*
scope	The search requests that you want flushed from the *cache*

Errors
The functions that have int return values return LDAP_SUCCESS if the operation was successful. Otherwise, they return another LDAP error code. See ldap_error(3LDAP) for a list of the LDAP error codes.

Attributes See [attributes(5)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap_error(3LDAP), ldap_open(3LDAP), ldap_search(3LDAP), attributes(5)

Name ldap_memfree – free memory allocated by LDAP API functions

Synopsis cc [*flag* ...] *file*... -lldap [*library* ...]
#include < lber.h>
#include < ldap.h>

void ldap_memfree(void *p);

Description The ldap_memfree() function frees the memory allocated by certain LDAP API functions that do not have corresponding functions to free memory. These functions include ldap_get_dn(3LDAP), ldap_first_attribute(3LDAP), and ldap_next_attribute(3LDAP).

The ldap_memfree() function takes one parameter, *p*, which is a pointer to the memory to be freed.

Parameters *p*   A pointer to the memory to be freed.

Return Values There are no return values for the ldap_memfree() function.

Errors No errors are defined for the ldap_memfree() function.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap(3LDAP), ldap_first_attribute(3LDAP), ldap_get_dn(3LDAP), ldap_next_attribute(3LDAP), attributes(5)

Name ldap_modify, ldap_modify_s, ldap_mods_free, ldap_modify_ext, ldap_modify_ext_s – LDAP entry modification functions

Synopsis cc[*flag*...] *file*... -lldap[*library*...]
```
#include <lber.h>
#include <ldap.h>
```

```
int ldap_modify(LDAP *ld, char *dn, LDAPMod *mods[]);
```

```
int ldap_modify_s(LDAP *ld, char *dn, LDAPMod *mods[]);
```

```
void ldap_mods_free(LDAPMod **mods, int freemods);
```

```
int ldap_modify_ext(LDAP *ld, char *dn, LDAPMod **mods,
     LDAPControl **serverctrls, LDAPControl **clientctrls, int *msgidp);
```

```
int ldap_modify_ext_s(LDAP *ld, char *dn, LDAPMod **mods,
     LDAPControl **serverctrls, LDAPControl **clientctrls);
```

Description The function ldap_modify_s() is used to perform an LDAP modify operation. *dn* is the DN of the entry to modify, and *mods* is a null-terminated array of modifications to make to the entry. Each element of the *mods* array is a pointer to an LDAPMod structure, which is defined below.

```
typedef struct ldapmod {
    int mod_op;
    char *mod_type;
    union {
    char **modv_strvals;
    struct berval **modv_bvals;
      } mod_vals;
    } LDAPMod;
#define mod_values mod_vals.modv_strvals
#define mod_bvalues mod_vals.modv_bvals
```

The *mod_op* field is used to specify the type of modification to perform and should be one of LDAP_MOD_ADD, LDAP_MOD_DELETE, or LDAP_MOD_REPLACE. The *mod_type* and *mod_values* fields specify the attribute type to modify and a null-terminated array of values to add, delete, or replace respectively.

If you need to specify a non-string value (for example, to add a photo or audio attribute value), you should set *mod_op* to the logical OR of the operation as above (for example, LDAP_MOD_REPLACE) and the constant LDAP_MOD_BVALUES. In this case, *mod_bvalues* should be used instead of *mod_values*, and it should point to a null-terminated array of struct bervals, as defined in <lber.h>.

For LDAP_MOD_ADD modifications, the given values are added to the entry, creating the attribute if necessary. For LDAP_MOD_DELETE modifications, the given values are deleted from the entry, removing the attribute if no values remain. If the entire attribute is to be deleted, the *mod_values* field should be set to NULL. For LDAP_MOD_REPLACE modifications, the attribute

will have the listed values after the modification, having been created if necessary. All modifications are performed in the order in which they are listed.

ldap_modify_s() returns the LDAP error code resulting from the modify operation.

The ldap_modify() operation works the same way as ldap_modify_s(), except that it is asynchronous, returning the message id of the request it initiates, or −1 on error. The result of the operation can be obtained by calling ldap_result(3LDAP).

ldap_mods_free() can be used to free each element of a null-terminated array of mod structures. If *freemods* is non-zero, the *mods* pointer itself is freed as well.

The ldap_modify_ext() function initiates an asynchronous modify operation and returns LDAP_SUCCESS if the request was successfully sent to the server, or else it returns a LDAP error code if not. See ldap_error(3LDAP). If successful, ldap_modify_ext() places the message id of the request in *msgidp*. A subsequent call to ldap_result(3LDAP), can be used to obtain the result of the add request.

The ldap_modify_ext_s() function initiates a synchronous modify operation and returns the result of the operation itself.

Errors ldap_modify_s() returns an LDAP error code, either LDAP_SUCCESS or an error. See ldap_error(3LDAP).

ldap_modify() returns −1 in case of trouble, setting the error field of *ld*.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap(3LDAP), ldap_add(3LDAP), ldap_error(3LDAP), ldap_get_option(3LDAP), attributes(5)

Name   ldap_modrdn, ldap_modrdn_s, ldap_modrdn2, ldap_modrdn2_s, ldap_rename, ldap_rename_s – modify LDAP entry RDN

Synopsis   cc[*flag*...] *file*...-l ldap [*library*...]
#include <lber.h>
#include <ldap.h>

int ldap_modrdn(LDAP *ld*, const char *dn*, const char *newrdn*);

int ldap_modrdn_s(LDAP *ld*, const char *dn*, const char *newrdn*,
    int *deleteoldrdn*);

int ldap_modrdn2(LDAP *ld*, const char *dn*, const char *newrdn*,
    int *deleteoldrdn*);

int ldap_modrdn2_s(LDAP *ld*, const char *dn*,
    const char *newrdn*, int *deleteoldrdn*);

int ldap_rename(LDAP *ld*, const char *dn*, const char *newrdn*,
    const char *newparent*, int *deleteoldrdn*,
    LDAPControl **serverctrls*, LDAPControl **clientctrls*,
    int *msgidp*);

int ldap_rename_s(LDAP *ld*, const char *dn*, const char *newrdn*,
    const char *newparent*, const int *deleteoldrdn*,
    LDAPControl **serverctrls*,LDAPControl **clientctrls*);

Description   The ldap_modrdn() and ldap_modrdn_s() functions perform an LDAP modify RDN (Relative Distinguished Name) operation. They both take *dn*, the DN (Distinguished Name) of the entry whose RDN is to be changed, and *newrdn*, the new RDN, to give the entry. The old RDN of the entry is never kept as an attribute of the entry. ldap_modrdn() is asynchronous. It return the message id of the operation it initiates. ldap_modrdn_s() is synchronous. It returns the LDAP error code that indicates the success or failure of the operation.

The ldap_modrdn2() and ldap_modrdn2_s() functions also perform an LDAP modify RDN operation. They take the same parameters as above. In addition, they both take the *deleteoldrdn* parameter ,which is used as a boolean value to indicate whether or not the old RDN values should be deleted from the entry.

The ldap_rename(), ldap_rename_s() routines are used to change the name, that is, the RDN of an entry. These routines deprecate the ldap_modrdn() and ldap_modrdn_s() routines, as well as ldap_modrdn2() and ldap_modrdn2_s().

The ldap_rename() and ldap_rename_s() functions both support LDAPv3 server controls and client controls.

Errors   The synchronous (_s) versions of these functions return an LDAP error code, either LDAP_SUCCESS or an error. See ldap_error(3LDAP).

The asynchronous versions return −1 in the event of an error, setting the ld_errno field of *ld*. See ldap_error(3LDAP) for more details. Use ldap_result(3LDAP) to determine a

particular unsuccessful result.

Attributes See attributes(5) for a description of the following attributes of the ldap_modrdn(),
ldap_modrdn_s(), ldap_modrdn2() and ldap_modrdn2_s() functions:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Obsolete

The ldap_rename() and ldap_rename_s() functions have the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap(3LDAP), ldap_error(3LDAP), attributes(5)

Name ldap_open, ldap_init – initialize an LDAP session

Synopsis cc [*flag*...] *file*... -lldap [*library*...]
#include <lber.h>
#include <ldap.h>

LDAP *ldap_open(const char **host*, int *port*);

LDAP *ldap_init(const char **host*, int *port*);

Description The ldap_open() function initializes an LDAP session and also opens a connection to an LDAP server before it returns to the caller. Unlike ldap_open(), ldap_init() does not open a connection to the LDAP server until an operation, such as a search request, is performed.

The ldap_open() function is deprecated and should no longer be used. Call ldap_init() instead.

A list of LDAP hostnames or an IPv4 or IPv6 address can be specified with the ldap_open() and ldap_init() functions. The hostname can include a port number, separated from the hostname by a colon (:). A port number included as part of the hostname takes precedence over the *port* parameter. The ldap_open() and ldap_init() functions attempt connections with LDAP hosts in the order listed and return the first successful connection.

Parameters These functions support the following parameters.

host   The hostname, IPv4 or IPv6 address of the host that runs the LDAP server. A space-separated list of hostnames can also be used for this parameter.

port   TCP port number of a connection. Supply the constant LDAP_PORT to obtain the default LDAP port of 389. If a host includes a port number, the default parameter is ignored.

Return Values The ldap_open() and ldap_init() functions return a handle to an LDAP session that contains a pointer to an opaque structure. The structure must be passed to subsequent calls for the session. If a session cannot be initialized, the functions return NULL and errno should be set appropriately.

Various aspects of this opaque structure can be read or written to control the session-wide parameters. Use the ldap_get_option(3LDAP) to access the current option values and the ldap_set_option(3LDAP) to set values for these options.

Examples EXAMPLE 1   Specifying IPv4 and IPv6 Addresses

LDAP sessions can be initialized with hostnames, IPv4 or IPv6 addresses, such as those shown in the following examples.

```
ldap_init("hosta:636 hostb", 389)
ldap_init("192.168.82.110:389", 389)
ldap_init("[fec0::114:a00:20ff:ab3d:83ed]", 389)
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also errno(3C), ldap(3LDAP), ldap_bind(3LDAP), ldap_get_option(3LDAP), ldap_set_option(3LDAP), attributes(5)

Name ldap_parse_result, ldap_parse_extended_result, ldap_parse_sasl_bind_result – LDAP message result parser

Synopsis cc [*flag*...] *file*... -lldap [*library*...]
#include <lber.h>
#include <ldap.h>

int ldap_parse_result(LDAP *ld*, LDAPMessage *res*, int *errcodep*,
    char **matcheddnp*,char **errmsgp*, char ***referralsp*,
    LDAPControl ***serverctrlsp*, int *freeit*);

int ldap_parse_sasl_bind_result(LDAP *ld*, LDAPMessage *res*,
    struct berval **servercredp*,int *freeit*);

int ldap_parse_extended_result(LDAP *ld*, LDAPMessage *res*,
    char **resultoidp*, struct berval **resultdata*, int *freeit*);

Description The ldap_parse_extended_result(),ldap_parse_result() and ldap_parse_sasl_bind_result() routines search for a message to parse. These functions skip messages of type LDAP_RES_SEARCH_ENTRY and LDAP_RES_SEARCH_REFERENCE.

Return Values They return LDAP_SUCCESS if the result was successfully parsed or an LDAP error code if not (see ldap_error(3LDAP)).

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap_error(3LDAP), ldap_result(3LDAP), attributes(5)

Name ldap_result, ldap_msgfree – wait for and return LDAP operation result

Synopsis cc[*flag*...] *file*... -lldap[*library*...]
#include <lber.h>
#include <ldap.h>

int ldap_result(LDAP *ld, int msgid, int all,
    struct timeval *timeout, LDAPMessage **result);

int ldap_msgfree(LDAPMessage *msg);

Description The ldap_result() function is used to wait for and return the result of an operation previously initiated by one of the LDAP asynchronous operation functions , for example, ldap_search(3LDAP), and ldap_modify(3LDAP). Those functions all return −1 in case of error, and an invocation identifier upon successful initiation of the operation. The invocation identifier is picked by the library and is guaranteed to be unique across the LDAP session. It can be used to request the result of a specific operation from ldap_result() through the *msgid* parameter.

The ldap_result() function will block or not, depending upon the setting of the *timeout* parameter. If *timeout* is not a null pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a null pointer, the select blocks indefinitely. To effect a poll, the *timeout* argument should be a non-null pointer, pointing to a zero-valued timeval structure. See select(3C) for further details.

If the result of a specific operation is required, *msgid* should be set to the invocation identifier returned when the operation was initiated, otherwise LDAP_RES_ANY should be supplied. The *all* parameter only has meaning for search responses and is used to select whether a single entry of the search response should be returned, or all results of the search should be returned.

A search response is made up of zero or more search entries followed by a search result. If *all* is set to 0, search entries will be returned one at a time as they come in, by means of separate calls to ldap_result(). If it is set to a non-zero value, the search response will only be returned in its entirety, that is, after all entries and the final search result have been received.

Upon success, the type of the result received is returned and the *result* parameter will contain the result of the operation. This result should be passed to the LDAP parsing functions, (see ldap_first_entry(3LDAP)) for interpretation.

The possible result types returned are:

```
#define LDAP_RES_BIND          0x61L
#define LDAP_RES_SEARCH_ENTRY  0x64L
#define LDAP_RES_SEARCH_RESULT 0x65L
#define LDAP_RES_MODIFY        0x67L
#define LDAP_RES_ADD           0x69L
#define LDAP_RES_DELETE        0x6bL
#define LDAP_RES_MODRDN        0x6dL
#define LDAP_RES_COMPARE       0x6fL
```

The ldap_msgfree() function is used to free the memory allocated for a result by ldap_result() or ldap_search_s(3LDAP) functions. It takes a pointer to the result to be freed and returns the type of the message it freed.

Errors The ldap_result() function returns −1 on error and 0 if the specified timeout was exceeded.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also select(1), ldap(3LDAP), ldap_search(3LDAP), attributes(5)

Notes The ldap_result() function allocates memory for results that it receives. The memory can be freed by calling ldap_msgfree(3LDAP).

Name ldap_search, ldap_search_s, ldap_search_ext, ldap_search_ext_s, ldap_search_st – LDAP search operations

Synopsis cc [*flag...*] *file...* -lldap[*library...*]
```
#include <sys/time.h> /* for struct timeval definition */
#include <lber.h>
#include <ldap.h>
```

```
int ldap_search(LDAP *ld, char *base, int scope, char *filter,
    char *attrs[], int attrsonly);
```

```
int ldap_search_s(LDAP *ld, char *base, int scope, char *filter,
    char *attrs[],int attrsonly, LDAPMessage **res);
```

```
int ldap_search_st(LDAP *ld, char *base, int scope, char *filter,
    char *attrs[], int attrsonly, struct timeval *timeout,
    LDAPMessage **res);
```

```
int ldap_search_ext(LDAP *ld, char *base, int scope, char
    *filter, char **attrs, int attrsonly, LDAPControl **serverctrls,
    LDAPControl **clientctrls, struct timeval *timeoutp,
    int sizelimit, int *msgidp);
```

```
int ldap_search_ext_s(LDAP *ld,char *base, int scope, char *filter,
    char **attrs, int attrsonly, LDAPControl **serverctrls,
    LDAPControl **clientctrls, struct timeval *timeoutp,
    int sizelimit, LDAPMessage **res);
```

Description These functions are used to perform LDAP search operations. The ldap_search_s() function does the search synchronously (that is, not returning until the operation completes). The ldap_search_st() function does the same, but allows a *timeout* to be specified. The ldap_search() function is the asynchronous version, initiating the search and returning the message ID of the operation it initiated.

The *base* is the DN of the entry at which to start the search. The *scope* is the scope of the search and should be one of LDAP_SCOPE_BASE, to search the object itself, LDAP_SCOPE_ONELEVEL, to search the object's immediate children, or LDAP_SCOPE_SUBTREE, to search the object and all its descendents.

The *filter* is a string representation of the filter to apply in the search. Simple filters can be specified as *attributetype=attributevalue*. More complex filters are specified using a prefix notation according to the following BNF:

```
<filter> ::= '(' <filtercomp> ')'
<filtercomp> ::= <and> | <or> | <not> | <simple>
<and> ::= '&' <filterlist>
<or> ::= '|' <filterlist>
<not> ::= '!' <filter>
<filterlist> ::= <filter> | <filter> <filterlist>
<simple> ::= <attributetype> <filtertype> <attributevalue>
<filtertype> ::= '=' | '~=' | '<=' | '>='
```

The '~=' construct is used to specify approximate matching. The representation for <attributetype> and <attributevalue> are as described in RFC 1778. In addition, <attributevalue> can be a single * to achieve an attribute existence test, or can contain text and *'s interspersed to achieve substring matching.

For example, the filter mail=* finds entries that have a mail attribute. The filter mail=*@terminator.rs.itd.umich.edu finds entries that have a mail attribute ending in the specified string. Use a backslash (\) to escape parentheses characters in a filter. See RFC 1588 for a more complete description of the filters that are allowed. See ldap_getfilter(3LDAP) for functions to help construct search filters automatically.

The *attrs* is a null-terminated array of attribute types to return from entries that match *filter*. If NULL is specified, all attributes are returned. The *attrsonly* is set to 1 when attribute types only are wanted. The *attrsonly* is set to 0 when both attributes types and attribute values are wanted.

The *sizelimit* argument returns the number of matched entries specified for a search operation. When *sizelimit* is set to 50, for example, no more than 50 entries are returned. When *sizelimit* is set to 0, all matched entries are returned. The LDAP server can be configured to send a maximum number of entries, different from the size limit specified. If 5000 entries are matched in the database of a server configured to send a maximum number of 500 entries, no more than 500 entries are returned even when *sizelimit* is set to 0.

The ldap_search_ext() function initiates an asynchronous search operation and returns LDAP_SUCCESS when the request is successfully sent to the server. Otherwise, ldap_search_ext() returns an LDAP error code. See ldap_error(3LDAP). If successful, ldap_search_ext() places the message ID of the request in **msgidp*. A subsequent call to ldap_result(3LDAP) can be used to obtain the result of the add request.

The ldap_search_ext_s() function initiates a synchronous search operation and returns the result of the operation itself.

Errors The ldap_search_s() and ldap_search_st() functions return the LDAP error code that results from a search operation. See ldap_error(3LDAP) for details.

The ldap_search() function returns −1 when the operation terminates unsuccessfully.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also ldap(3LDAP), ldap_result(3LDAP), ldap_getfilter(3LDAP), ldap_error(3LDAP), attributes(5)

Howes, T., Kille, S., Yeong, W., Robbins, C., Wenn, J. *RFC 1778, The String Representation of Standard Attribute Syntaxes*. Network Working Group. March 1995.

Postel, J., Anderson, C. *RFC 1588, White Pages Meeting Report*. Network Working Group. February 1994.

Notes   The read and list functionality are subsumed by ldap_search() functions, when a filter such as objectclass=* is used with the scope LDAP_SCOPE_BASE to emulate read or the scope LDAP_SCOPE_ONELEVEL to emulate list.

The ldap_search() functions may allocate memory which must be freed by the calling application. Return values are contained in <ldap.h>.

Name ldap_searchprefs, ldap_init_searchprefs, ldap_init_searchprefs_buf, ldap_free_searchprefs, ldap_first_searchobj, ldap_next_searchobj – LDAP search preference configuration routines

Synopsis cc [*flag*...] *file*... -lldap [*library*...]
#include <lber.h>
#include <ldap.h>

int ldap_init_searchprefs(char ***file,
    struct ldap_searchobj ***solistp);

int ldap_init_searchprefs_buf(char ***buf, unsigned *longlen*,
    struct ldap_searchobj **solistp);

struct ldap_searchobj **ldap_free_searchprefs
    (struct ldap_searchobj **solist);

struct ldap_searchobj **ldap_first_searchobj
    (struct ldap_seachobj **solist);

struct ldap_searchobj **ldap_next_searchobj
    (struct ldap_seachobj **solist, struct ldap_seachobj **so);

Description These functions provide a standard way to access LDAP search preference configuration data. LDAP search preference configurations are typically used by LDAP client programs to specify which attributes a user may search by, labels for the attributes, and LDAP filters and scopes associated with those searches. Client software presents these choices to a user, who can then specify the type of search to be performed.

ldap_init_searchprefs() reads a sequence of search preference configurations from a valid LDAP searchpref configuration file. See ldapsearchprefs.conf(4). Upon success, 0 is returned and *solistp* is set to point to a list of search preference data structures.

ldap_init_searchprefs_buf() reads a sequence of search preference configurations from *buf*, whose size is *buflen*. *buf* should point to the data in the format defined for an LDAP search preference configuration file. See ldapsearchprefs.conf(4). Upon success, 0 is returned and *solistp* is set to point to a list of search preference data structures.

ldap_free_searchprefs() disposes of the data structures allocated by ldap_init_searchprefs().

ldap_first_searchpref() returns the first search preference data structure in the list *solist*. The *solist* is typically obtained by calling ldap_init_searchprefs().

ldap_next_searchpref() returns the search preference after *so* in the template list *solist*. A NULL pointer is returned if *so* is the last entry in the list.

Errors ldap_init_search_prefs() and ldap_init_search_prefs_bufs() return:

LDAP_SEARCHPREF_ERR_VERSION   ***buf* points to data that is newer than can be handled.

LDAP_SEARCHPREF_ERR_MEM   Memory allocation problem.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap(3LDAP), ldapsearchprefs.conf(4), attributes(5)

Yeong, W., Howes, T., and Hardcastle-Kille, S., "Lightweight Directory Access Protocol", OSI-DS-26, April 1992.

Howes, T., Hardcastle-Kille, S., Yeong, W., and Robbins, C., "Lightweight Directory Access Protocol", OSI-DS-26, April 1992.

Hardcastle-Kille, S., "A String Representation of Distinguished Names", OSI-DS-23, April 1992.

Information Processing - Open Systems Interconnection - The Directory, International Organization for Standardization. International Standard 9594, (1988).

Name ldap_sort, ldap_sort_entries, ldap_sort_values, ldap_sort_strcasecmp – LDAP entry sorting functions

Synopsis cc [*flag*...] *file*... -lldap [*library*...]
#include <lber.h>
#include <ldap.h>

```
int ldap_sort_entries(LDAP *ld, LDAPMessage **chain, char *attr,
    int (*cmp)());

int ldap_sort_values(LDAP *ld, char **vals, int (*cmp)());

int ldap_sort_strcasecmp(char *a, char *b);
```

Description These functions are used to sort lists of entries and values retrieved from an LDAP server. ldap_sort_entries() is used to sort a chain of entries retrieved from an LDAP search call either by DN or by some arbitrary attribute in the entries. It takes *ld*, the LDAP structure, which is only used for error reporting, *chain*, the list of entries as returned by ldap_search_s(3LDAP) or ldap_result(3LDAP). *attr* is the attribute to use as a key in the sort or NULL to sort by DN, and *cmp* is the comparison function to use when comparing values (or individual DN components if sorting by DN). In this case, *cmp* should be a function taking two single values of the *attr* to sort by, and returning a value less than zero, equal to zero, or greater than zero, depending on whether the first argument is less than, equal to, or greater than the second argument. The convention is the same as used by qsort(3C), which is called to do the actual sorting.

ldap_sort_values() is used to sort an array of values from an entry, as returned by ldap_get_values(3LDAP). It takes the LDAP connection structure *ld*, the array of values to sort *vals*, and *cmp*, the comparison function to use during the sort. Note that *cmp* will be passed a pointer to each element in the *vals* array, so if you pass the normal char ** for this parameter, *cmp* should take two char **'s as arguments (that is, you cannot pass *strcasecmp* or its friends for *cmp*). You can, however, pass the function ldap_sort_strcasecmp() for this purpose.

For example:

```
LDAP *ld;
LDAPMessage *res;
/* ... call to ldap_search_s( ), fill in res, retrieve sn attr ... */

/* now sort the entries on surname attribute */
if ( ldap_sort_entries( ld, &res, "sn", ldap_sort_strcasecmp ) != 0 )
    ldap_perror( ld, "ldap_sort_entries" );
```

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also ldap(3LDAP), ldap_search(3LDAP), ldap_result(3LDAP), qsort(3C), attributes(5)

Notes The ldap_sort_entries() function applies the comparison function to each value of the attribute in the array as returned by a call to ldap_get_values(3LDAP), until a mismatch is found. This works fine for single-valued attributes, but may produce unexpected results for multi-valued attributes. When sorting by DN, the comparison function is applied to an exploded version of the DN, without types. The return values for all of these functions are declared in the <ldap.h> header file. Some functions may allocate memory which must be freed by the calling application.

Name ldap_ufn, ldap_ufn_search_s, ldap_ufn_search_c, ldap_ufn_search_ct, ldap_ufn_setfilter, ldap_ufn_setprefix, ldap_ufn_timeout – LDAP user friendly search functions

Synopsis cc[*flag*...] *file*... -lldap[*library*...]
```
#include <lber.h>
#include <ldap.h>
```
```
int ldap_ufn_search_c(LDAP *ld, char *ufn, char **attrs,
     int attrsonly, LDAPMessage **res, int (*cancelproc)(),
     void *cancelparm);
```
```
int ldap_ufn_search_ct(LDAP *ld, char *ufn, char **attrs,
     int attrsonly, LDAPMessage **res,int (*cancelproc)(),
     void *cancelparm,char *tag1, char *tag2,
     char *tag3);
```
```
int ldap_ufn_search_s(LDAP *ld, char *ufn, char **attrs,
     int attrsonly, LDAPMessage **res);
```
```
LDAPFiltDesc *ldap_ufn_setfilter(LDAP *ld, char *fname);
```
```
void ldap_ufn_setprefix(LDAP *ld, char *prefix);
```
```
int ldap_ufn_timeout(void *tvparam);
```

Description These functions are used to perform LDAP user friendly search operations.
ldap_ufn_search_s() is the simplest form. It does the search synchronously. It takes *ld* to identify the the LDAP connection. The *ufn* parameter is the user friendly name for which to search. The *attrs*, *attrsonly* and *res* parameters are the same as for ldap_search(3LDAP).

The ldap_ufn_search_c() function functions the same as ldap_ufn_search_s(), except that it takes *cancelproc*, a function to call periodicly during the search. It should be a function taking a single void * argument, given by *calcelparm*. If *cancelproc* returns a non-zero result, the search will be abandoned and no results returned. The purpose of this function is to provide a way for the search to be cancelled, for example, by a user or because some other condition occurs.

The ldap_ufn_search_ct() function is like ldap_ufn_search_c(), except that it takes three extra parameters. *tag1* is passed to the ldap_init_getfilter(3LDAP) function when resolving the first component of the UFN. *tag2* is used when resolving intermediate components. *tag3* is used when resolving the last component. By default, the tags used by the other UFN search functions during these three phases of the search are "ufn first", "ufn intermediate", and "ufn last".

The ldap_ufn_setfilter() function is used to set the ldapfilter.conf(4) file for use with the ldap_init_getfilter(3LDAP) function to *fname*.

The ldap_ufn_setprefix() function is used to set the default prefix (actually, it's a suffix) appended to UFNs before searhing. UFNs with fewer than three components have the prefix appended first, before searching. If that fails, the UFN is tried with progressively shorter

versions of the prefix, stripping off components. If the UFN has three or more components, it is tried by itself first. If that fails, a similar process is applied with the prefix appended.

The ldap_ufn_timeout() function is used to set the timeout associated with ldap_ufn_search_s() searches. The *timeout* parameter should actually be a pointer to a struct timeval. This is so ldap_ufn_timeout() can be used as a cancelproc in the above functions.

Attributes   See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed

See Also   gettimeofday(3C), ldap(3LDAP), ldap_search(3LDAP), ldap_getfilter(3LDAP), ldapfilter.conf(4), ldap_error(3LDAP), attributes(5)

Notes   These functions may allocates memory. Return values are contained in <ldap.h>.

Name   ldap_url, ldap_is_ldap_url, ldap_url_parse, ldap_url_parse_nodn, ldap_free_urldesc, ldap_url_search, ldap_url_search_s, ldap_url_search_st, ldap_dns_to_url, ldap_dn_to_url – LDAP Uniform Resource Locator functions

Synopsis   cc [*flag*...] *file*... -lldap [*library*...]
```
#include <lber.h>
#include <ldap.h>
```

```
int ldap_is_ldap_url(char *url);
```

```
int ldap_url_parse(char *url, LDAPURLDesc **ludpp);
```

```
int ldap_url_parse_nodn(char *url, LDAPURLDesc **ludpp);
```

```
void ldap_free_urldesc(LDAPURLDesc *ludp);
```

```
int ldap_url_search(LDAP *ld, char *url, int attrsonly);
```

```
int ldap_url_search_s(LDAP *ld, char *url,
     int attrsonly, LDAPMessage **res);
```

```
int ldap_url_search_st(LDAP *ld, char *url, int attrsonly,
     struct timeval *timeout, LDAPMessage **res);
```

```
char *ldap_dns_to_url(LDAP *ld, char *dns_name, char *attrs,
     char *scope, char *filter);
```

```
char *ldap_dn_to_url(LDAP *ld, char *dn, int nameparts);
```

Description   These functions support the use of LDAP URLs (Uniform Resource Locators). The following shows the formatting used for LDAP URLs.

ldap://*hostport*/*dn*[?*attributes*[?*scope*[?*filter*]]]

where:

hostport	Host name with an optional :portnumber.
dn	Base DN to be used for an LDAP search operation.
attributes	Comma separated list of attributes to be retrieved.
scope	One of these three strings: base one sub (default=base).
filter	LDAP search filter as used in a call to ldap_search(3LDAP).

The following is an example of an LDAP URL:

ldap://ldap.itd.umich.edu/c=US?o,description?one?o=umich

URLs preceded URL: or wrapped in angle-brackets are tolerated. URLs can also be preceded by URL: and wrapped in angle-brackets.

ldap_is_ldap_url() returns a non-zero value if *url* looks like an LDAP URL (as opposed to some other kind of URL). It can be used as a quick check for an LDAP URL; the ldap_url_parse() function should be used if a more thorough check is needed.

ldap_url_parse() breaks down an LDAP URL passed in *url* into its component pieces. If successful, zero is returned, an LDAP URL description is allocated, filled in, and *ludpp* is set to point to it. See RETURN VALUES for values returned upon error.

ldap_url_parse_nodn() acts just like ldap_url_parse() but does not require *dn* in the LDAP URL.

ldap_free_urldesc() should be called to free an LDAP URL description that was obtained from a call to ldap_url_parse().

ldap_url_search() initiates an asynchronous LDAP search based on the contents of the *url* string. This function acts just like ldap_search(3LDAP) except that many search parameters are pulled out of the URL.

ldap_url_search_s() performs a synchronous LDAP search based on the contents of the *url* string. This function acts just like ldap_search_s(3LDAP) except that many search parameters are pulled out of the URL.

ldap_url_search_st() performs a synchronous LDAP URL search with a specified *timeout*. This function acts just like ldap_search_st(3LDAP) except that many search parameters are pulled out of the URL.

ldap_dns_to_url() locates the LDAP URL associated with a DNS domain name. The supplied DNS domain name is converted into a distinguished name. The directory entry specified by that distinguished name is searched for a labeled URI attribute. If successful then the corresponding LDAP URL is returned. If unsuccessful then that entry's parent is searched and so on until the target distinguished name is reduced to only two nameparts. If *dns_name* is NULL then the environment variable LOCALDOMAIN is used. If *attrs* is not NULL then it is appended to the URL's attribute list. If *scope* is not NULL then it overrides the URL's scope. If *filter* is not NULL then it is merged with the URL's filter. If an error is encountered then zero is returned, otherwise a string URL is returned. The caller should free the returned string if it is non-zero.

ldap_dn_to_url() locates the LDAP URL associated with a distinguished name. The number of nameparts in the supplied distinguished name must be provided. The specified directory entry is searched for a labeledURI attribute. If successful then the LDAP URL is returned. If unsuccessful then that entry's parent is searched and so on until the target distinguished name is reduced to only two nameparts. If an error is encountered then zero is returned, otherwise a string URL is returned. The caller should free the returned string if it is non-zero.

Return Values Upon error, one of these values is returned for ldap_url_parse():

LDAP_URL_ERR_BADSCOPE   URL scope string is invalid.

LDAP_URL_ERR_HOSTPORT   URL hostport is invalid.

LDAP_URL_ERR_MEM   Can't allocate memory space.

LDAP_URL_ERR_NODN   URL has no DN (required).

LDAP_URL_ERR_NOTLDAP   URL doesn't begin with ldap://.

Attributes See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also ldap(3LDAP), ldap_search(3LDAP), attributes(5)

An LDAP URL Format, Tim Howes and Mark Smith, December 1995. Internet Draft (work in progress). Currently available at this URL.

ftp://ds.internic.net/internet-drafts/draft-ietf-asid-ldap-format-03.txt

Name ldap_version – get version information about the LDAP SDK for C

Synopsis cc [*flag* ...] *file*... -lldap [*library* ...]
#include <ldap.h>

int ldap_version(LDAPVERSION *ver);

Description A call to this function returns the version information for the LDAP SDK for C. This is a deprecated function. Use ldap_get_option(3LDAP) instead. The version information is returned in the LDAPVersion structure pointed to by *ver*. If NULL is passed for *ver*, then only the SDK version will be returned.

Return Values The ldap_version() function returns the version number of the LDAP SDK for C, multiplied by 100. For example, for version 1.0 of the LDAP SDK for C, the function returns 100.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Obsolete

See Also ldap_get_option(3LDAP), attributes(5)

Name listen – listen for connections on a socket

Synopsis cc [*flag* ...] *file* ... -lsocket -lnsl [*library* ...]
#include <sys/types.h>
#include <sys/socket.h>

int listen(int *s*, int *backlog*);

Description To accept connections, a socket is first created with socket(3SOCKET), a backlog for incoming connections is specified with listen() and then the connections are accepted with accept(3SOCKET). The listen() call applies only to sockets of type SOCK_STREAM or SOCK_SEQPACKET.

The *backlog* parameter defines the maximum length the queue of pending connections may grow to.

If a connection request arrives with the queue full, the client will receive an error with an indication of ECONNREFUSED for AF_UNIX sockets. If the underlying protocol supports retransmission, the connection request may be ignored so that retries may succeed. For AF_INET and AF_INET6 sockets, the TCP will retry the connection. If the *backlog* is not cleared by the time the tcp times out, the connect will fail with ETIMEDOUT.

Return Values A 0 return value indicates success; −1 indicates an error.

Errors The call fails if:

EBADF   The argument *s* is not a valid file descriptor.

ENOTSOCK   The argument *s* is not a socket.

EOPNOTSUPP   The socket is not of a type that supports the operation listen().

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also accept(3SOCKET), connect(3SOCKET), socket(3SOCKET), attributes(5), socket.h(3HEAD)

Notes There is currently no *backlog* limit.

Name   listen – listen for socket connections and limit the queue of incoming connections

Synopsis   cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <sys/socket.h>

int listen(int *socket*, int *backlog*);

Description   The listen() function marks a connection-mode socket, specified by the *socket* argument, as accepting connections, and limits the number of outstanding connections in the socket's listen queue to the value specified by the *backlog* argument.

If listen() is called with a *backlog* argument value that is less than 0, the function sets the length of the socket's listen queue to 0.

The implementation may include incomplete connections in the queue subject to the queue limit. The implementation may also increase the specified queue limit internally if it includes such incomplete connections in the queue subject to this limit.

Implementations may limit the length of the socket's listen queue. If *backlog* exceeds the implementation-dependent maximum queue length, the length of the socket's listen queue will be set to the maximum supported value.

The socket in use may require the process to have appropriate privileges to use the listen() function.

Return Values   Upon successful completions, listen() returns 0. Otherwise, −1 is returned and errno is set to indicate the error.

Errors   The listen() function will fail if:

EBADF   The *socket* argument is not a valid file descriptor.

EDESTADDRREQ   The socket is not bound to a local address, and the protocol does not support listening on an unbound socket.

EINVAL   The *socket* is already connected.

ENOTSOCK   The *socket* argument does not refer to a socket.

EOPNOTSUPP   The socket protocol does not support listen().

The listen() function may fail if:

EACCES   The calling process does not have the appropriate privileges.

EINVAL   The *socket* has been shut down.

ENOBUFS   Insufficient resources are available in the system to complete the call.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also accept(3XNET), connect(3XNET), socket(3XNET), attributes(5), standards(5)

Name   netdir, netdir_getbyname, netdir_getbyaddr, netdir_free, netdir_options, taddr2uaddr, uaddr2taddr, netdir_perror, netdir_sperror, netdir_mergeaddr – generic transport name-to-address translation

Synopsis   cc [*flag*...] *file*... -lnsl [*library*...]
```
#include <netdir.h>
```
```
int netdir_getbyname(struct netconfig  *config,
    struct nd_hostserv *service, struct nd_addrlist **addrs);
```
```
int netdir_getbyaddr(struct netconfig  *config,
    struct nd_hostservlist **service, struct netbuf  *netaddr);
```
```
void netdir_free(void *ptr, int struct_type);
```
```
int netdir_options(struct netconfig  *config, int option, int fildes,
    char *point_to_args);
```
```
char *taddr2uaddr(struct netconfig *config, struct netbuf *addr);
```
```
struct netbuf *uaddr2taddr(struct netconfig *config, char *uaddr);
```
```
void netdir_perror(char *s);
```
```
char *netdir_sperror(void);
```

Description   The netdir functions provide a generic interface for name-to-address mapping that will work with all transport protocols. This interface provides a generic way for programs to convert transport specific addresses into common structures and back again. The netconfig structure, described on the netconfig(4) manual page, identifies the transport.

The netdir_getbyname() function maps the machine name and service name in the nd_hostserv structure to a collection of addresses of the type understood by the transport identified in the netconfig structure. This function returns all addresses that are valid for that transport in the nd_addrlist structure. The nd_hostserv structure contains the following members:

```
char *h_host;      /* host name */
char *h_serv;      /* service name */
```

The nd_addrlist structure contains the following members:

```
int   n_cnt;       /* number of addresses */
struct netbuf *n_addrs;
```

The netdir_getbyname() function accepts some special-case host names. The host names are defined in <netdir.h>. The currently defined host names are:

HOST_SELF   Represents the address to which local programs will bind their endpoints. HOST_SELF differs from the host name provided by gethostname(3C), which represents the address to which *remote* programs will bind their endpoints.

HOST_ANY   Represents any host accessible by this transport provider. HOST_ANY allows applications to specify a required service without specifying a particular host name.

HOST_SELF_CONNECT   Represents the host address that can be used to connect to the local host.

HOST_BROADCAST   Represents the address for all hosts accessible by this transport provider. Network requests to this address are received by all machines.

All fields of the nd_hostserv structure must be initialized.

To find the address of a given host and service on all available transports, call the netdir_getbyname() function with each struct netconfig structure returned by getnetconfig(3NSL).

The netdir_getbyaddr() function maps addresses specified in the netbuf structure pointed to by *netaddr* to service names. The function returns *service*, a list of host and service pairs that yield these addresses. If more than one tuple of host and service name is returned, the first tuple contains the preferred host and service names:

```
struct nd_hostservlist {
   int  *h_cnt;                  /* number of hostservs found */
   struct hostserv *h_hostservs;
}
```

The netdir_free() structure is used to free the structures allocated by the name to address translation functions. The *ptr* parameter points to the structure that has to be freed. The parameter struct_type identifies the structure:

```
struct netbuf              ND_ADDR
struct nd_addrlist         ND_ADDRLIST
struct hostserv            ND_HOSTSERV
struct nd_hostservlist     ND_HOSTSERVLIST
```

The free() function is used to free the universal address returned by the taddr2uaddr() function.

The netdir_options() function is used to do all transport-specific setups and option management. *fildes* is the associated file descriptor. *option*, *fildes*, and *pointer_to_args* are passed to the netdir_options() function for the transport specified in *config*. Currently four values are defined for *option*:

```
ND_SET_BROADCAST
ND_SET_RESERVEDPORT
ND_CHECK_RESERVEDPORT
```

ND_MERGEADDR

The taddr2uaddr() and uaddr2taddr() functions support translation between universal addresses and TLI type netbufs. The taddr2uaddr() function takes a struct netbuf data structure and returns a pointer to a string that contains the universal address. It returns NULL if the conversion is not possible. This is not a fatal condition as some transports do not support a universal address form.

The uaddr2taddr() function is the reverse of the taddr2uaddr() function. It returns the struct netbuf data structure for the given universal address.

If a transport provider does not support an option, netdir_options returns -1 and the error message can be printed through netdir_perror() or netdir_sperror().

The specific actions of each option follow.

ND_SET_BROADCAST
Sets the transport provider up to allow broadcast if the transport supports broadcast. *fildes* is a file descriptor into the transport, that is, the result of a t_open of /dev/udp. *pointer_to_args* is not used. If this completes, broadcast operations can be performed on file descriptor *fildes*.

ND_SET_RESERVEDPORT
Allows the application to bind to a reserved port if that concept exists for the transport provider. *fildes* is an unbound file descriptor into the transport. If *pointer_to_args* is NULL, *fildes* is bound to a reserved port. If *pointer_to_args* is a pointer to a netbuf structure, an attempt is made to bind to any reserved port on the specified address.

ND_CHECK_RESERVEDPORT
Used to verify that the address corresponds to a reserved port if that concept exists for the transport provider. *fildes* is not used. *pointer_to_args* is a pointer to a netbuf structure that contains the address. This option returns 0 only if the address specified in *pointer_to_args* is reserved.

ND_MERGEADDR
Used to take a "local address" such as a 0.0.0.0 TCP address and return a "real address" to which client machines can connect. *fildes* is not used. *pointer_to_args* is a pointer to a struct nd_mergearg which has the following members:

```
char s_uaddr;    /* server's universal address */
char c_uaddr;    /* client's universal address */
char m_uaddr;     /* the result */
```

If s_uaddr is an address such as 0.0.0.0.1.12, and the call is successful m_uaddr is set to an address such as 192.11.109.89.1.12. For most transports, m_uaddr is identical to s_uaddr.

Return Values The netdir_perror() function prints an error message in standard output that states the cause of a name-to-address mapping failure. The error message is preceded by the string given as an argument.

The netdir_sperror() function returns a string with an error message that states the cause of a name-to-address mapping failure.

The netdir_sperror() function returns a pointer to a buffer which contains the error message string. The buffer is overwritten on each call. In multithreaded applications, thise buffer is implemented as thread-specific data.

The netdir_getbyaddr() function returns 0 on success and a non-zero value on failure.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also gethostname(3C), getnetconfig(3NSL), getnetpath(3NSL), netconfig(4), attributes(5)

Name ns_sign, ns_sign_tcp, ns_sign_tcp_init, ns_verify, ns_verify_tcp, ns_verify_tcp_init,
ns_find_tsig – TSIG system

Synopsis cc [*flag*...] *file*... -lresolv -lsocket -lnsl [*library*...]
```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
int ns_sign(u_char *msg, int *msglen, int msgsize, int error, void *k,
    const u_char *querysig, int querysiglen, u_char *sig, int *siglen,
    time_t in_timesigned);
```

```
int ns_sign_tcp(u_char *msg, int *msglen, int msgsize, int error,
    ns_tcp_tsig_state *state, int done);
```

```
int ns_sign_tcp_init(void *k, const u_char *querysig, int querysiglen,
    ns_tcp_tsig_state *state);
```

```
int ns_verify(u_char *msg, int *msglen, void *k, const u_char *querysig,
    int querysiglen, u_char *sig, int *siglen, time_t in_timesigned,
    int nostrip);
```

```
int ns_verify_tcp(u_char *msg, int *msglen, ns_tcp_tsig_state *state,
    int required);
```

```
int ns_verify_tcp_init(void *k, const u_char *querysig, int querysiglen,
    ns_tcp_tsig_state *state);
```

```
u_char *ns_find_tsig(u_char *msg, u_char *eom);
```

Parameters

ns_sign() *msg*   the incoming DNS message, which will be modified

msglen   the length of the DNS message, on input and output

msgsize   the size of the buffer containing the DNS message on input

error   the value to be placed in the TSIG error field

k   the (DST_KEY *) to sign the data

querysig   for a response, the signature contained in the query

querysiglen   the length of the query signature

sig   a buffer to be filled with the generated signature

siglen   the length of the signature buffer on input, the signature length on output

ns_sign_tcp() *msg*   the incoming DNS message, which will be modified

msglen   the length of the DNS message, on input and output

msgsize   the size of the buffer containing the DNS message on input

	error	the value to be placed in the TSIG error field
	state	the state of the operation
	done	non-zero value signifies that this is the last packet
ns_sign_tcp_init()	*k*	the (DST_KEY *) to sign the data
	querysig	for a response, the signature contained in the query
	querysiglen	the length of the query signature
	state	the state of the operation, which this initializes
ns_verify()	*msg*	the incoming DNS message, which will be modified
	msglen	the length of the DNS message, on input and output
	k	the (DST_KEY *) to sign the data
	querysig	for a response, the signature contained in the query
	querysiglen	the length of the query signature
	sig	a buffer to be filled with the signature contained
	siglen	the length of the signature buffer on input, the signature length on output
	nostrip	non-zero value means that the TSIG is left intact
ns_verify_tcp()	*msg*	the incoming DNS message, which will be modified
	msglen	the length of the DNS message, on input and output
	state	the state of the operation
	required	non-zero value signifies that a TSIG record must be present at this step
ns_verify_tcp_init()	*k*	the (DST_KEY *) to verify the dat
	querysig	for a response, the signature contained in the quer
	querysiglen	the length of the query signature
	state	the state of the operation, which this initializes
ns_find_tsig()	*msg*	the incoming DNS messag
	eom	the length of the DNS message

Description   The TSIG functions are used to implement transaction/request security of DNS messages.

The ns_sign() and ns_verify() functions are the basic routines. The ns_sign_tcp() and ns_verify_tcp() functions are used to sign/verify TCP messages that may be split into multiple packets, such as zone transfers. The ns_sign_tcp_init() and

ns_verify_tcp_init() functions initialize the state structure necessary for TCP operations. The ns_find_tsig() function locates the TSIG record in a message if one is present.

Return Values The ns_find_tsig() function returns a pointer to the TSIG record if one is found, and NULL otherwise.

All other functions return 0 on success, modifying arguments when necessary.

The ns_sign() and ns_sign_tcp() functions return the following values:

-1	bad input data
-ns_r_badkey	The key was invalid or the signing failed.
NS_TSIG_ERROR_NO_SPACE	The message buffer is too small.

The ns_verify() and ns_verify_tcp() functions return the following values:

-1	bad input data
NS_TSIG_ERROR_FORMERR	The message is malformed.
NS_TSIG_ERROR_NO_TSIG	The message does not contain a TSIG record.
NS_TSIG_ERROR_ID_MISMATCH	The TSIG original ID field does not match the message ID.
-ns_r_badkey	Verification failed due to an invalid key.
-ns_r_badsig	Verification failed due to an invalid signature.
-ns_r_badtime	Verification failed due to an invalid timestamp.
ns_r_badkey	Verification succeeded but the message had an error of BADKEY.
ns_r_badsig	Verification succeeded but the message had an error of BADSIG.
ns_r_badtime	Verification succeeded but the message had an error of BADTIME.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also resolver(3RESOLV), attributes(5)

Name   rcmd, rcmd_af, rresvport, rresvport_af, ruserok – routines for returning a stream to a remote command

Synopsis   cc [*flag* ...] *file*... -lsocket -lnsl [*library*...]
#include <netdb.h>
#include <unistd.h>

int rcmd(char **ahost, unsigned short *inport*, const char *luser,
    const char *ruser, const char *cmd, int *fd2p);

int rcmd_af(char **ahost, unsigned short *inport*, const char *luser,
    const char *ruser, const char *cmd, int *fd2p, int *af*);

int rresvport(int *port);

int rresvport_af(int *port, int *af*);

int ruserok(const char *rhost, int *suser*, const char *ruser,
    const char *luser);

Description   The rcmd() function is used by the superuser to execute a command on a remote machine with an authentication scheme based on reserved port numbers. An AF_INET socket is returned with rcmd(). The rcmd_af() function supports AF_INET, AF_INET6 or AF_UNSPEC for the address family. An application can choose which type of socket is returned by passing AF_INET or AF_INET6 as the address family. The use of AF_UNSPEC means that the caller will accept any address family. Choosing AF_UNSPEC provides a socket that best suits the connectivity to the remote host.

The rresvport() function returns a descriptor to a socket with an address in the privileged port space. The rresvport_af() function is the equivalent to rresvport(), except that you can choose AF_INET or AF_INET6 as the socket address family to be returned by rresvport_af(). AF_UNSPEC does not apply to the rresvport() function.

The ruserok() function is a routine used by servers to authenticate clients that request as service with rcmd.

All of these functions are present in the same file and are used by the in.rshd(1M) server among others.

The rcmd() and rcmd_af() functions look up the host *ahost using getaddrinfo(3SOCKET) and return −1 if the host does not exist. Otherwise, *ahost is set to the standard name of the host and a connection is established to a server residing at the Internet port *inport*.

If the connection succeeds, a socket in the Internet domain of type SOCK_STREAM is returned to the caller. The socket is given to the remote command as standard input (file descriptor 0) and standard output (file descriptor 1). If *fd2p* is non-zero, an auxiliary channel to a control process is set up and a descriptor for it is placed in *fd2p. The control process returns diagnostic output file (descriptor 2) from the command on the auxiliary channel. The control process also accepts bytes on this channel as signal numbers to be forwarded to the process

group of the command. If *fd2p* is 0, the standard error (file descriptor 2) of the remote command is made the same as its standard output. No provision is made for sending arbitrary signals to the remote process, other than possibly sending out-of-band data.

The protocol is described in detail in in.rshd(1M).

The rresvport() and rresvport_af() functions are used to obtain a socket bound to a privileged port number. The socket is suitable for use by rcmd() and rresvport_af() and several other routines. Privileged Internet ports are those in the range 1 to 1023. Only the superuser is allowed to bind a socket to a privileged port number. The application must pass in *port*, which must be in the range 512 to 1023. The system first tries to bind to that port number. If it fails, the system then tries to bind to another unused privileged port, if one is available.

The ruserok() function takes a remote host name returned by the gethostbyaddr() function with two user names and a flag to indicate whether the local user's name is that of the superuser. See gethostbyname(3NSL). The ruserok() function then checks the files /etc/hosts.equiv and possibly .rhosts in the local user's home directory to see if the request for service is allowed. A 0 value is returned if the machine name is listed in the /etc/hosts.equiv file, or if the host and remote user name are found in the .rhosts file. Otherwise, the ruserok() function returns −1. If the superuser flag is 1, the /etc/hosts.equiv is not checked.

The error code EAGAIN is overloaded to mean "All network ports in use."

Return Values The rcmd() and rcmd_af() functions return a valid socket descriptor upon success. The functions returns −1 upon error and print a diagnostic message to standard error.

The rresvport() and rresvport_af() functions return a valid, bound socket descriptor upon success. The functions return −1 upon error with the global value errno set according to the reason for failure.

Files /etc/hosts.equiv   system trusted hosts and users

~/.rhosts   user's trusted hosts and users

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

This interface is Unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

See Also   rlogin(1), rsh(1), in.rexecd(1M), in.rshd(1M), Intro(2), getaddrinfo(3SOCKET), gethostbyname(3NSL), rexec(3SOCKET), attributes(5)

Name   recv, recvfrom, recvmsg – receive a message from a socket

Synopsis   cc [*flag*...] *file*... -lsocket -lnsl [*library*...]
#include <sys/socket.h>

ssize_t recv(int *s*, void *buf*, size_t *len*, int *flags*);

ssize_t recvfrom(int *s*, void *buf*, size_t *len*, int *flags*,
    struct sockaddr *from*, socklen_t *fromlen*);

ssize_t recvmsg(int *s*, struct msghdr *msg*, int *flags*);

Description   The recv(), recvfrom(), and recvmsg() functions are used to receive messages from another
socket. The *s* socket is created with socket(3SOCKET).

If *from* is a non-NULL pointer, the source address of the message is filled in. The value-result
parameter *fromlen* is initialized to the size of the buffer associated with *from* and modified on
return to indicate the actual size of the address stored in the buffer. The length of the message
is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded
depending on the type of socket from which the message is received. See socket(3SOCKET).

If no messages are available at the socket, the receive call waits for a message to arrive. If the
socket is non-blocking, -1 is returned with the external variable errno set to EWOULDBLOCK. See
fcntl(2).

For processes on the same host, recvmsg() can be used to receive a file descriptor from
another process, but it cannot receive ancillary data. See libxnet(3LIB).

If a zero-length buffer is specified for a message, an EOF condition results that is
indistinguishable from the successful transfer of a file descriptor. For that reason, one or more
bytes of data should be provided when recvmsg() passes a file descriptor.

The select(3C) call can be used to determine when more data arrives.

The *flags* parameter is formed by an OR operation on one or more of the following:

MSG_OOB   Read any *out-of-band* data present on the socket rather than the regular
               in-band data.

MSG_PEEK   Peek at the data present on the socket. The data is returned, but not
               consumed to allow a subsequent receive operation to see the same data.

MSG_WAITALL   Messages are blocked until the full amount of data requested is returned.
               The recv() function can return a smaller amount of data if a signal is
               caught, the connection is terminated, MSG_PEEK is specified, or if an error is
               pending for the socket.

MSG_DONTWAIT   Pending messages received on the connection are returned. If data is
               unavailable, the function does not block. This behavior is the equivalent to
               specifying O_NONBLOCK on the file descriptor of a socket, except that write

requests are unaffected.

The recvmsg() function call uses a msghdr structure defined in <sys/socket.h> to minimize the number of directly supplied parameters.

Return Values Upon successful completion, these functions return the number of bytes received. Otherwise, they return -1 and set errno to indicate the error.

Errors The recv(), recvfrom(), and recvmsg() functions return errors under the following conditions:

EBADF	The *s* file descriptor is invalid.
EINVAL	The MSG_OOB flag is set and no out-of-band data is available.
EINTR	The operation is interrupted by the delivery of a signal before any data is available to be received.
EIO	An I/O error occurs while reading from or writing to the file system.
ENOMEM	Insufficient user memory is available to complete operation.
ENOSR	Insufficient STREAMS resources are available for the operation to complete.
ENOTSOCK	*s* is not a socket.
ESTALE	A stale NFS file handle exists.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.
ECONNREFUSED	The requested connection was refused by the peer. For connected IPv4 and IPv6 datagram sockets, this indicates that the system received an ICMP Destination Port Unreachable message from the peer.

The recv() and recvfrom() functions fail under the following conditions:

EINVAL   The *len* argument overflows a ssize_t.

The recvmsg() function returns errors under the following conditions:

EINVAL   The msg_iovlen member of the msghdr structure pointed to by *msg* is less than or equal to 0, or greater than [IOV_MAX]. See Intro(2) for a definition of [IOV_MAX].

EINVAL   One of the *iov_len* values in the msg_iov array member of the msghdr structure pointed to by *msg* is negative, or the sum of the *iov_len* values in the msg_iov array overflows a ssize_t.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   fcntl(2), ioctl(2), read(2), connect(3SOCKET), getsockopt(3SOCKET), libxnet(3LIB), select(3C), send(3SOCKET), socket(3SOCKET), socket.h(3HEAD), attributes(5)

Name recv – receive a message from a connected socket

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <sys/socket.h>

ssize_t recv(int *socket*, void **buffer*, size_t *length*, int *flags*);

Description The recv() function receives a message from a connection-mode or connectionless-mode
socket. It is normally used with connected sockets because it does not permit the application
to retrieve the source address of received data. The function takes the following arguments:

socket Specifies the socket file descriptor.

buffer Points to a buffer where the message should be stored.

length Specifies the length in bytes of the buffer pointed to by the *buffer* argument.

flags Specifies the type of message reception. Values of this argument are formed by
logically OR'ing zero or more of the following values:

MSG_PEEK Peeks at an incoming message. The data is treated as unread
and the next recv() or similar function will still return this
data.

MSG_OOB Requests out-of-band data. The significance and semantics of
out-of-band data are protocol-specific.

MSG_WAITALL Requests that the function block until the full amount of data
requested can be returned. The function may return a smaller
amount of data if a signal is caught, if the connection is
terminated, if MSG_PEEK was specified, or if an error is
pending for the socket.

The recv() function returns the length of the message written to the buffer pointed to by the
buffer argument. For message-based sockets such as SOCK_DGRAM and SOCK_SEQPACKET, the
entire message must be read in a single operation. If a message is too long to fit in the supplied
buffer, and MSG_PEEK is not set in the *flags* argument, the excess bytes are discarded. For
stream-based sockets such as SOCK_STREAM, message boundaries are ignored. In this case, data
is returned to the user as soon as it becomes available, and no data is discarded.

If the MSG_WAITALL flag is not set, data will be returned only up to the end of the first message.

If no messages are available at the socket and O_NONBLOCK is not set on the socket's file
descriptor, recv() blocks until a message arrives. If no messages are available at the socket and
O_NONBLOCK is set on the socket's file descriptor, recv() fails and sets errno to EAGAIN or
EWOULDBLOCK.

Usage The recv() function is identical to recvfrom(3XNET) with a zero *address_len* argument, and to read() if no flags are used.

The select(3C) and poll(2) functions can be used to determine when data is available to be received.

Return Values Upon successful completion, recv() returns the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, recv() returns 0. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The recv() function will fail if:

EAGAIN
EWOULDBLOCK   The socket's file descriptor is marked O_NONBLOCK and no data is waiting to be received; or MSG_OOB is set and no out-of-band data is available and either the socket's file descriptor is marked O_NONBLOCK or the socket does not support blocking to await out-of-band data.

EBADF   The *socket* argument is not a valid file descriptor.

ECONNRESET   A connection was forcibly closed by a peer.

EFAULT   The *buffer* parameter can not be accessed or written.

EINTR   The recv() function was interrupted by a signal that was caught, before any data was available.

EINVAL   The MSG_OOB flag is set and no out-of-band data is available.

ENOTCONN   A receive is attempted on a connection-mode socket that is not connected.

ENOTSOCK   The *socket* argument does not refer to a socket.

EOPNOTSUPP   The specified flags are not supported for this socket type or protocol.

ETIMEDOUT   The connection timed out during connection establishment, or due to a transmission timeout on active connection.

The recv() function may fail if:

EIO   An I/O error occurred while reading from or writing to the file system.

ENOBUFS   Insufficient resources were available in the system to perform the operation.

ENOMEM   Insufficient memory was available to fulfill the request.

ENOSR   There were insufficient STREAMS resources available for the operation to complete.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also poll(2), recvmsg(3XNET), recvfrom(3XNET), select(3C), send(3XNET), sendmsg(3XNET), sendto(3XNET), shutdown(3XNET), socket(3XNET), attributes(5), standards(5)

Name   recvfrom – receive a message from a socket

Synopsis   cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <sys/socket.h>

ssize_t recvfrom(int *socket*, void *restrict *buffer*, size_t *length*,
    int *flags*, struct sockaddr *restrict *address*,
    socklen_t *restrict *address_len*);

Description   The recvfrom() function receives a message from a connection-mode or
connectionless-mode socket. It is normally used with connectionless-mode sockets because it
permits the application to retrieve the source address of received data.

The function takes the following arguments:

socket   Specifies the socket file descriptor.

buffer   Points to the buffer where the message should be stored.

length   Specifies the length in bytes of the buffer pointed to by the *buffer* argument.

flags   Specifies the type of message reception. Values of this argument are formed
                by logically OR'ing zero or more of the following values:

    MSG_PEEK   Peeks at an incoming message. The data is treated as
                    unread and the next recvfrom() or similar function
                    will still return this data.

    MSG_OOB   Requests out-of-band data. The significance and
                    semantics of out-of-band data are protocol-specific.

    MSG_WAITALL   Requests that the function block until the full amount
                    of data requested can be returned. The function may
                    return a smaller amount of data if a signal is caught, if
                    the connection is terminated, if MSG_PEEK was
                    specified, or if an error is pending for the socket.

address   A null pointer, or points to a sockaddr structure in which the sending address
                is to be stored. The length and format of the address depend on the address
                family of the socket.

address_len   Specifies the length of the sockaddr structure pointed to by the *address*
                argument.

The recvfrom() function returns the length of the message written to the buffer pointed to by
the *buffer* argument. For message-based sockets such as SOCK_DGRAM and SOCK_SEQPACKET, the
entire message must be read in a single operation. If a message is too long to fit in the supplied
buffer, and MSG_PEEK is not set in the *flags* argument, the excess bytes are discarded. For
stream-based sockets such as SOCK_STREAM, message boundaries are ignored. In this case, data
is returned to the user as soon as it becomes available, and no data is discarded.

If the MSG_WAITALL flag is not set, data will be returned only up to the end of the first message.

Not all protocols provide the source address for messages. If the *address* argument is not a null pointer and the protocol provides the source address of messages, the source address of the received message is stored in the sockaddr structure pointed to by the *address* argument, and the length of this address is stored in the object pointed to by the *address_len* argument.

If the actual length of the address is greater than the length of the supplied sockaddr structure, the stored address will be truncated.

If the *address* argument is not a null pointer and the protocol does not provide the source address of messages, the the value stored in the object pointed to by *address* is unspecified.

If no messages are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, recvfrom() blocks until a message arrives. If no messages are available at the socket and O_NONBLOCK is set on the socket's file descriptor, recvfrom() fails and sets errno to EAGAIN or EWOULDBLOCK.

Usage   The select(3C) and poll(2) functions can be used to determine when data is available to be received.

Return Values   Upon successful completion, recvfrom() returns the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, recvfrom() returns 0. Otherwise the function returns −1 and sets errno to indicate the error.

Errors   The recvfrom() function will fail if:

EAGAIN

EWOULDBLOCK   The socket's file descriptor is marked O_NONBLOCK and no data is waiting to be received, or MSG_OOB is set and no out-of-band data is available and either the socket's file descriptor is marked O_NONBLOCK or the socket does not support blocking to await out-of-band data.

EBADF   The *socket* argument is not a valid file descriptor.

ECONNRESET   A connection was forcibly closed by a peer.

EFAULT   The *buffer*, *address* or *address_len* parameter can not be accessed or written.

EINTR   A signal interrupted recvfrom() before any data was available.

EINVAL   The MSG_OOB flag is set and no out-of-band data is available.

ENOTCONN   A receive is attempted on a connection-mode socket that is not connected.

ENOTSOCK   The *socket* argument does not refer to a socket.

EOPNOTSUPP   The specified flags are not supported for this socket type.

ETIMEDOUT   The connection timed out during connection establishment, or due to a
                   transmission timeout on active connection.

The recvfrom() function may fail if:

EIO   An I/O error occurred while reading from or writing to the file system.

ENOBUFS   Insufficient resources were available in the system to perform the operation.

ENOMEM   Insufficient memory was available to fulfill the request.

ENOSR   There were insufficient STREAMS resources available for the operation to
             complete.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also poll(2), recv(3XNET), recvmsg(3XNET), select(3C) send(3XNET), sendmsg(3XNET),
sendto(3XNET), shutdown(3XNET), socket(3XNET), attributes(5), standards(5)

Name recvmsg – receive a message from a socket

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <sys/socket.h>

ssize_t recvmsg(int *socket*, struct msghdr **message*, int *flags*);

Description The recvmsg() function receives a message from a connection-mode or connectionless-mode socket. It is normally used with connectionless-mode sockets because it permits the application to retrieve the source address of received data.

The recvmsg() function receives messages from unconnected or connected sockets and returns the length of the message.

The recvmsg() function returns the total length of the message. For message-based sockets such as SOCK_DGRAM and SOCK_SEQPACKET, the entire message must be read in a single operation. If a message is too long to fit in the supplied buffers, and MSG_PEEK is not set in the *flags* argument, the excess bytes are discarded, and MSG_TRUNC is set in the msg_flags member of the msghdr structure. For stream-based sockets such as SOCK_STREAM, message boundaries are ignored. In this case, data is returned to the user as soon as it becomes available, and no data is discarded.

If the MSG_WAITALL flag is not set, data will be returned only up to the end of the first message.

If no messages are available at the socket, and O_NONBLOCK is not set on the socket's file descriptor, recvmsg() blocks until a message arrives. If no messages are available at the socket and O_NONBLOCK is set on the socket's file descriptor, the recvmsg() function fails and sets errno to EAGAIN or EWOULDBLOCK.

In the msghdr structure, defined in socket.h(3HEAD), the msg_name and msg_namelen members specify the source address if the socket is unconnected. If the socket is connected, the msg_name and msg_namelen members are ignored. The msg_name member may be a null pointer if no names are desired or required.

The msg_control and msg_controllen members specify a buffer to receive ancillary data sent along with a message. Ancillary data consists of a sequence of pairs. Each pair is composed of a cmsghdr structure followed by a data array. The cmsghdr structure, defined in socket.h(3HEAD), contains descriptive information which allows an application to correctly parse data. The data array contains the ancillary data message.

If ancillary data is not transferred, msg_control is set to NULL and msg_controllen is set to 0.

The msg_iov and msg_iovlen fields of the msghdr structure are used to specify where the received data will be stored. msg_iov points to an array of iovec structures. The msg_iovlen must be set to the dimension of this array. In each iovec structure, the iov_base field specifies a storage area and the iov_len field gives its size in bytes. Each storage area indicated by msg_iov is filled with received data in turn until all of the received data is stored or all of the areas have been filled.

If the SO_TIMESTAMP option has been enabled through setsockopt(), then a struct timeval is returned following the cmsghdr, and the cmsg_len field of the cmsghdr indicates the size of the struct timeval.

On successful completion, the msg_flags member of the message header is the bitwise-inclusive OR of all of the following flags that indicate conditions detected for the received message:

MSG_EOR	End of record was received (if supported by the protocol).
MSG_OOB	Out-of-band data was received.
MSG_TRUNC	Normal data was truncated.
MSG_CTRUNC	Control data was truncated.

Parameters   The function takes the following arguments:

socket	Specifies the socket file descriptor.
message	Points to a msghdr structure, containing both the buffer to store the source address and the buffers for the incoming message. The length and format of the address depend on the address family of the socket. The msg_flags member is ignored on input, but may contain meaningful values on output.
flags	Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:

MSG_OOB	Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
MSG_PEEK	Peeks at the incoming message.
MSG_WAITALL	Requests that the function block until the full amount of data requested can be returned. The function may return a smaller amount of data if a signal is caught, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socket.

Usage   The select(3C) and poll(2) functions can be used to determine when data is available to be received.

Return Values   Upon successful completion, recvmsg() returns the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, recvmsg() returns 0. Otherwise, −1 is returned and errno is set to indicate the error.

Errors   The recvmsg() function will fail if:

EAGAIN

EWOULDBLOCK   The socket's file descriptor is marked O_NONBLOCK and no data is waiting to
                     be received; or MSG_OOB is set and no out-of-band data is available and either
                     the socket's file descriptor is marked O_NONBLOCK or the socket does not
                     support blocking to await out-of-band data.

EBADF   The *socket* argument is not a valid open file descriptor.

ECONNRESET   A connection was forcibly closed by a peer.

EFAULT   The *message* parameter, or storage pointed to by the *msg_name*,
                     msg_control or *msg_iov* fields of the *message* parameter, or storage pointed
                     to by the iovec structures pointed to by the *msg_iov* field can not be
                     accessed or written.

EINTR   This function was interrupted by a signal before any data was available.

EINVAL   The sum of the iov_len values overflows an ssize_t. or the MSG_OOB flag
                     is set and no out-of-band data is available.

EMSGSIZE   The msg_iovlen member of the msghdr structure pointed to by *message* is
                     less than or equal to 0, or is greater than IOV_MAX.

ENOTCONN   A receive is attempted on a connection-mode socket that is not connected.

ENOTSOCK   The *socket* argument does not refer to a socket.

EOPNOTSUPP   The specified flags are not supported for this socket type.

ETIMEDOUT   The connection timed out during connection establishment, or due to a
                     transmission timeout on active connection.

The recvmsg() function may fail if:

EIO   An IO error occurred while reading from or writing to the file system.

ENOBUFS   Insufficient resources were available in the system to perform the operation.

ENOMEM   Insufficient memory was available to fulfill the request.

ENOSR   There were insufficient STREAMS resources available for the operation to
             complete.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Standard	See standards(5).

See Also   poll(2), recv(3XNET), recvfrom(3XNET), select(3C), send(3XNET), sendmsg(3XNET), sendto(3XNET), setsockopt(3XNET), shutdown(3XNET), socket(3XNET), socket.h(3HEAD), attributes(5), standards(5)

Name   resolver, res_state, res_ninit, fp_resstat, res_hostalias, res_nquery, res_nsearch,
res_nquerydomain, res_nmkquery, res_nsend, res_nclose, res_nsendsigned, dn_comp,
dn_expand, hstrerror, res_init, res_query, res_search, res_mkquery, res_send, herror,
res_getservers, res_setservers, res_ndestroy – resolver interface

Synopsis   BIND 8.2.2 Interfaces

```
cc [ flag ... ] file ... -lresolv  -lsocket   -lnsl [ library ... ]
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>
#include <netdb.h>

typedef struct __res_state *res_state;

int res_ninit(res_state statp);

void res_ndestroy(res_state statp);

void fp_resstat(const res_state statp, FILE *fp);

const char *res_hostalias(const res_state statp, const char *name,
    char *buf, size_t buflen);

int res_nquery(res_state statp, const char *dname, int class, int type,
    uchar_t *answer, int anslen);

int res_nsearch(res_state statp, const char *dname, int class, int type,
    u_char *answer, int anslen);

int res_nquerydomain(res_state statp, const char *name,
    const char *domain, int class, int type, u_char *answer,
    int anslen);

int res_nmkquery(res_state statp, int op, const char *dname, int class,
    int type, const u_char *data, int datalen,
    const u_char *newrr_in, u_char *buf, int buflen);

int res_nsend(res_state statp, const u_char *msg, int msglen,
    u_char *answer, int anslen);

void res_nclose(res_state statp);

int res_nsendsigned(res_state statp, const uchar_t *msg,
    int msglen, ns_tsig_key *key, uchar_t *answer, int anslen);

int dn_comp(const char *exp_dn, u_char *comp_dn, int length,
    u_char **dnptrs, **lastdnptr);

int dn_expand(const u_char *msg, *eomorig, *comp_dn, char *exp_dn,
    int length);

const char *hstrerror(int err);
```

```
void res_setservers(res_state statp, const union res_sockaddr_union *set,
     int cnt);

int res_getservers(res_state statp, union res_sockaddr_union *set,
     int cnt);
```

Deprecated Interfaces

```
cc [ flag ... ] file ... -lresolv -lsocket  -lnsl [ library ... ]
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>
#include <netdb.h>


int res_init(void)

int res_query(const char *dname, int class, int type,
     u_char *answer, int anslen);

int res_search(const char *dname, int class, int type,
     u_char *answer, int anslen);

int res_mkquery(int op, const char *dname, int class,
     int type, const uchar_t *data, int datalen,
     const uchar_t *newrr, u_char *buf, int buflen);

int res_send(const u_char *msg, int msglen, u_char *answer,
     int anslen);

void herror(const char *s);
```

Description These routines are used for making, sending, and interpreting query and reply messages with Domain Name System (DNS) servers.

State information is kept in *statp* and is used to control the behavior of these functions. Set *statp* to all zeros prior to making the first call to any of these functions.

The res_ndestroy() function should be called to free memory allocated by res_ninit() after the last use of *statp*.

The functions res_init(), res_query(), res_search(), res_mkquery(), res_send(), and herror() are deprecated. They are supplied for backwards compatibility. They use global configuration and state information that is kept in the structure _res rather than state information referenced through *statp*.

Most of the values in *statp* and _res are initialized to reasonable defaults on the first call to res_ninit() or res_init() and can be ignored. Options stored in statp->options or _res.options are defined in <resolv.h>. They are stored as a simple bit mask containing the bitwise OR of the options enabled.

RES_INIT   True if the initial name server address and default domain name are initialized, that is, res_init() or res_ninit() has been called.

RES_DEBUG   Print debugging messages.

RES_AAONLY   Accept authoritative answers only. With this option, res_send() will
                   continue until it finds an authoritative answer or finds an error. Currently
                   this option is not implemented.

RES_USEVC   Use TCP connections for queries instead of UDP datagrams.

RES_STAYOPEN   Use with RES_USEVC to keep the TCP connection open between queries.
                   This is a useful option for programs that regularly do many queries. The
                   normal mode used should be UDP.

RES_IGNTC   Ignore truncation errors; that is, do not retry with TCP.

RES_RECURSE   Set the recursion-desired bit in queries. This is the default. res_send()
                   and res_nsend() do not do iterative queries and expect the name server
                   to handle recursion.

RES_DEFNAMES   If set, res_search() and res_nsearch() append the default domain
                   name to single-component names, that is, names that do not contain a
                   dot. This option is enabled by default.

RES_DNSRCH   If this option is set, res_search() and res_nsearch() search for host
                   names in the current domain and in parent domains. See hostname(1).
                   This option is used by the standard host lookup routine
                   gethostbyname(3NSL). This option is enabled by default.

RES_NOALIASES   This option turns off the user level aliasing feature controlled by the
                   HOSTALIASES environment variable. Network daemons should set this
                   option.

RES_BLAST   If the RES_BLAST option is defined, resolver() queries will be sent to all
                   servers. If the RES_BLAST option is not defined, but RES_ROTATE is , the list
                   of nameservers are rotated according to a round-robin scheme.
                   RES_BLAST overrides RES_ROTATE.

RES_ROTATE   This option causes res_nsend() and res_send() to rotate the list of
                   nameservers in statp->nsaddr_list or _res.nsaddr_list.

RES_KEEPTSIG   This option causes res_nsendsigned() to leave the message unchanged
                   after TSIG verification. Otherwise the TSIG record would be removed
                   and the header would be updated.

res_ninit(),   The res_ninit() and res_init() routines read the configuration file, if any is present, to get
res_init()   the default domain name, search list and the Internet address of the local name server(s). See
                resolv.conf(4). If no server is configured, res_init() or res_ninit() will try to obtain
                name resolution services from the host on which it is running. The current domain name is
                defined by domainname(1M), or by the hostname if it is not specified in the configuration file.
                Use the environment variable LOCALDOMAIN to override the domain name. This environment

variable may contain several blank-separated tokens if you wish to override the search list on a per-process basis. This is similar to the search command in the configuration file. You can set the RES_OPTIONS environment variable to override certain internal resolver options. You can otherwise set them by changing fields in the statp /_res structure. Alternatively, they are inherited from the configuration file's options command. See `resolv.conf(4)` for information regarding the syntax of the RES_OPTIONS environment variable. Initialization normally occurs on the first call to one of the other resolver routines.

res_nquery(),
res_query()
The res_nquery() and res_query() functions provide interfaces to the server query mechanism. They construct a query, send it to the local server, await a response, and make preliminary checks on the reply. The query requests information of the specified *type* and *class* for the specified fully-qualified domain name *dname*. The reply message is left in the *answer* buffer with length *anslen* supplied by the caller. res_nquery() and res_query() return the length of the *answer*, or -1 upon error.

The res_nquery() and res_query() routines return a length that may be bigger than *anslen*. In that case, retry the query with a larger *buf*. The *answer* to the second query may be larger still], so it is recommended that you supply a *buf* larger than the *answer* returned by the previous query. *answer* must be large enough to receive a maximum UDP response from the server or parts of the *answer* will be silently discarded. The default maximum UDP response size is 512 bytes.

res_nsearch(),
res_search()
The res_nsearch() and res_search() routines make a query and await a response, just like like res_nquery() and res_query(). In addition, they implement the default and search rules controlled by the RES_DEFNAMES and RES_DNSRCH options. They return the length of the first successful reply which is stored in *answer*. On error, they reurn –1.

The res_nsearch() and res_search() routines return a length that may be bigger than *anslen*. In that case, retry the query with a larger *buf*. The *answer* to the second query may be larger still], so it is recommended that you supply a *buf* larger than the *answer* returned by the previous query. *answer* must be large enough to receive a maximum UDP response from the server or parts of the *answer* will be silently discarded. The default maximum UDP response size is 512 bytes.

res_nquerydomain()
The res_nquerydomain() function calls res_query() on the concatenation of *name* and *domain*, removing a trailing dot from *name* if *domain* is NULL.

res_nmkquery(),
res_mkquery()
These routines are used by res_nquery() and res_query(). The res_nmkquery() and res_mkquery() functions construct a standard query message and place it in *buf*. The routine returns the *size* of the query, or -1 if the query is larger than *buflen*. The query type *op* is usually QUERY, but can be any of the query types defined in <arpa/nameser.h>. The domain name for the query is given by *dname*. *newrr* is currently unused but is intended for making update messages.

res_nsend(), res_send(), res_nsendsigned()	The res_nsend(), res_send(), and res_nsendsigned() routines send a pre-formatted query that returns an *answer*. The routine calls res_ninit() or res_init(). If RES_INIT is not set, the routine sends the query to the local name server and handles timeouts and retries. Additionally, the res_nsendsigned() uses TSIG signatures to add authentication to the query and verify the response. In this case, only one name server will be contacted. The routines return the length of the reply message, or -1 if there are errors.

The res_nsend() and res_send() routines return a length that may be bigger than *anslen*. In that case, retry the query with a larger *buf*. The *answer* to the second query may be larger still], so it is recommended that you supply a *buf* larger than the *answer* returned by the previous query. *answer* must be large enough to receive a maximum UDP response from the server or parts of the *answer* will be silently discarded. The default maximum UDP response size is 512 bytes. |
fp_resstat()	The function fp_resstat() prints out the active flag bits in *statp*->options preceded by the text ";; res options:" on *file*.
res_hostalias()	The function res_hostalias() looks up *name* in the file referred to by the HOSTALIASES environment variable and returns the fully qualified host name. If *name* is not found or an error occurs, NULL is returned. res_hostalias() stores the result in *buf*.
res_nclose()	The res_nclose() function closes any open files referenced through *statp*.
res_ndestroy()	The res_ndestroy() function calls res_nclose(), then frees any memory allocated by res_ninit() referenced through *statp*.
dn_comp()	The dn_comp() function compresses the domain name *exp_dn* and stores it in *comp_dn*. The dn_comp() function returns the size of the compressed name, or −1 if there were errors. *length* is the size of the array pointed to by *comp_dn*.

The *dnptrs* parameter is a pointer to the head of the list of pointers to previously compressed names in the current message. The first pointer must point to the beginning of the message. The list ends with NULL. The limit to the array is specified by *lastdnptr*.

A side effect of calling dn_comp() is to update the list of pointers for labels inserted into the message by dn_comp() as the name is compressed. If *dnptrs* is NULL, names are not compressed. If *lastdnptr* is NULL, dn_comp() does not update the list of labels. |
| dn_expand() | The dn_expand() function expands the compressed domain name *comp_dn* to a full domain name. The compressed name is contained in a query or reply message. *msg* is a pointer to the beginning of that message. The uncompressed name is placed in the buffer indicated by *exp_dn*, which is of size *length*. The dn_expand() function returns the size of the compressed name, or −1 if there was an error. |

hstrerror(),   The variables *statp->res_h_errno* and *_res.res_h_errno* and external variable *h_errno* are set
herror()   whenever an error occurs during a resolver operation. The following definitions are given in
   <netdb.h>:

```
#define NETDB_INTERNAL -1 /* see errno */
#define NETDB_SUCCESS  0  /* no problem */
#define HOST_NOT_FOUND 1  /* Authoritative Answer Host not found */
#define TRY_AGAIN      2  /* Non-Authoritative not found, or SERVFAIL */
#define NO_RECOVERY    3  /* Non-Recoverable: FORMERR, REFUSED, NOTIMP*/
#define NO_DATA        4  /* Valid name, no data for requested type */
```

The herror() function writes a message to the diagnostic output consisting of the string
parameters, the constant string ": ", and a message corresponding to the value of *h_errno*.

The hstrerror() function returns a string, which is the message text that corresponds to the
value of the *err* parameter.

res_setservers(),   The functions res_getservers() and res_setservers() are used to get and set the list of
res_getservers()   servers to be queried.

Examples   EXAMPLE 1   The following example shows basic usage of the resolver's API.

```
#include <resolv.h>
#include <string.h>

int main(int argc, char **argv)
{
    int len;
    struct __res_state statp;
    union msg {
        uchar_t buf[NS_MAXMSG];
        HEADER  h;
    } resbuf;

    /* Initialize resolver */
    memset(&statp, 0, sizeof(statp));
    if (res_ninit(&statp) < 0) {
        fprintf(stderr, "Can't initialize statp.\n");
        return (1);
    }

    /*
     * Turning on DEBUG mode keeps this example simple,
     * without need to output anything.
     */
    statp.options |= RES_DEBUG;

    /* Search for A type records */
```

EXAMPLE 1 The following example shows basic usage of the resolver's API. *(Continued)*

```
len = res_nsearch(&statp, "example.com", C_IN, T_A,
    resbuf.buf, NS_MAXMSG);
if (len < 0) {
    fprintf(stderr, "Error occured during search.\n");
    return (1);
}
if (len > NS_MAXMSG) {
    fprintf(stderr, "The buffer is too small.\n");
    return (1);
}

/* ... Process the received answer ... */

/* Cleanup */
res_ndestroy(&statp);
return (0);
}
```

Files /etc/resolv.conf   resolver configuration file

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library
Interface Stability	Committed
MT-Level	Unsafe for deprecated interfaces; MT-Safe for all others.

See Also domainname(1M), gethostbyname(3NSL), libresolv(3LIB), resolv.conf(4), attributes(5)

Lottor, M. *RFC 1033, Domain Administrators Operations Guide*. Network Working Group. November 1987.

Mockapetris, Paul. *RFC 1034, Domain Names - Concepts and Facilities*. Network Working Group. November 1987.

Mockapetris, Paul. *RFC 1035, Domain Names - Implementation and Specification*. Network Working Group. November 1987.

Partridge, Craig. *RFC 974, Mail Routing and the Domain System*. Network Working Group. January 1986.

Stahl, M. *RFC 1032, Domain Administrators Guide*. Network Working Group. November 1987.

Vixie, Paul, Dunlap, Kevin J., Karels, Michael J. *Name Server Operations Guide for BIND*. Internet Software Consortium, 1996.

Notes When the caller supplies a work buffer, for example the *answer* buffer argument to res_nsend() or res_send(), the buffer should be aligned on an eight byte boundary. Otherwise, an error such as a SIGBUS may result.

Name   rexec, rexec_af – return stream to a remote command

Synopsis   cc [*flag* ...] *file*... -lsocket -lnsl [*library*...]
#include <netdb.h>
#include <unistd.h>

int rexec(char **ahost, unsigned short *inport*, const char *user,
    const char *passwd, const char *cmd, int *fd2p);

int rexec_af(char **ahost, unsigned short *inport*, const char *user,
    const char *passwd, const char *cmd, int *fd2p, int *af*);

Description   The rexec() and rexec_af() functions look up the host *ahost* using
getaddrinfo(3SOCKET) and return −1 if the host does not exist. Otherwise *ahost* is set to the
standard name of the host. The username and password are used in remote host
authentication. When a username and password are not specified, the .netrc file in the user's
home directory is searched for the appropriate information. If the search fails, the user is
prompted for the information.

The rexec() function always returns a socket of the AF_INET address family. The rexec_af()
function supports AF_INET, AF_INET6, or AF_UNSPEC for the address family. An application
can choose which type of socket is returned by passing AF_INET or AF_INET6 as the address
family. The use of AF_UNSPEC means that the caller will accept any address family. Choosing
AF_UNSPEC provides a socket that best suits the connectivity to the remote host.

The port *inport* specifies which DARPA Internet port to use for the connection. The port
number used must be in network byte order, as supplied by a call to htons(3XNET). The
protocol for connection is described in detail in in.rexecd(1M).

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller, and given to the
remote command as its standard input and standard output. If *fd2p* is non-zero, an auxiliary
channel to a control process is set up and a file descriptor for it is placed in *fd2p. The control
process returns diagnostic output (file descriptor 2), from the command on the auxiliary
channel. The control process also accepts bytes on this channel as signal numbers to be
forwarded to the process group of the command. If *fd2p* is 0, the standard error (file descriptor
2) of the remote command is made the same as its standard output. No provision is made for
sending arbitrary signals to the remote process, other than possibly sending out-of-band data.

There is no way to specify options to the socket() call made by the rexec() or
rexec_af()functions.

Return Values   If rexec() succeeds, a file descriptor number is returned of the socket type SOCK_STREAM and
the address family AF_INET. The parameter *ahost is set to the standard name of the host. If the
value of *fd2p* is other than NULL, a file descriptor number is placed in *fd2p which represents
the standard error stream of the command.

If rexec_af() succeeds, the routine returns a file descriptor number of the socket type
SOCK_STREAM in the address family AF_INET or AF_INET6, as determined by the value of the *af*
parameter.

If either rexec() or rexec_af() fails, −1 is returned.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

This interface is Unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

See Also in.rexecd(1M), getaddrinfo(3SOCKET), gethostbyname(3NSL), getservbyname(3SOCKET), htonl(3XNET), socket(3SOCKET), attributes(5)

Name   rpc – library routines for remote procedure calls

Synopsis   cc [*flag* ...] *file* ... -lnsl [*library* ...]
```
#include <rpc/rpc.h>
#include <netconfig.h>
```

Description   These routines allow C language programs to make procedure calls on other machines across a network. First, the client sends a request to the server. On receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply.

All RPC routines require the header <rpc/rpc.h>. Routines that take a netconfig structure also require that <netconfig.h> be included. Applications using RPC and XDR routines should be linked with the libnsl library.

Multithread   In the case of multithreaded applications, the -mt option must be specified on the command
Considerations   line at compilation time to enable a thread-specific version of rpc_createerr(). See rpc_clnt_create(3NSL) and threads(5).

When used in multithreaded applications, client-side routines are MT-Safe. CLIENT handles can be shared between threads; however, in this implementation, requests by different threads are serialized (that is, the first request will receive its results before the second request is sent). See rpc_clnt_create(3NSL).

When used in multithreaded applications, server-side routines are usually Unsafe. In this implementation the service transport handle, SVCXPRT contains a single data area for decoding arguments and encoding results. See rpc_svc_create(3NSL). Therefore, this structure cannot be freely shared between threads that call functions that do this. Routines that are affected by this restriction are marked as unsafe for MT applications. See rpc_svc_calls(3NSL).

Nettyp   Some of the high-level RPC interface routines take a *nettype* string as one of the parameters (for example, clnt_create(), svc_create(), rpc_reg(), rpc_call()). This string defines a class of transports which can be used for a particular application.

nettype can be one of the following:

netpath   Choose from the transports which have been indicated by their token names in the NETPATH environment variable. If NETPATH is unset or NULL, it defaults to visible. netpath is the default *nettype*.

visible   Choose the transports which have the visible flag (v) set in the /etc/netconfig file.

circuit_v   This is same as visible except that it chooses only the connection oriented transports (semantics tpi_cots or tpi_cots_ord) from the entries in the /etc/netconfig file.

datagram_v   This is same as visible except that it chooses only the connectionless datagram transports (semantics tpi_clts) from the entries in the /etc/netconfig file.

circuit_n	This is same as netpath except that it chooses only the connection oriented datagram transports (semantics tpi_cots or tpi_cots_ord).	
datagram_n	This is same as netpath except that it chooses only the connectionless datagram transports (semantics tpi_clts).	
udp	This refers to Internet UDP.	
tcp	This refers to Internet TCP.	

If *nettype* is NULL, it defaults to netpath. The transports are tried in left to right order in the NETPATH variable or in top to down order in the /etc/netconfig file.

Derived Types   In a 64-bit environment, the derived types are defined as follows:

```
typedef                     uint32_t                 rpcprog_t;

typedef                     uint32_t                 rpcvers_t;

typedef                     uint32_t                 rpcproc_t;

typedef                     uint32_t                 rpcprot_t;

typedef                     uint32_t                 rpcport_t;

typedef                     int32_t                  rpc_inline_t;
```

In a 32-bit environment, the derived types are defined as follows:

```
typedef                     unsigned long            rpcprog_t;

typedef                     unsigned long            rpcvers_t;

typedef                     unsigned long            rpcproc_t;

typedef                     unsigned long            rpcprot_t;

typedef                     unsigned long            rpcport_t;

typedef                     long                     rpc_inline_t;
```

Data Structures   Some of the data structures used by the RPC package are shown below.

The AUTH Structure
```
union  des_block  {
        struct  {
        u_int32  high;
        u_int32  low;
        }  key;
char   c[8];
};
typedef  union  des_block  des_block;
```

```
                   extern  bool_t  xdr_des_block( );
                   /*
                    *  Authentication  info.  Opaque  to  client.
                    */
                   struct  opaque_auth  {
                           enum_t oa_flavor;       /* flavor of auth */
                           caddr_t oa_base;        /* address of more auth stuff */
                           uint_t oa_length;       /* not to exceed MAX_AUTH_BYTES */
                   };
                   /*
                    * Auth handle, interface to client side  authenticators.
                    */
                    typedef  struct  {
                           struct opaque_auth ah_cred;
                           struct opaque_auth ah_verf;
                           union  des_block ah_key;
                           struct  auth_ops  {
                                   void(*ah_nextverf)( );
                                   int(*ah_marshal)( );     /* nextverf & serialize */
                                   int(*ah_validate)( );    /* validate verifier */
                                   int(*ah_refresh)( );     /* refresh credentials */
                                   void(*ah_destroy)( );    /* destroy this structure */
                           }  *ah_ops;
                           caddr_t  ah_private;
                   }  AUTH;
```

The CLIENT Structure
```
                   /*
                    *  Client  rpc  handle.
                    *  Created  by  individual  implementations.
                    *  Client is responsible for initializing auth.
                    */
                         typedef  struct  {
                         AUTH    *cl_auth;       /* authenticator */
                         struct clnt_ops {
                             enum clnt_stat (*cl_call)( );   /* call remote procedure */
                             void (*cl_abort)( );            /* abort a call */
                             void (*cl_geterr)( );           /* get specific error code */
                             bool_t (*cl_freeres)( );        /* frees results */
                             void (*cl_destroy)( );          /* destroy this structure */
                             bool_t (*cl_control)( );        /* the ioctl( ) of rpc */
                             int (*cl_settimers)( );         /* set rpc level timers */
                             } *cl_ops;
                             caddr_t    cl_private;                /* private stuff */
                             char       *cl_netid;                /* network identifier */
                             char       *cl_tp;                   /* device name */
                   }  CLIENT;
```

The SVCXPRT Structure
```
enum  xprt_stat  {
XPRT_DIED,
XPRT_MOREREQS,
XPRT_IDLE
};
/*
 *  Server  side  transport  handle
 */
typedef  struct  {
        int      xp_fd;                    /* file descriptor for the
        ushort_t xp_port;                  /* obsolete */
        struct xp_ops {
           bool_t (*xp_recv)( ); /* receive incoming requests */
           enum xprt_stat (*xp_stat)( ); /* get transport status */
           bool_t (*xp_getargs)( );      /* get arguments */
           bool_t (*xp_reply)( );        /* send reply */
           bool_t (*xp_freeargs)( );     /* free mem allocated
                                               for args */
           void (*xp_destroy)( );        /* destroy this struct */
        } *xp_ops;
        int xp_addrlen;                    /* length of remote addr.
                                              Obsolete */
        char *xp_tp;                       /* transport provider device
                                              name */
        char *xp_netid;                    /* network identifier */
        struct netbuf xp_ltaddr;           /* local transport address */
        struct netbuf xp_rtaddr;           /* remote transport address */
        char xp_raddr[16];                 /* remote address. Obsolete */
        struct opaque_auth xp_verf;        /* raw response verifier */
        caddr_t xp_p1;                     /* private: for use
                                              by svc ops */
        caddr_t xp_p2;                     /* private: for use
                                              by svc ops */
        caddr_t xp_p3;                     /* private: for use
                                              by svc lib */
        int xp_type                        /* transport type */
} SVCXPRT;
```

The svc_reg Structure
```
struct  svc_req  {
        rpcprog_t rq_prog;         /* service program number */
        rpcvers_t rq_vers;         /* service protocol version */
        rpcproc_t rq_proc;         /* the desired procedure */
        struct opaque_auth rq_cred; /* raw creds from the wire */
        caddr_t rq_clntcred;       /* read only cooked cred */
        SVCXPRT *rq_xprt;          /* associated transport */

};
```

The XDR Structure

```
/*
 * XDR operations.
 * XDR_ENCODE causes the type to be encoded into the stream.
 * XDR_DECODE causes the type to be extracted from the stream.
 * XDR_FREE can be used to release the space allocated by an XDR_DECODE
 * request.
 */
enum xdr_op {
    XDR_ENCODE=0,
    XDR_DECODE=1,
    XDR_FREE=2
};
/*
 * This is the number of bytes per unit of external data.
 */
#define BYTES_PER_XDR_UNIT     (4)
#define RNDUP(x)  ((((x) + BYTES_PER_XDR_UNIT - 1) /
                    BYTES_PER_XDR_UNIT) \ * BYTES_PER_XDR_UNIT)
/*
 * A xdrproc_t exists for each data type which is to be encoded or
 * decoded.  The second argument to the xdrproc_t is a pointer to
 * an opaque pointer.  The opaque pointer generally points to a
 * structure of the data type to be decoded.  If this points to 0,
 * then the type routines should allocate dynamic storage of the
 * appropriate size and return it.
 * bool_t  (*xdrproc_t)(XDR *, caddr_t *);
 */
typedef  bool_t (*xdrproc_t)( );
/*
 * The XDR handle.
 * Contains operation which is being applied to the stream,
 * an operations vector for the particular implementation
 */
typedef struct {

enum xdr_op x_op;     /* operation; fast additional param */
struct  xdr_ops  {

bool_t      (*x_getlong)( );     /* get long from underlying stream */
bool_t      (*x_putlong)( );     /* put long to underlying stream */
bool_t      (*x_getbytes)( );    /* get bytes from underlying stream */
bool_t      (*x_putbytes)( );    /* put bytes to underlying stream */
uint_t      (*x_getpostn)( );    /* returns bytes off from beginning */
bool_t      (*x_setpostn)( );    /* reposition the stream */
rpc_inline_t *(*x_inline)( );    /* buf quick ptr to buffered data */
void        (*x_destroy)( );     /* free privates of this xdr_stream */
bool_t      (*x_control)( );     /* changed/retrieve client object info*/
bool_t      (*x_getint32)( );    /* get int from underlying stream */
```

```
bool_t      (*x_putint32)( );    /* put int to underlying stream */

} *x_ops;

caddr_t     x_public;            /* users' data */
caddr_t     x_priv               /* pointer to private data */
caddr_t     x_base;              /* private used for position info */
int         x_handy;             /* extra private word */
XDR;
```

Index to Routines   The following table lists RPC routines and the manual reference pages on which they are
described:

RPC Routine	Manual Reference Page
auth_destroy	rpc_clnt_auth(3NSL)
authdes_create	rpc_soc(3NSL)
authdes_getucred	secure_rpc(3NSL)
authdes_seccreate	secure_rpc(3NSL)
authnone_create	rpc_clnt_auth(3NSL)
authsys_create	rpc_clnt_auth(3NSL)
authsys_create_default	rpc_clnt_auth(3NSL)
authunix_create	rpc_soc(3NSL)
authunix_create_default	rpc_soc(3NSL)
callrpc	rpc_soc(3NSL)
clnt_broadcast	rpc_soc(3NSL)
clnt_call	rpc_clnt_calls(3NSL)
clnt_control	rpc_clnt_create(3NSL)
clnt_create	rpc_clnt_create(3NSL)
clnt_destroy	rpc_clnt_create(3NSL)
clnt_dg_create	rpc_clnt_create(3NSL)
clnt_freeres	rpc_clnt_calls(3NSL)
clnt_geterr	rpc_clnt_calls(3NSL)
clnt_pcreateerror	rpc_clnt_create(3NSL)
clnt_perrno	rpc_clnt_calls(3NSL)
clnt_perror	rpc_clnt_calls(3NSL)

clnt_raw_create	rpc_clnt_create(3NSL)
clnt_spcreateerror	rpc_clnt_create(3NSL)
clnt_sperrno	rpc_clnt_calls(3NSL)
clnt_sperror	rpc_clnt_calls(3NSL)
clnt_tli_create	rpc_clnt_create(3NSL)
clnt_tp_create	rpc_clnt_create(3NSL)
clnt_udpcreate	rpc_soc(3NSL)
clnt_vc_create	rpc_clnt_create(3NSL)
clntraw_create	rpc_soc(3NSL)
clnttcp_create	rpc_soc(3NSL)
clntudp_bufcreate	rpc_soc(3NSL)
get_myaddress	rpc_soc(3NSL)
getnetname	secure_rpc(3NSL)
host2netname	secure_rpc(3NSL)
key_decryptsession	secure_rpc(3NSL)
key_encryptsession	secure_rpc(3NSL)
key_gendes	secure_rpc(3NSL)
key_setsecret	secure_rpc(3NSL)
netname2host	secure_rpc(3NSL)
netname2user	secure_rpc(3NSL)
pmap_getmaps	rpc_soc(3NSL)
pmap_getport	rpc_soc(3NSL)
pmap_rmtcall	rpc_soc(3NSL)
pmap_set	rpc_soc(3NSL)
pmap_unset	rpc_soc(3NSL)
registerrpc	rpc_soc(3NSL)
rpc_broadcast	rpc_clnt_calls(3NSL)
rpc_broadcast_exp	rpc_clnt_calls(3NSL)
rpc_call	rpc_clnt_calls(3NSL)

rpc_reg	rpc_svc_calls(3NSL)
svc_create	rpc_svc_create(3NSL)
svc_destroy	rpc_svc_create(3NSL)
svc_dg_create	rpc_svc_create(3NSL)
svc_dg_enablecache	rpc_svc_calls(3NSL)
svc_fd_create	rpc_svc_create(3NSL)
svc_fds	rpc_soc(3NSL)
svc_freeargs	rpc_svc_reg(3NSL)
svc_getargs	rpc_svc_reg(3NSL)
svc_getcaller	rpc_soc(3NSL)
svc_getreq	rpc_soc(3NSL)
svc_getreqset	rpc_svc_calls(3NSL)
svc_getrpccaller	rpc_svc_calls(3NSL)
svc_raw_create	rpc_svc_create(3NSL)
svc_reg	rpc_svc_calls(3NSL)
svc_register	rpc_soc(3NSL)
svc_run	rpc_svc_reg(3NSL)
svc_sendreply	rpc_svc_reg(3NSL)
svc_tli_create	rpc_svc_create(3NSL)
svc_tp_create	rpc_svc_create(3NSL)
svc_unreg	rpc_svc_calls(3NSL)
svc_unregister	rpc_soc(3NSL)
svc_vc_create	rpc_svc_create(3NSL)
svcerr_auth	rpc_svc_err(3NSL)
svcerr_decode	rpc_svc_err(3NSL)
svcerr_noproc	rpc_svc_err(3NSL)
svcerr_noprog	rpc_svc_err(3NSL)
svcerr_progvers	rpc_svc_err(3NSL)
svcerr_systemerr	rpc_svc_err(3NSL)

svcerr_weakauth	rpc_svc_err(3NSL)
svcfd_create	rpc_soc(3NSL)
svcraw_create	rpc_soc(3NSL)
svctcp_create	rpc_soc(3NSL)
svcudp_bufcreate	rpc_soc(3NSL)
svcudp_create	rpc_soc(3NSL)
user2netname	secure_rpc(3NSL)
xdr_accepted_reply	rpc_xdr(3NSL)
xdr_authsys_parms	rpc_xdr(3NSL)
xdr_authunix_parms	rpc_soc(3NSL)
xdr_callhdr	rpc_xdr(3NSL)
xdr_callmsg	rpc_xdr(3NSL)
xdr_opaque_auth	rpc_xdr(3NSL)
xdr_rejected_reply	rpc_xdr(3NSL)
xdr_replymsg	rpc_xdr(3NSL)
xprt_register	rpc_svc_calls(3NSL)
xprt_unregister	rpc_svc_calls(3NSL)

Files /etc/netconfig

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe with exceptions

See Also getnetconfig(3NSL), getnetpath(3NSL), rpc_clnt_auth(3NSL), rpc_clnt_calls(3NSL), rpc_clnt_create(3NSL), rpc_svc_calls(3NSL), rpc_svc_create(3NSL), rpc_svc_err(3NSL), rpc_svc_reg(3NSL), rpc_xdr(3NSL), rpcbind(3NSL), secure_rpc(3NSL), threads(5), xdr(3NSL), netconfig(4), rpc(4), attributes(5), environ(5)

Name   rpcbind, rpcb_getmaps, rpcb_getaddr, rpcb_gettime, rpcb_rmtcall, rpcb_set, rpcb_unset –
library routines for RPC bind service

Synopsis   #include <rpc/rpc.h>

```
struct rpcblist *rpcb_getmaps(const struct netconfig *nnetconf,
    const char *host);
```

```
bool_t  rpcb_getaddr(const rpcprog_t   prognum, const rpcvers_t versnum,
    const struct netconfig *netconf, struct netbuf *ssvcaddr,
    const char *host);
```

```
bool_t rpcb_gettime(const char *host, time_t *timep);
```

```
enum clnt_stat rpcb_rmtcall(const struct netconfig *netconf,
    const char *host, const rpcprog_t prognum,
    const rpcvers_t versnum, const rpcproc_t procnum,
    const xdrproc_t inproc, const caddr_t in,
    const xdrproc_t outproc caddr_t out,,
    const struct timeval tout, struct netbuf  *svcaddr);
```

```
bool_t rpcb_set(const rpcprog_t prognum, const rpcvers_t versnum,
    const struct netconfig *netconf, const struct netbuf *svcaddr);
```

```
bool_t rpcb_unset(const rpcprog_t prognum, const rpcvers_t versnum,
    const struct netconfig *netconf);
```

Description   These routines allow client C programs to make procedure calls to the RPC binder service.
rpcbind maintains a list of mappings between programs and their universal addresses. See
rpcbind(1M).

Routines   rpcb_getmaps()   An interface to the rpcbind service, which returns a list of the current
RPC program-to-address mappings on *host*. It uses the transport
specified through *netconf* to contact the remote rpcbind service on *host*.
This routine will return NULL, if the remote rpcbind could not be
contacted.

rpcb_getaddr()   An interface to the rpcbind service, which finds the address of the
service on *host* that is registered with program number *prognum*,
version *versnum*, and speaks the transport protocol associated with
netconf. The address found is returned in *svcaddr*. *svcaddr* should be
preallocated. This routine returns TRUE if it succeeds. A return value of
FALSE means that the mapping does not exist or that the RPC system
failed to contact the remote rpcbind service. In the latter case, the global
variable rpc_createerr contains the RPC status. See
rpc_clnt_create(3NSL).

rpcb_gettime()   This routine returns the time on *host* in *timep*. If *host* is NULL,
rpcb_gettime() returns the time on its own machine. This routine
returns TRUE if it succeeds, FALSE if it fails. rpcb_gettime() can be used

	to synchronize the time between the client and the remote server. This routine is particularly useful for secure RPC.
rpcb_rmtcall()	An interface to the rpcbind service, which instructs rpcbind on *host* to make an RPC call on your behalf to a procedure on that host. The netconfig structure should correspond to a connectionless transport. The parameter *svcaddr* will be modified to the server's address if the procedure succeeds. See rpc_call() and clnt_call() in rpc_clnt_calls(3NSL) for the definitions of other parameters.

This procedure should normally be used for a "ping" and nothing else. This routine allows programs to do lookup and call, all in one step.

Note: Even if the server is not running rpcbind does not return any error messages to the caller. In such a case, the caller times out.

Note: rpcb_rmtcall() is only available for connectionless transports.

rpcb_set()
An interface to the rpcbind service, which establishes a mapping between the triple [*prognum*, *versnum*, *netconf->nc_netid]* and *svcaddr* on the machine's rpcbind service. The value of *nc_netid* must correspond to a network identifier that is defined by the netconfig database. This routine returns TRUE if it succeeds, FALSE otherwise. See also svc_reg() in rpc_svc_calls(3NSL). If there already exists such an entry with rpcbind, rpcb_set() will fail.

rpcb_unset()
An interface to the rpcbind service, which destroys the mapping between the triple [*prognum*, *versnum*, *netconf->nc_netid]* and the address on the machine's rpcbind service. If *netconf* is NULL, rpcb_unset() destroys all mapping between the triple [*prognum*, *versnum*, *all-transports*] and the addresses on the machine's rpcbind service. This routine returns TRUE if it succeeds, FALSE otherwise. Only the owner of the service or the super-user can destroy the mapping. See also svc_unreg() in rpc_svc_calls(3NSL).

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also   rpcbind(1M), rpcinfo(1M), rpc_clnt_calls(3NSL), rpc_clnt_create(3NSL), rpc_svc_calls(3NSL), attributes(5)

Name rpc_clnt_auth, auth_destroy, authnone_create, authsys_create, authsys_create_default –
library routines for client side remote procedure call authentication

Synopsis `void auth_destroy(AUTH *`*auth*`);`

`AUTH *authnone_create    (void)`

`AUTH *authsys_create(const cha r*`*host*`,  const uid_t `*uid*`, const gid_t `*gid*`,`
`    const int `*len*`, const gid_t *`*aup_gids*`);`

`AUTH *authsys_create_default(void)`

Description These routines are part of the RPC library that allows C language programs to make procedure
calls on other machines across the network, with desired authentication.

These routines are normally called after creating the `CLIENT` handle. The `cl_auth` field of the
`CLIENT` structure should be initialized by the `AUTH` structure returned by some of the following
routines. The client's authentication information is passed to the server when the RPC call is
made.

Only the `NULL` and the `SYS` style of authentication is discussed here. For the `DES` style
authentication, please refer to secure_rpc(3NSL).

The `NULL` and `SYS` style of authentication are safe in multithreaded applications. For the
MT-level of the `DES` style, see its pages.

Routines The following routines require that the header `<rpc/rpc.h>` be included (see rpc(3NSL) for
the definition of the `AUTH` data structure).

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

auth_destroy()   A function macro that destroys the authentication information
                    associated with *auth*. Destruction usually involves deallocation of
                    private data structures. The use of *auth* is undefined after calling
                    `auth_destroy()`.

authnone_create()   Create and return an RPC authentication handle that passes
                    nonusable authentication information with each remote
                    procedure call. This is the default authentication used by RPC.

authsys_create()   Create and return an RPC authentication handle that contains
                    `AUTH_SYS` authentication information. The parameter *host* is the
                    name of the machine on which the information was created; *uid* is
                    the user's user ID; *gid* is the user's current group ID; *len* and
                    aup_gids refer to a counted array of groups to which the user
                    belongs.

authsys_create_default   Call `authsys_create()` with the appropriate parameters.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also rpc(3NSL), rpc_clnt_calls(3NSL), rpc_clnt_create(3NSL), secure_rpc(3NSL), attributes(5)

Name rpc_clnt_calls, clnt_call, clnt_send, clnt_freeres, clnt_geterr, clnt_perrno, clnt_perror, clnt_sperrno, clnt_sperror, rpc_broadcast, rpc_broadcast_exp, rpc_call – library routines for client side calls

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

```
enum  clnt_stat clnt_call(CLIENT *clnt, const rpcproc_t procnum,
    const xdrproc_t inproc, const caddr_t in, const xdrproc_t outproc,
    caddr_t out, const struct timeval tout);

enum clnt_stat clnt_send (CLIENT *clnt, const u_long
    procnum, const xdrproc_t proc, const caddr_t in);

bool_t clnt_freeres(CLIENT *clnt, const xdrproc_t outproc,
    caddr_t out);

void clnt_geterr(const CLIENT *clnt, struct rpc_err *errp);

void clnt_perrno(const enum clnt_stat stat);

void clnt_perror(const CLIENT *clnt, const char *s);

char *clnt_sperrno(const enum clnt_stat stat);

char *clnt_sperror(const CLIENT *clnt, const char *s);

enum clnt_stat rpc_broadcast(const rpcprog_t prognum,
    const rpcvers_t versnum, const rpcproc_t procnum,
    const xdrproc_t inproc, const caddr_t in,
    const xdrproc_t outproc, caddr_t out,
    const resultproc_t eachresult, const char *nettype);

enum clnt_stat rpc_broadcast_exp(const rpcprog_t prognum,
    const rpcvers_t versnum, const rpcproc_t procnum,
    const xdrproc_t xargs, caddr_t argsp, const xdrproc_t xresults,
    caddr_t resultsp, const resultproc_t eachresult, const int inittime,
    const int waittime, const char *nettype);

enum clnt_stat rpc_call(const char *host, const rpcprog_t prognum,
    const rpcvers_t versnum, const rpcproc_t procnum, const xdrproc_t inproc,
    const char *in, const xdrproc_t outproc, char *out, const char *nettype);
```

Description RPC library routines allow C language programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a request to the server. Upon receipt of the request, the server calls a dispatch routine to perform the requested service and then sends back a reply.

The `clnt_call()`, `rpc_call()`, and `rpc_broadcast()` routines handle the client side of the procedure call. The remaining routines deal with error handling.

Some of the routines take a CLIENT handle as one of the parameters. A CLIENT handle can be created by an RPC creation routine such as `clnt_create()`. See `rpc_clnt_create(3NSL)`.

These routines are safe for use in multithreaded applications. CLIENT handles can be shared between threads; however, in this implementation requests by different threads are serialized. In other words, the first request will receive its results before the second request is sent.

Routines   See rpc(3NSL) for the definition of the CLIENT data structure.

clnt_call()   A function macro that calls the remote procedure *procnum* associated with the client handle, *clnt*, which is obtained with an RPC client creation routine such as clnt_create(). See rpc_clnt_create(3NSL). The parameter *inproc* is the XDR function used to encode the procedure's parameters, and *outproc* is the XDR function used to decode the procedure's results. *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s). *tout* is the time allowed for results to be returned, which is overridden by a time-out set explicitly through clnt_control(). See rpc_clnt_create(3NSL).

If the remote call succeeds, the status returned is RPC_SUCCESS. Otherwise, an appropriate status is returned.

clnt_send()   Use the clnt_send() function to call a remote asynchronous function.

The clnt_send() function calls the remote function procnum() associated with the client handle, *clnt*, which is obtained with an RPC client creation routine such as clnt_create(). See rpc_clnt_create(3NSL). The parameter *proc* is the XDR function used to encode the procedure's parameters. The parameter *in* is the address of the procedure's argument(s).

By default, the blocking I/O mode is used. See the clnt_control(3NSL) man page for more information on I/O modes.

The clnt_send() function does not check if the program version number supplied to clnt_create() is registered with the rpcbind service. Use clnt_create_vers() instead of clnt_create() to check on incorrect version number registration. clnt_create_vers() will return a valid handle to the client only if a version within the range supplied to clnt_create_vers() is supported by the server.

RPC_SUCCESS is returned when a request is successfully delivered to the transport layer. This does not mean that the request was received. If an error is returned, use the clnt_geterr() routine

	to find the failure status or the clnt_perrno() routine to translate the failure status into error messages.
clnt_freeres()	A function macro that frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call. The parameter *out* is the address of the results, and *outproc* is the XDR routine describing the results. This routine returns 1 if the results were successfully freed; otherwise it returns 0.
clnt_geterr()	A function macro that copies the error structure out of the client handle to the structure at address *errp*.
clnt_perrno()	Prints a message to standard error corresponding to the condition indicated by *stat*. A newline is appended. It is normally used after a procedure call fails for a routine for which a client handle is not needed, for instance rpc_call()
clnt_perror()	Prints a message to the standard error indicating why an RPC call failed; *clnt* is the handle used to do the call. The message is prepended with string *s* and a colon. A newline is appended. This routine is normally used after a remote procedure call fails for a routine that requires a client handle, for instance clnt_call().
clnt_sperrno()	Takes the same arguments as clnt_perrno(), but instead of sending a message to the standard error indicating why an RPC call failed, returns a pointer to a string that contains the message.
	clnt_sperrno() is normally used instead of clnt_perrno() when the program does not have a standard error, as a program running as a server quite likely does not. clnt_sperrno() is also used if the programmer does not want the message to be output with printf(), or if a message format different than that supported by clnt_perrno() is to be used. See printf(3C). Unlike clnt_sperror() and clnt_spcreaterror(), clnt_sperrno() does not return a pointer to static data. Therefore, the result is not overwritten on each call. See rpc_clnt_create(3NSL).
clnt_sperror()	Similar to clnt_perror(), except that like clnt_sperrno(), it returns a string instead of printing to standard error. However, clnt_sperror() does not append a newline at the end of the message.
	clnt_sperror() returns a pointer to a buffer that is overwritten on each call. In multithreaded applications, this buffer is implemented as thread-specific data.

| rpc_broadcast() | Similar to rpc_call(), except that the call message is broadcast to all the connectionless transports specified by *nettype*. If *nettype* is NULL, it defaults to netpath. Each time it receives a response, this routine calls eachresult(), whose form is: |

```
bool_t eachresult(caddr_t out,  const struct netbuf *addr,
const struct netconfig *netconf);
```

where *out* is the same as *out* passed to rpc_broadcast(), except that the remote procedure's output is decoded there. *addr* points to the address of the machine that sent the results, and *netconf* is the netconfig structure of the transport on which the remote server responded. If eachresult() returns 0, rpc_broadcast() waits for more replies; otherwise, it returns with appropriate status.

The broadcast file descriptors are limited in size to the maximum transfer size of that transport. For Ethernet, this value is 1500 bytes. rpc_broadcast() uses AUTH_SYS credentials by default. See rpc_clnt_auth(3NSL).

| rpc_broadcast_exp() | Similar to rpc_broadcast(), except that the initial timeout, *inittime* and the maximum timeout, *waittime*, are specified in milliseconds. |

inittime is the initial time that rpc_broadcast_exp() waits before resending the request. After the first resend, the retransmission interval increases exponentially until it exceeds *waittime*.

| rpc_call() | Calls the remote procedure associated with *prognum*, *versnum*, and *procnum* on the machine, *host*. The parameter *inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results. *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s). *nettype* can be any of the values listed on rpc(3NSL). This routine returns RPC_SUCCESS if it succeeds, or it returns an appropriate status. Use the clnt_perrno() routine to translate failure status into error messages. |

The rpc_call() function uses the first available transport belonging to the class *nettype* on which it can create a connection. You do not have control of timeouts or authentication using this routine.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	All
Availability	system/library
Interface Stability	Committed
MT-Level	MT-Safe

See Also printf(3C), rpc(3NSL), rpc_clnt_auth(3NSL), rpc_clnt_create(3NSL), attributes(5)

Name rpc_clnt_create, clnt_control, clnt_create, clnt_create_timed, clnt_create_vers, clnt_create_vers_timed, clnt_destroy, clnt_dg_create, clnt_pcreateerror, clnt_raw_create, clnt_spcreateerror, clnt_tli_create, clnt_tp_create, clnt_tp_create_timed, clnt_vc_create, rpc_createerr, clnt_door_create – library routines for dealing with creation and manipulation of CLIENT handles

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

```
bool_t clnt_control(CLIENT *clnt, const uint_t req, char *info);
```

```
CLIENT *clnt_create(const char *host, const rpcprog_t prognum,
    const rpcvers_t versnum, const char *nettype);
```

```
CLIENT *clnt_create_timed(const char *host, const rpcprog_t prognum,
    const rpcvers_t versnum, const nettype,
    const struct timeval *timetout);
```

```
CLIENT *clnt_create_vers (const char *host,
    const rpcprog_t prognum, rpcvers_t *vers_outp,
    const rpcvers_t vers_low, const rpcvers_t vers_high,
    const char *nettype);
```

```
CLIENT *clnt_create_vers_timed(const char *host,
    const rpcprog_t prognum, rpcvers_t *vers_outp,
    const rpcvers_t vers_low, const rpcvers_t vers_high,
    char *nettype, const struct timeval *timeout);
```

```
void clnt_destroy(CLIENT *clnt);
```

```
CLIENT *clnt_dg_create(const int fildes,
    const struct netbuf *svcaddr, const rpcprog_t prognum,
    const rpcvers_t versnum, const uint_t sendsz,
    const uint_t recsz);
```

```
void clnt_pcreateerror(const char *s);
```

```
CLIENT *clnt_raw_create(const rpcprog_t prognum,
    const rpcvers_t versnum);
```

```
char *clnt_spcreateerror(const char *s);
```

```
CLIENT *clnt_tli_create(const int fildes,
    const struct netconfig *netconf, const struct netbuf *svcaddr,
    const rpcprog_t prognum, const rpcvers_t versnum,
    const uint_t sendsz, const uint_t recsz);
```

```
CLIENT *clnt_tp_create(const char *host,
    const rpcprog_t prognum, const rpcvers_t versnum,
    const struct netconfig *netconf);
```

```
CLIENT *clnt_tp_create_timed(const char *host,
    const rpcprog_t prognum, const rpcvers_t versnum,
    const struct netconfig *netconf, const struct timeval *timeout);
```

```
CLIENT *clnt_vc_create(const int fildes,
    const struct netbuf *svcaddr, const rpcprog_t prognum,
    const rpcvers_t versnum, const uint_t sendsz,
    const uint_t recsz);

struct rpc_createerr rpc_createerr

CLIENT *clnt_door_create(const rpcprog_t prognum,
    const rpcvers_t versnum, const uint_t sendsz);
```

Description RPC library routines allow C language programs to make procedure calls on other machines across the network. First a CLIENT handle is created and then the client calls a procedure to send a request to the server. On receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends a reply.

These routines are MT-Safe. In the case of multithreaded applications, the -mt option must be specified on the command line at compilation time. When the -mt option is specified, rpc_createerr becomes a macro that enables each thread to have its own rpc_createerr. See threads(5).

Routines See rpc(3NSL) for the definition of the CLIENT data structure.

clnt_control()
    A function macro to change or retrieve various information about a client object. *req* indicates the type of operation, and *info* is a pointer to the information. For both connectionless and connection-oriented transports, the supported values of *req* and their argument types and what they do are:

```
CLSET_TIMEOUT struct timeval * set total timeout
CLGET_TIMEOUT    struct timeval *    get total timeout
```

If the timeout is set using clnt_control(), the timeout argument passed by clnt_call() is ignored in all subsequent calls. If the timeout value is set to 0, clnt_control() immediately returns RPC_TIMEDOUT. Set the timeout parameter to 0 for batching calls.

```
CLGET_SERVER_ADDR struct netbuf * get server's address
CLGET_SVC_ADDR    struct netbuf *    get server's address
CLGET_FD     int *    get associated file descriptor
CLSET_FD_CLOSE    void    close the file descriptor when
        destroying the client handle
        (see clnt_destroy())
CLSET_FD_NCLOSE    void    do not close the file
        descriptor when destroying the client handle
CLGET_VERS    rpcvers_t get the RPC program's version
        number associated with the
        client handle
CLSET_VERS    rpcvers_t set the RPC program's version
        number associated with the
        client handle.  This assumes
        that the RPC server for this
```

```
                new version is still listening
                at the address of the previous
                version.
     CLGET_XID    uint32_t    get the XID of the previous
                remote procedure call
     CLSET_XID    uint32_t    set the XID of the next
                remote procedure call
     CLGET_PROG   rpcprog_t   get program number
     CLSET_PROG   rpcprog_t   set program number
```

The following operations are valid for connection-oriented transports only:

```
CLSET_IO_MODE rpciomode_t*    set the IO mode used
        to send one-way requests. The argument for this operation
        can be either:
   - RPC_CL_BLOCKING    all sending operations block
        until the underlying transport protocol has
        accepted requests. If you specify this argument
        you cannot use flush and getting and setting buffer
        size is meaningless.
   - RPC_CL_NONBLOCKING   sending operations do not
        block and return as soon as requests enter the buffer.
        You can now use non-blocking I/O. The requests in the
        buffer are pending. The requests are sent to
        the server as soon as a two-way request is sent
        or a flush is done. You are responsible for flushing
        the buffer. When you choose RPC_CL_NONBLOCKING argument
         you have a choice of flush modes as specified by
        CLSET_FLUSH_MODE.
CLGET_IO_MODE rpciomode_t*        get the current IO mode
CLSET_FLUSH_MODE rpcflushmode_t*        set the flush mode.
         The flush mode can only be used in non-blocking I/O mode.
         The argument can be either of the following:
   - RPC_CL_BESTEFFORT_FLUSH: All flushes send requests
         in the buffer until the transport end-point blocks.
         If the transport connection is congested, the call
         returns directly.
   - RPC_CL_BLOCKING_FLUSH: Flush blocks until the
         underlying transport protocol accepts all pending
         requests into the queue.
CLGET_FLUSH_MODE rpcflushmode_t*     get the current flush mode.
CLFLUSH rpcflushmode_t          flush the pending requests.
        This command can only be used in non-blocking I/O mode.
        The flush policy depends on which of the following
        parameters is specified:
   - RPC_CL_DEFAULT_FLUSH, or NULL:    The flush is done
        according to the current flush mode policy
        (see CLSET_FLUSH_MODE option).
   - RPC_CL_BESTEFFORT_FLUSH:        The flush tries
```

```
                            to send pending requests without blocking; the call
                            returns directly. If the transport connection is
                            congested, this call could return without the request
                            being sent.
                      - RPC_CL_BLOCKING_FLUSH:        The flush sends all pending
                            requests. This call will block until all the requests
                            have been accepted by the transport layer.
        CLSET_CONNMAXREC_SIZE int*     set the buffer size.
                             It is not possible to dynamically
                            resize the buffer if it contains data.
                            The default size of the buffer is 16 kilobytes.
        CLGET_CONNMAXREC_SIZE int*              get the current size of the
                            buffer
        CLGET_CURRENT_REC_SIZE int*          get the size of
                            the pending requests stored in the buffer. Use of this
                            command is only recommended when you are in non-blocking
                            I/O mode. The current size of the buffer is always zero
                            when the handle is in blocking mode as the buffer is not
                            used in this mode.
```

The following operations are valid for connectionless transports only:

```
CLSET_RETRY_TIMEOUT  struct timeval *    set the retry timeout
CLGET_RETRY_TIMEOUT  struct timeval *    get the retry timeout
```

The retry timeout is the time that RPC waits for the server to reply before retransmitting the request.

clnt_control() returns TRUE on success and FALSE on failure.

clnt_create()

Generic client creation routine for program *prognum* and version *versnum*. *host* identifies the name of the remote host where the server is located. *nettype* indicates the class of transport protocol to use. The transports are tried in left to right order in NETPATH variable or in top to bottom order in the netconfig database.

clnt_create() tries all the transports of the *nettype* class available from the NETPATH environment variable and the netconfig database, and chooses the first successful one. A default timeout is set and can be modified using clnt_control(). This routine returns NULL if it fails. The clnt_pcreateerror() routine can be used to print the reason for failure.

Note that clnt_create() returns a valid client handle even if the particular version number supplied to clnt_create() is not registered with the rpcbind service. This mismatch will be discovered by a clnt_call later (see rpc_clnt_calls(3NSL)).

clnt_create_timed()
Generic client creation routine which is similar to clnt_create() but which also has the additional parameter *timeout* that specifies the maximum amount of time allowed for each transport class tried. In all other respects, the clnt_create_timed() call behaves exactly like the clnt_create() call.

clnt_create_vers()
Generic client creation routine which is similar to clnt_create() but which also checks for the version availability. *host* identifies the name of the remote host where the server is located. *nettype* indicates the class transport protocols to be used. If the routine is successful it returns a client handle created for the highest version between *vers_low* and *vers_high* that is supported by the server. *vers_outp* is set to this value. That is, after a successful return *vers_low* <= *vers_outp* <= *vers_high*. If no version between *vers_low* and *vers_high* is supported by the server then the routine fails and returns NULL. A default timeout is set and can be modified using clnt_control(). This routine returns NULL if it fails. The clnt_pcreateerror() routine can be used to print the reason for failure.

Note: clnt_create() returns a valid client handle even if the particular version number supplied to clnt_create() is not registered with the rpcbind service. This mismatch will be discovered by a clnt_call later (see rpc_clnt_calls(3NSL)). However, clnt_create_vers() does this for you and returns a valid handle only if a version within the range supplied is supported by the server.

clnt_create_vers_timed()
Generic client creation routine similar to clnt_create_vers() but with the additional parameter *timeout*, which specifies the maximum amount of time allowed for each transport class tried. In all other respects, the clnt_create_vers_timed() call behaves exactly like the clnt_create_vers() call.

clnt_destroy()
A function macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including *clnt* itself. Use of *clnt* is undefined after calling clnt_destroy(). If the RPC library opened the associated file descriptor, or CLSET_FD_CLOSE was set using clnt_control(), the file descriptor will be closed.

The caller should call auth_destroy(*clnt*->cl_auth) (before calling clnt_destroy()) to destroy the associated AUTH structure (see rpc_clnt_auth(3NSL)).

clnt_dg_create()
This routine creates an RPC client for the remote program *prognum* and version *versnum*; the client uses a connectionless transport. The remote program is located at address *svcaddr*. The parameter *fildes* is an open and bound file descriptor. This routine will resend the call message in intervals of 15 seconds until a response is received or until the call times out. The total time for the call to time out is specified by clnt_call() (see clnt_call() in rpc_clnt_calls(3NSL)). The retry time out and the total time out periods can be changed

using clnt_control(). The user may set the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of 0 choose suitable defaults. This routine returns NULL if it fails.

clnt_pcreateerror()
    Print a message to standard error indicating why a client RPC handle could not be created. The message is prepended with the string *s* and a colon, and appended with a newline.

clnt_raw_create()
    This routine creates an RPC client handle for the remote program *prognum* and version *versnum*. The transport used to pass messages to the service is a buffer within the process's address space, so the corresponding RPC server should live in the same address space; (see svc_raw_create() in rpc_svc_create(3NSL)). This allows simulation of RPC and measurement of RPC overheads, such as round trip times, without any kernel or networking interference. This routine returns NULL if it fails. clnt_raw_create() should be called after svc_raw_create().

clnt_spcreateerror()
    Like clnt_pcreateerror(), except that it returns a string instead of printing to the standard error. A newline is not appended to the message in this case.

    Warning: returns a pointer to a buffer that is overwritten on each call. In multithread applications, this buffer is implemented as thread-specific data.

clnt_tli_create()
    This routine creates an RPC client handle for the remote program *prognum* and version *versnum*. The remote program is located at address *svcaddr*. If *svcaddr* is NULL and it is connection-oriented, it is assumed that the file descriptor is connected. For connectionless transports, if *svcaddr* is NULL, RPC_UNKNOWNADDR error is set. *fildes* is a file descriptor which may be open, bound and connected. If it is RPC_ANYFD, it opens a file descriptor on the transport specified by *netconf*. If *fildes* is RPC_ANYFD and *netconf* is NULL, a RPC_UNKNOWNPROTO error is set. If *fildes* is unbound, then it will attempt to bind the descriptor. The user may specify the size of the buffers with the parameters *sendsz* and *recvsz*; values of 0 choose suitable defaults. Depending upon the type of the transport (connection-oriented or connectionless), clnt_tli_create() calls appropriate client creation routines. This routine returns NULL if it fails. The clnt_pcreateerror() routine can be used to print the reason for failure. The remote rpcbind service (see rpcbind(1M)) is not consulted for the address of the remote service.

clnt_tp_create()
    Like clnt_create() except clnt_tp_create() tries only one transport specified through *netconf*.

    clnt_tp_create() creates a client handle for the program *prognum*, the version *versnum*, and for the transport specified by *netconf*. Default options are set, which can be changed using clnt_control() calls. The remote rpcbind service on the host *host* is consulted for the address of the remote service. This routine returns NULL if it fails. The clnt_pcreateerror() routine can be used to print the reason for failure.

clnt_tp_create_timed()
> Like clnt_tp_create() except clnt_tp_create_timed() has the extra parameter *timeout* which specifies the maximum time allowed for the creation attempt to succeed. In all other respects, the clnt_tp_create_timed() call behaves exactly like the clnt_tp_create() call.

clnt_vc_create()
> This routine creates an RPC client for the remote program *prognum* and version *versnum*; the client uses a connection-oriented transport. The remote program is located at address *svcaddr*. The parameter *fildes* is an open and bound file descriptor. The user may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of 0 choose suitable defaults. This routine returns NULL if it fails.
>
> The address *svcaddr* should not be NULL and should point to the actual address of the remote program. clnt_vc_create() does not consult the remote rpcbind service for this information.

rpc_createerr
> A global variable whose value is set by any RPC client handle creation routine that fails. It is used by the routine clnt_pcreateerror() to print the reason for the failure.
>
> In multithreaded applications, rpc_createerr becomes a macro which enables each thread to have its own rpc_createerr.

clnt_door_create()
> This routine creates an RPC client handle over doors for the given program *prognum* and version *versnum*. Doors is a transport mechanism that facilitates fast data transfer between processes on the same machine. The user may set the size of the send buffer with the parameter *sendsz*. If *sendsz* is 0, the corresponding default buffer size is 16 Kbyte. The clnt_door_create() routine returns NULL if it fails and sets a value for rpc_createerr.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	All
Availability	system/library
Interface Stability	Committed
MT-Level	MT-Safe

See Also   rpcbind(1M), rpc(3NSL), rpc_clnt_auth(3NSL), rpc_clnt_calls(3NSL), rpc_svc_create(3NSL), svc_raw_create(3NSL), threads(5), attributes(5)

Name rpc_control – library routine for manipulating global RPC attributes for client and server applications

Synopsis bool_t rpc_control(int *op*, void **info*);

Description This RPC library routine allows applications to set and modify global RPC attributes that apply to clients as well as servers. At present, it supports only server side operations. This function allows applications to set and modify global attributes that apply to client as well as server functions. *op* indicates the type of operation, and *info* is a pointer to the operation specific information. The supported values of *op* and their argument types, and what they do are:

```
RPC_SVC_MTMODE_SET      int *    set multithread mode
RPC_SVC_MTMODE_GET      int *    get multithread mode
RPC_SVC_THRMAX_SET      int *    set maximum number of threads
RPC_SVC_THRMAX_GET      int *    get maximum number of threads
RPC_SVC_THRTOTAL_GET    int *    get number of active threads
RPC_SVC_THRCREATES_GET  int *    get number of threads created
RPC_SVC_THRERRORS_GET   int *    get number of thread create errors
RPC_SVC_USE_POLLFD      int *    set number of file descriptors to unlimited
RPC_SVC_CONNMAXREC_SET  int *    set non-blocking max rec size
RPC_SVC_CONNMAXREC_GET  int *    get non-blocking max rec size
```

There are three multithread (MT) modes. These are:

```
RPC_SVC_MT_NONE    Single threaded mode    (default)
RPC_SVC_MT_AUTO    Automatic MT mode
RPC_SVC_MT_USER    User MT mode
```

Unless the application sets the Automatic or User MT modes, it will stay in the default (single threaded) mode. See the *Network Interfaces Programmer's Guide* for the meanings of these modes and programming examples. Once a mode is set, it cannot be changed.

By default, the maximum number of threads that the server will create at any time is 16. This allows the service developer to put a bound on thread resources consumed by a server. If a server needs to process more than 16 client requests concurrently, the maximum number of threads must be set to the desired number. This parameter may be set at any time by the server.

Set and get operations will succeed even in modes where the operations don't apply. For example, you can set the maximum number of threads in any mode, even though it makes sense only for the Automatic MT mode. All of the get operations except RPC_SVC_MTMODE_GET apply only to the Automatic MT mode, so values returned in other modes may be undefined.

By default, RPC servers are limited to a maximum of 1024 file descriptors or connections due to limitations in the historical interfaces svc_fdset(3NSL) and svc_getreqset(3NSL). Applications written to use the preferred interfaces of svc_pollfd(3NSL) and svc_getreq_poll(3NSL) can use an unlimited number of file descriptors. Setting info to point to a non-zero integer and *op* to RPC_SVC_USE_POLLFD removes the limitation.

Connection oriented RPC transports read RPC requests in blocking mode by default. Thus, they may be adversely affected by network delays and broken clients. RPC_SVC_CONNMAXREC_SET enables non-blocking mode and establishes the maximum record size (in bytes) for RPC requests; RPC responses are not affected. Buffer space is allocated as needed up to the specified maximum, starting at the maximum or RPC_MAXDATASIZE, whichever is smaller.

The value established by RPC_SVC_CONNMAXREC_SET is used when a connection is created, and it remains in effect for that connection until it is closed. To change the value for existing connections on a per-connection basis, see svc_control(3NSL).

RPC_SVC_CONNMAXREC_GET retrieves the current maximum record size. A zero value means that no maximum is in effect, and that the connections are in blocking mode.

info is a pointer to an argument of type int. Non-connection RPC transports ignore RPC_SVC_CONNMAXREC_SET and RPC_SVC_CONNMAXREC_GET.

Return Values This routine returns TRUE if the operation was successful and returnsFALSE otherwise.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also rpcbind(1M), rpc(3NSL), rpc_svc_calls(3NSL), attributes(5)

Network Interfaces Programmer's Guide

Name rpc_gss_getcred – get credentials of client

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

```
bool_t rpc_gss_getcred(struct svc_req *req, rpc_gss_rawcred_ t **rcred,
    rpc_gss_ucred **ucred, void **cookie);
```

Description rpc_gss_getcred() is used by a server to fetch the credentials of a client. These credentials may either be network credentials (in the form of a rpc_gss_rawcred_t structure) or UNIX credentials.

For more information on RPCSEC_GSS data types, see the rpcsec_gss(3NSL) man page.

Parameters Essentially, rpc_gss_getcred() passes a pointer to a request (svc_req) as well as pointers to two credential structures and a user-defined cookie; if rpc_gss_getcred() is successful, at least one credential structure is "filled out" with values, as is, optionally, the cookie.

> *req*   Pointer to the received service request. svc_req is an RPC structure containing information on the context of an RPC invocation, such as program, version, and transport information.
>
> *rcred*   A pointer to an rpc_gss_rawcred_t structure pointer. This structure contains the version number of the RPCSEC_GSS protocol being used; the security mechanism and QOPs for this session (as strings); principal names for the client (as a rpc_gss_principal_t structure) and server (as a string); and the security service (integrity, privacy, etc., as an enum). If an application is not interested in these values, it may pass NULL for this parameter.
>
> *ucred*   The caller's UNIX credentials, in the form of a pointer to a pointer to a rpc_gss_ucred_t structure, which includes the client's uid and gids. If an application is not interested in these values, it may pass NULL for this parameter.
>
> *cookie*   A four-byte quantity that an application may use in any manner it wants to; RPC does not interpret it. (For example, a cookie may be a pointer or index to a structure that represents a context initiator.) See also rpc_gss_set_callback(3NSL).

Return Values rpc_gss_getcred() returns TRUE if it is successful; otherwise, use rpc_gss_get_error() to get the error associated with the failure.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/rpcsec
MT-Level	MT-Safe

See Also rpc(3NSL), rpc_gss_set_callback(3NSL), rpc_gss_set_svc_name(3NSL),
rpcsec_gss(3NSL), attributes(5)

ONC+ Developer's Guide

Linn, J. *RFC 2078, Generic Security Service Application Program Interface, Version 2*. Network
Working Group. January 1997.

Name   rpc_gss_get_error – get error codes on failure

Synopsis   #include <rpc/rpcsec_gss.h>

bool_t rpc_gss_get_error(rpc_gss_error_t *error);

Description   rpc_gss_get_error() fetches an error code when an RPCSEC_GSS routine fails.

rpc_gss_get_error() uses a rpc_gss_error_t structure of the following form:

```
typedef struct {
int    rpc_gss_error;        RPCSEC_GSS error
int    system_error;         system error
} rpc_gss_error_t;
```

Currently the only error codes defined for this function are

```
#define RPC_GSS_ER_SUCCESS        0    /* no error */
#define RPC_GSS_ER_SYSTEMERROR    1    /* system error */
```

Parameters   Information on RPCSEC_GSS data types for parameters may be found on the rpcsec_gss(3NSL) man page.

error   A rpc_gss_error_t structure. If the rpc_gss_error field is equal to RPC_GSS_ER_SYSTEMERROR, the system_error field will be set to the value of errno.

Return Values   Unless there is a failure indication from an invoked RPCSEC_GSS function, rpc_gss_get_error() does not set error to a meaningful value.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/rpcsec
MT-Level	MT-Safe

See Also   perror(3C), rpc(3NSL), rpcsec_gss(3NSL), attributes(5)

ONC+ Developer's Guide

Linn, J. *RFC 2078, Generic Security Service Application Program Interface, Version 2*. Network Working Group. January 1997.

Notes   Only system errors are currently returned.

Name rpc_gss_get_mechanisms, rpc_gss_get_mech_info, rpc_gss_get_versions, rpc_gss_is_installed – get information on mechanisms and RPC version

Synopsis #include <rpc/rpcsec_gss.h>

char **rpc_gss_get_mechanisms();

char **rpc_gss_get_mech_info(char *mech, rpc_gss_service_t *service);

bool_t rpc_gss_get_versions(u_int *vers_hi, u_int *vers_lo);

bool_t rpc_gss_is_installed(char *mech);

Description These "convenience functions" return information on available security mechanisms and versions of RPCSEC_GSS .

rpc_gss_get_mechanisms()	Returns a list of supported security mechanisms as a null-terminated list of character strings.
rpc_gss_get_mech_info()	Takes two arguments: an ASCII string representing a mechanism type, for example, kerberosv5, and a pointer to a rpc_gss_service_t enum. rpc_gss_get_mech_info() will return NULL upon error or if no /etc/gss/qop file is present. Otherwise, it returns a null-terminated list of character strings of supported Quality of Protections (QOPs) for this mechanism. NULL or empty list implies only that the default QOP is available and can be specified to routines that need to take a QOP string parameter as NULL or as an empty string.
rpc_gss_get_versions()	Returns the highest and lowest versions of RPCSEC_GSS supported.
rpc_gss_is_installed()	Takes an ASCII string representing a mechanism, and returns TRUE if the mechanism is installed.

Parameters Information on RPCSEC_GSS data types for parameters may be found on the rpcsec_gss(3NSL) man page.

mech An ASCII string representing the security mechanism in use. Valid strings may also be found in the /etc/gss/mech file.

service A pointer to a rpc_gss_service_t enum, representing the current security service (privacy, integrity, or none).

vers_hi
vers_lo The highest and lowest versions of RPCSEC_GSS supported.

Files /etc/gss/mech File containing valid security mechanisms

/etc/gss/qop File containing valid QOP values

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/rpcsec
MT-Level	MT-Safe

See Also rpc(3NSL), rpcsec_gss(3NSL), mech(4), qop(4), attributes(5)

ONC+ Developer's Guide

Linn, J. *RFC 2743, Generic Security Service Application Program Interface Version 2, Update 1*. Network Working Group. January 2000.

Notes This function will change in a future release.

Name rpc_gss_get_principal_name – Get principal names at server

Synopsis #include <rpc/rpcsec_gss.h>

bool_t rpc_gss_get_principal_name(rpc_gss_principal_ *principal,
    char *mech, char *name, char *node, char *domain);

Description Servers need to be able to operate on a client's principal name. Such a name is stored by the server as a rpc_gss_principal_t structure, an opaque byte string which can be used either directly in access control lists or as database indices which can be used to look up a UNIX credential. A server may, for example, need to compare a principal name it has received with the principal name of a known entity, and to do that, it must be able to generate rpc_gss_principal_t structures from known entities.

rpc_gss_get_principal_name() takes as input a security mechanism, a pointer to a rpc_gss_principal_t structure, and several parameters which uniquely identify an entity on a network: a user or service name, a node name, and a domain name. From these parameters it constructs a unique, mechanism-dependent principal name of the rpc_gss_principal_t structure type.

Parameters How many of the identifying parameters (*name* , *node*, and domain*)* are necessary to specify depends on the mechanism being used. For example, Kerberos V5 requires only a user name but can accept a node and domain name. An application can choose to set unneeded parameters to NULL .

Information on RPCSEC_GSS data types for parameters may be found on the rpcsec_gss(3NSL) man page.

principal	An opaque, mechanism-dependent structure representing the client's principal name.
mech	An ASCII string representing the security mechanism in use. Valid strings may be found in the /etc/gss/mech file, or by using rpc_gss_get_mechanisms().
name	A UNIX login name (for example, 'gwashington') or service name, such as 'nfs'.
node	A node in a domain; typically, this would be a machine name (for example, 'valleyforge').
domain	A security domain; for example, a DNS or NIS domain name ('eng.company.com').

Return Values rpc_gss_get_principal_name() returns TRUE if it is successful; otherwise, use rpc_gss_get_error() to get the error associated with the failure.

Files /etc/gss/mech   File containing valid security mechanisms

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/rpcsec
MT-Level	MT-Safe

See Also free(3C), rpc(3NSL), rpc_gss_get_mechanisms(3NSL), rpc_gss_set_svc_name(3NSL), rpcsec_gss(3NSL), mech(4), attributes(5)

ONC+ Developer's Guide

Linn, J. *RFC 2078, Generic Security Service Application Program Interface, Version 2.* Network Working Group. January 1997.

Notes Principal names may be freed up by a call to free(3C). A principal name need only be freed in those instances where it was constructed by the application. (Values returned by other routines point to structures already existing in a context, and need not be freed.)

Name rpc_gss_max_data_length, rpc_gss_svc_max_data_length – get maximum data length for transmission

Synopsis #include <rpc/rpcsec_gss.h>

int rpc_gss_max_data_length(AUTH *handle, int *max_tp_unit_len*);

int rpc_gss_svc_max_data_length(struct svc_req *req, int *max_tp_unit_len*);

Description Performing a security transformation on a piece of data generally produces data with a different (usually greater) length. For some transports, such as UDP, there is a maximum length of data which can be sent out in one data unit. Applications need to know the maximum size a piece of data can be before it's transformed, so that the resulting data will still "fit" on the transport. These two functions return that maximum size.

rpc_gss_max_data_length() is the client-side version; rpc_gss_svc_max_data_length() is the server-side version.

Parameters *handle*   An RPC context handle of type AUTH, returned when a context is created (for example, by rpc_gss_seccreate(). Security service and QOP are bound to this handle, eliminating any need to specify them.

max_tp_unit_len   The maximum size of a piece of data allowed by the transport.

req   A pointer to an RPC svc_req structure, containing information on the context (for example, program number and credentials).

Return Values Both functions return the maximum size of untransformed data allowed, as an int.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/rpcsec
MT-Level	MT-Safe

See Also rpc(3NSL), rpcsec_gss(3NSL), attributes(5)

ONC+ Developer's Guide

Linn, J. *RFC 2078, Generic Security Service Application Program Interface, Version 2*. Network Working Group. January 1997.

Name   rpc_gss_mech_to_oid, rpc_gss_qop_to_num – map mechanism, QOP strings to non-string values

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

`bool_t rpc_gss_mech_to_oid(charc*`*mech*`, rpc_gss_OIDc*`*oid*`);`

`bool_t rpc_gss_qop_to_num(char *`*qop*`, char *`*mech*`, u_int *`*num*`);`

Description   Because in-kernel RPC routines use non-string values for mechanism and Quality of Protection (QOP), these routines exist to map strings for these attributes to their non-string counterparts. (The non-string values for QOP and mechanism are also found in the /etc/gss/qop and /etc/gss/mech files, respectively.) `rpc_gss_mech_to_oid()` takes a string representing a mechanism, as well as a pointer to a `rpc_gss_OID` object identifier structure. It then gives this structure values corresponding to the indicated mechanism, so that the application can now use the OID directly with RPC routines. `rpc_gss_qop_to_num()` does much the same thing, taking strings for QOP and mechanism and returning a number.

Parameters   Information on RPCSEC_GSS data types for parameters may be found on the rpcsec_gss(3NSL) man page.

   mech   An ASCII string representing the security mechanism in use. Valid strings may be found in the /etc/gss/mech file.

   oid   An object identifier of type `rpc_gss_OID`, whose elements are usable by kernel-level RPC routines.

   qop   This is an ASCII string which sets the quality of protection (QOP) for the session. Appropriate values for this string may be found in the file /etc/gss/qop.

   num   The non-string value for the QOP.

Return Values   Both functions return TRUE if they are successful, FALSE otherwise.

Files   /etc/gss/mech   File containing valid security mechanisms

   /etc/gss/qop   File containing valid QOP values

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/rpcsec
MT-Level	MT-Safe

See Also   rpc(3NSL), rpc_gss_get_error(3NSL), rpc_gss_get_mechanisms(3NSL), rpcsec_gss(3NSL), mech(4), qop(4), attributes(5)

ONC+ Developer's Guide

Linn, J. *RFC 2078, Generic Security Service Application Program Interface, Version 2*. Network Working Group. January 1997.

Name   rpc_gss_seccreate – create a security context using the RPCSEC_GSS protocol

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

```
AUTH  *rpc_gss_seccreate(CLIENT *clnt, char *principal, char *mechanism,
     rpc_gss_service_t service_type, char *qop,
     rpc_gss_options_req_t *options_req,
     rpc_gss_options_ret_t *options_ret);
```

Description   rpc_gss_seccreate() is used by an appliction to create a security context using the
RPCSEC_GSS protocol, making use of the underlying GSS-API network layer.
rpc_gss_seccreate() allows an application to specify the type of security mechanism (for
example, Kerberos v5), the type of service (for example, integrity checking), and the Quality of
Protection (QOP) desired for transferring data.

Parameters   Information on RPCSEC_GSS data types for parameters may be found on the
rpcsec_gss(3NSL) man page.

clnt	This is the RPC client handle. *clnt* may be obtained, for example, from `clnt_create()`.
principal	This is the identity of the server principal, specified in the form *service@host*, where *service* is the name of the service the client wishes to access and *host* is the fully qualified name of the host where the service resides — for example, nfs@mymachine.eng.company.com.
mechanism	This is an ASCII string which indicates which security mechanism to use with this data. Appropriate mechanisms may be found in the file /etc/gss/mech; additionally, rpc_gss_get_mechanisms() returns a list of supported security mechanisms (as null-terminated strings).
service_type	This sets the initial type of service for the session — privacy, integrity, authentication, or none.
qop	This is an ASCII string which sets the quality of protection (QOP) for the session. Appropriate values for this string may be found in the file /etc/gss/qop. Additionally, supported QOPs are returned (as null-terminated strings) by rpc_gss_get_mech_info().
options_req	This structure contains options which are passed directly to the underlying GSS_API layer. If the caller specifies NULL for this parameter, defaults are used. (See NOTES, below.)
options_ret	These GSS-API options are returned to the caller. If the caller does not need to see these options, then it may specify NULL for this parameter. (See NOTES, below.)

Return Values rpc_gss_seccreate() returns a security context handle (an RPC authentication handle) of type AUTH. If rpc_gss_seccreate() cannot return successfully, the application can get an error number by calling rpc_gss_get_error().

Files /etc/gss/mech   File containing valid security mechanisms

/etc/gss/qop   File containing valid QOP values.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/rpcsec
MT-Level	MT-Safe

See Also auth_destroy(3NSL), rpc(3NSL), rpc_gss_get_error(3NSL), rpc_gss_get_mechanisms(3NSL), rpcsec_gss(3NSL), mech(4), qop(4), attributes(5)

ONC+ Developer's Guide

Linn, J. *RFC 2743, Generic Security Service Application Program Interface Version 2, Update 1.* Network Working Group. January 2000.

Notes Contexts may be destroyed normally, with auth_destroy(). See auth_destroy(3NSL)

Name rpc_gss_set_callback – specify callback for context

Synopsis #include <rpc/rpcsec_gss.h>

bool_t rpc_gss_set_callback(struct rpc_gss_callback_t *cb);

Description A server may want to specify a callback routine so that it knows when a context gets first used. This user-defined callback may be specified through the rpc_gss_set_callback() routine. The callback routine is invoked the first time a context is used for data exchanges, after the context is established for the specified program and version.

The user-defined callback routine should take the following form:

bool_t callback(struct svc_req *req, gss_cred_id_t *deleg,
    gss_ctx_id_t *gss_context, rpc_gss_lock_t *lock, void **cookie);

Parameters rpc_gss_set_callback() takes one argument: a pointer to a rpc_gss_callback_t structure. This structure contains the RPC program and version number as well as a pointer to a user-defined callback() routine. (For a description of rpc_gss_callback_t and other RPCSEC_GSS data types, see the rpcsec_gss(3NSL) man page.)

The user-defined callback() routine itself takes the following arguments:

req	Pointer to the received service request. svc_req is an RPC structure containing information on the context of an RPC invocation, such as program, version, and transport information.
deleg	Delegated credentials, if any. (See NOTES, below.)
gss_context	GSS context (allows server to do GSS operations on the context to test for acceptance criteria). See NOTES, below.
lock	This parameter is used to enforce a particular QOP and service for a session. This parameter points to a RPCSEC_GSS rpc_gss_lock_t structure. When the callback is invoked, the rpc_gss_lock_t.locked field is set to TRUE, thus locking the context. A locked context will reject all requests having different values for QOP or service than those specified by the raw_cred field of the rpc_gss_lock_t structure.
cookie	A four-byte quantity that an application may use in any manner it wants to — RPC does not interpret it. (For example, the cookie could be a pointer or index to a structure that represents a context initiator.) The cookie is returned, along with the caller's credentials, with each invocation of rpc_gss_getcred().

Return Values rpc_gss_set_callback() returns TRUE if the use of the context is accepted; false otherwise.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/rpcsec
MT-Level	MT-Safe

See Also rpc(3NSL), rpc_gss_getcred(3NSL), rpcsec_gss(3NSL), attributes(5)

ONC+ Developer's Guide

Linn, J. *RFC 2078, Generic Security Service Application Program Interface, Version 2*. Network Working Group. January 1997.

Notes If a server does not specify a callback, all incoming contexts will be accepted.

Because the GSS-API is not currently exposed, the *deleg* and *gss_context* arguments are mentioned for informational purposes only, and the user-defined callback function may choose to do nothing with them.

Name rpc_gss_set_defaults – change service, QOP for a session

Synopsis #include <rpc/rpcsec_gss.h>

bool_t rpc_gss_set_defaults(AUTH *auth, rpc_gss_service_t *service*, char *qop*);

Description rpc_gss_set_defaults() allows an application to change the service (privacy, integrity, authentication, or none) and Quality of Protection (QOP) for a transfer session. New values apply to the rest of the session (unless changed again).

Parameters Information on RPCSEC_GSS data types for parameters may be found on the rpcsec_gss(3NSL) man page.

auth   An RPC authentication handle returned by rpc_gss_seccreate()).

service   An enum of type rpc_gss_service_t, representing one of the following types of security service: authentication, privacy, integrity, or none.

qop   A string representing Quality of Protection. Valid strings may be found in the file /etc/gss/qop or by using rpc_gss_get_mech_info().

Return Values rpc_gss_set_svc_name() returns TRUE if it is successful; otherwise, use rpc_gss_get_error() to get the error associated with the failure.

Files /etc/gss/qop   File containing valid QOPs

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/rpcsec
MT-Level	MT-Safe

See Also rpc(3NSL), rpc_gss_get_mech_info(3NSL), rpcsec_gss(3NSL), qop(4), attributes(5)

ONC+ Developer's Guide

Linn, J. *RFC 2078, Generic Security Service Application Program Interface, Version 2.* Network Working Group. January 1997.

Name   rpc_gss_set_svc_name – send a principal name to a server

Synopsis   #include <rpc/rpcsec_gss.h>

bool_t rpc_gss_set_svc_name(char *principal, char *mechanism,
    u_int req_time,u_int program, u_int version);

Description   rpc_gss_set_svc_name() sets the name of a principal the server is to represent. If a server is going to act as more than one principal, this procedure can be invoked for every such principal.

Parameters   Information on RPCSEC_GSS data types for parameters may be found on the rpcsec_gss(3NSL) man page.

principal	An ASCII string representing the server's principal name, given in the form of *service@host*.
mech	An ASCII string representing the security mechanism in use. Valid strings may be found in the /etc/gss/mech file, or by using rpc_gss_get_mechanisms().
req_time	The time, in seconds, for which a credential should be valid. Note that the *req_time* is a hint to the underlying mechanism. The actual time that the credential will remain valid is mechanism dependent. In the case of kerberos the actual time will be GSS_C_INDEFINITE.
program	The RPC program number for this service.
version	The RPC version number for this service.

Return Values   rpc_gss_set_svc_name() returns TRUE if it is successful; otherwise, use rpc_gss_get_error() to get the error associated with the failure.

Files   /etc/gss/mech   File containing valid security mechanisms

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/rpcsec
MT-Level	MT-Safe

See Also   rpc(3NSL), rpc_gss_get_mechanisms(3NSL), rpc_gss_get_principal_name(3NSL), rpcsec_gss(3NSL), mech(4), attributes(5)

ONC+ Developer's Guide

Linn, J. *RFC 2078, Generic Security Service Application Program Interface, Version 2*. Network Working Group. January 1997.

Name rpcsec_gss – security flavor incorporating GSS-API protections

Synopsis cc [*flag*...] *file*... -lnsl [*library*...]
#include <rpc/rpcsec_gss.h>

Description RPCSEC_GSS is a security flavor which sits "on top" of the GSS-API (Generic Security Service API) for network transmissions. Applications using RPCSEC_GSS can take advantage of GSS-API security features; moreover, they can use any security mechanism (such as RSA public key or Kerberos) that works with the GSS-API.

The GSS-API offers two security services beyond the traditional authentication services (AUTH_DH, AUTH_SYS, and AUTH_KERB): integrity and privacy. With integrity, the system uses cryptographic checksumming to ensure the authenticity of a message (authenticity of originator, recipient, and data); privacy provides additional security by encrypting data. Applications using RPCSEC_GSS specify which service they wish to use. Type of security service is mechanism-independent.

Before exchanging data with a peer, an application must establish a context for the exchange. RPCSEC_GSS provides a single function for this purpose, rpc_gss_seccreate(), which allows the application to specify the security mechanism, Quality of Protection (QOP), and type of service at context creation. (The QOP parameter sets the cryptographic algorithms to be used with integrity or privacy, and is mechanism-dependent.) Once a context is established, applications can reset the QOP and type of service for each data unit exchanged, if desired.

Valid mechanisms and QOPs may be obtained from configuration files or from the name service. Each mechanism has a default QOP.

Contexts are destroyed with the usual RPC auth_destroy() call.

Data Structures Some of the data structures used by the RPCSEC_GSS package are shown below.

rpc_gss_service_t

This enum defines the types of security services the context may have. rpc_gss_seccreate() takes this as one argument when setting the service type for a session.

```
typedef enum {
   rpc_gss_svc_default = 0,
   rpc_gss_svc_none = 1,
   rpc_gss_svc_integrity = 2,
   rpc_gss_svc_privacy = 3
} rpc_gss_service_t ;
```

rpc_gss_options_req_t

Structure containing options passed directly through to the GSS-API. rpc_gss_seccreate() takes this as an argument when creating a context.

```
typedef struct {
   int  req_flags;          /*GSS request bits */
   int  time_req;           /*requested credential lifetime */
```

```
   gss_cred_id_t  my_cred;  /*GSS credential struct*/
   gss_channel_bindings_t;
   input_channel_bindings;
} rpc_gss_options_req_t ;
```

rpc_gss_OID

This data type is used by in-kernel RPC routines, and thus is mentioned here for informational purposes only.

```
typedef struct {
   u_int    length;
   void    *elements
} *rpc_gss_OID;
```

rpc_gss_options_ret_t

Structure containing GSS-API options returned to the calling function, rpc_gss_seccreate(). MAX_GSS_MECH is defined as 128.

```
typedef struct {
   int         major_status;
   int         minor_status;
   u_int       rpcsec_version              /*vers. of RPCSEC_GSS */
   int         ret_flags
   int         time_req
   gss_ctx_id_t  gss_context;
   char        actual_mechanism[MAX_GSS_MECH]; /*mechanism used*/
} rpc_gss_options_ret_t;
```

rpc_gss_principal_t

The (mechanism-dependent, opaque) client principal type. Used as an argument to the rpc_gss_get_principal_name() function, and in the gsscred table. Also referenced by the rpc_gss_rawcred_t structure for raw credentials (see below).

```
typedef struct {
   int len;
   char name[1];
} *rpc_gss_principal_t;
```

rpc_gss_rawcred_t

Structure for raw credentials. Used by rpc_gss_getcred() and rpc_gss_set_callback().

```
typedef struct {
   u_int               version;        /*RPC version # */
   char                *mechanism;     /*security mechanism*/
   char                *qop;           /*Quality of Protection*/
   rpc_gss_principal_t client_principal; /*client name*/
   char                *svc_principal; /*server name*/
   rpc_gss_service_t   service;        /*service (integrity, etc.)*/
} rpc_gss_rawcred_t;
```

rpc_gss_ucred_t

Structure for UNIX credentials. Used by rpc_gss_getcred() as an alternative to
rpc_gss_rawcred_t.

```
typedef struct {
   uid_t  uid;      /*user ID*/
   gid_t  gid;      /*group ID*/
   short  gidlen;
   git_t  *gidlist; /*list of groups*/
} rpc_gss_ucred_t;
```

rpc_gss_callback_t

Callback structure used by rpc_gss_set_callback().

```
typedef struct {
   u_int   program;      /*RPC program #*/
   u_int   version;      /*RPC version #*/
   bool_t  (*callback)(); /*user-defined callback routine*/
} rpc_gss_callback_t;
```

rpc_gss_lock_t

Structure used by a callback routine to enforce a particular QOP and service for a session. The
locked field is normally set to FALSE; the server sets it to TRUE in order to lock the session. (A
locked context will reject all requests having different QOP and service values than those
found in the raw_cred structure.) For more information, see the
rpc_gss_set_callback(3NSL) man page.

```
typedef struct {
   bool_t               locked;
   rpc_gss_rawcred_t    *raw_cred;
} rpc_gss_lock_t;
```

rpc_gss_error_t

Structure used by rpc_gss_get_error() to fetch an error code when a RPCSEC_GSS routine
fails.

```
typedef struct {
   int  rpc_gss_error;
   int  system_error;    /*same as errno*/
} rpc_gss_error_t;
```

Index to Routines   The following lists RPCSEC_GSS routines and the manual reference pages on which they are
described. An (S) indicates it is a server-side function:

Routine (Manual Page)	Description
rpc_gss_seccreate(3NSL)	Create a secure RPCSEC_GSS context
rpc_gss_set_defaults(3NSL)	Switch service, QOP for a session

rpc_gss_max_data_length(3NSL)	Get maximum data length allowed by transport
rpc_gss_set_svc_name(3NSL)	Set server's principal name (S)
rpc_gss_getcred(3NSL)	Get credentials of caller (S)
rpc_gss_set_callback(3NSL)	Specify callback to see context use (S)
rpc_gss_get_principal_name(3NSL)	Get client principal name (S)
rpc_gss_svc_max_data_length(3NSL)	Get maximum data length allowed by transport (S)
rpc_gss_get_error(3NSL)	Get error number
rpc_gss_get_mechanisms(3NSL)	Get valid mechanism strings
rpc_gss_get_mech_info(3NSL)	Get valid QOP strings, current service
rpc_gss_get_versions(3NSL)	Get supported RPCSEC_GSS versions
rpc_gss_is_installed(3NSL)	Checks if a mechanism is installed
rpc_gss_mech_to_oid(3NSL)	Maps ASCII mechanism to OID representation
rpc_gss_qop_to_num(3NSL)	Maps ASCII QOP, mechansim to u_int number

Utilities   The gsscred utility manages the gsscred table, which contains mappings of principal names between network and local credentials. See gsscred(1M).

Files   /etc/gss/mech   List of installed mechanisms

/etc/gss/qop   List of valid QOPs

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/rpcsec
MT-Level	MT-Safe

See Also   gsscred(1M), rpc(3NSL), rpc_clnt_auth(3NSL), xdr(3NSL), attributes(5), environ(5)

ONC+ Developer's Guide

Linn, J. *RFC 2743, Generic Security Service Application Program Interface Version 2, Update 1.* Network Working Group. January 2000.

Name rpc_soc, authdes_create, authunix_create, authunix_create_default, callrpc, clnt_broadcast,
clntraw_create, clnttcp_create, clntudp_bufcreate, clntudp_create, get_myaddress,
getrpcport, pmap_getmaps, pmap_getport, pmap_rmtcall, pmap_set, pmap_unset,
registerrpc, svc_fds, svc_getcaller, svc_getreq, svc_register, svc_unregister, svcfd_create,
svcraw_create, svctcp_create, svcudp_bufcreate, svcudp_create, xdr_authunix_parms –
obsolete library routines for RPC

Synopsis
```
#define PORTMAP
#include <rpc/rpc.h>

AUTH *authdes_create(char *name, uint_t window,
     struct sockaddr_in *syncaddr, des_block *ckey);

AUTH *authunix_create(char *host, uid_t uid, gid_t gid,
     int grouplen, gid_t *gidlistp);

AUTH *authunix_create_default(void)

callrpc(char *host, rpcprog_t prognum, rpcvers_t versnum,
     rpcproc_t procnum, xdrproc_t inproc, char *in,
     xdrproc_t outproc, char *out);

enum clnt_stat clnt_broadcast(rpcprog_t prognum, rpcvers_t versnum,
     rpcproc_t procnum, xdrproc_t inproc, char *in,
     xdrproc_t outproc, char *out, resultproc_t eachresult);

CLIENT *clntraw_create(rpcproc_t procnum, rpcvers_t versnum);

CLIENT *clnttcp_create(struct sockaddr_in *addr,
     rpcprog_t prognum, rpcvers_t versnum, int *fdp,
     uint_t sendz, uint_t recvsz);

CLIENT *clntudp_bufcreate(struct sockaddr_in *addr, rpcprog_t prognum,
     rpcvers_t versnum, struct timeval wait,
     int *fdp, uint_t sendz, uint_t recvsz);

CLIENT *clntudp_create(struct sockaddr_in *addr,
     rpcprog_t prognum, struct timeval wait, int *fdp);

void get_myaddress(struct sockaddr_in *addr);

ushort getrpcport(char *host, rpcprog_t prognum,
     rpcvers_t versnum, rpcprot_t proto);

struct pmaplist *pmap_getmaps(struct sockaddr_in *addr);

ushort pmap_getport(struct sockaddr_in *addr,
     rpcprog_t prognum, rpcvers_t versnum,
     rpcprot_t protocol);

enum clnt_stat pmap_rmtcall(struct sockaddr_in *addr,
     rpcprog_t prognum, rpcvers_t versnum,
     rpcproc_t progcnum, caddr_t in, xdrproct_t inproc,
     caddr_t out, cdrproct_t outproc,
     struct timeval tout, rpcport_t *portp);
```

```
bool_t pmap_set(rpcprog_t prognum, rpcvers_t versnum,
    rpcprot_t protocol, u_short port);

bool_t pmap_unset(rpcprog_t prognum, rpcvers_t versnum);

int svc_fds;

struct sockaddr_in *svc_getcaller(SVCXPRT *xprt);

void svc_getreq(int rdfds);

SVCXPRT *svcfd_create(int fd, uint_t sendsz,
    uint_t recvsz);

SVCXPRT *svcraw_create(void)

SVCXPRT *svctcp_create(int fd, uint_t sendsz,
    uint_t recvsz);

SVCXPRT *svcudp_bufcreate(int fd, uint_t sendsz,
    uint_t recvsz);

SVCXPRT *svcudp_create(int fd);

registerrpc(rpcprog_t prognum, rpcvers_t versnum, rpcproc_t procnum,
    char *(*procname)(), xdrproc_t inproc, xdrproc_t outproc);

bool_tsvc_register(SVCXPRT *xprt, rpcprog_t prognum, rpcvers_t versnum,
    void (*dispatch(), int protocol);

void svc_unregister(rpcprog_t prognum, rpcvers_t versnum);

bool_t xdr_authunix_parms(XDR *xdrs, struct authunix_parms *supp);
```

Description   RPC routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a request to the server. Upon receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

The routines described in this manual page have been superseded by other routines. The preferred routine is given after the description of the routine. New programs should use the preferred routines, as support for the older interfaces may be dropped in future releases.

File Descriptors   Transport independent RPC uses TLI as its transport interface instead of sockets.

Some of the routines described in this section (such as clnttcp_create()) take a pointer to a file descriptor as one of the parameters. If the user wants the file descriptor to be a socket, then the application will have to be linked with both librpcsoc and libnsl. If the user passed RPC_ANYSOCK as the file descriptor, and the application is linked with libnsl only, then the routine will return a TLI file descriptor and not a socket.

Routines   The following routines require that the header <rpc/rpc.h> be included. The symbol PORTMAP
            should be defined so that the appropriate function declarations for the old interfaces are
            included through the header files.

authdes_create()   authdes_create() is the first of two routines which
                                     interface to the RPC secure authentication system, known
                                     as DES authentication. The second is
                                     authdes_getucred(), below. Note: the keyserver
                                     daemon keyserv(1M) must be running for the DES
                                     authentication system to work.

                                     authdes_create(), used on the client side, returns an
                                     authentication handle that will enable the use of the
                                     secure authentication system. The first parameter *name* is
                                     the network name, or *netname*, of the owner of the server
                                     process. This field usually represents a hostname derived
                                     from the utility routine host2netname(), but could also
                                     represent a user name using user2netname(). See
                                     secure_rpc(3NSL). The second field is window on the
                                     validity of the client credential, given in seconds. A small
                                     window is more secure than a large one, but choosing too
                                     small of a window will increase the frequency of
                                     resynchronizations because of clock drift. The third
                                     parameter *syncaddr* is optional. If it is NULL, then the
                                     authentication system will assume that the local clock is
                                     always in sync with the server's clock, and will not attempt
                                     resynchronizations. If an address is supplied, however,
                                     then the system will use the address for consulting the
                                     remote time service whenever resynchronization is
                                     required. This parameter is usually the address of the RPC
                                     server itself. The final parameter *ckey* is also optional. If it
                                     is NULL, then the authentication system will generate a
                                     random DES key to be used for the encryption of
                                     credentials. If it is supplied, however, then it will be used
                                     instead.

                                     This routine exists for backward compatibility only, and it
                                     is made obsolete by authdes_seccreate(). See
                                     secure_rpc(3NSL).

authunix_create()   Create and return an RPC authentication handle that
                                     contains .UX authentication information. The parameter
                                     host is the name of the machine on which the information
                                     was created; *uid* is the user's user ID; *gid* is the user's

current group ID; *grouplen* and *gidlistp* refer to a counted
array of groups to which the user belongs.

It is not very difficult to impersonate a user.

This routine exists for backward compatibility only, and it
is made obsolete by authsys_create(). See
rpc_clnt_auth(3NSL).

authunix_create_default()   Call authunix_create() with the appropriate
                            parameters.

                            This routine exists for backward compatibility only, and it
                            is made obsolete by authsys_create_default(). See
                            rpc_clnt_auth(3NSL).

callrpc()   Call the remote procedure associated with *prognum*,
                            versnum, and *procnum* on the machine, *host*. The
                            parameter *inproc* is used to encode the procedure's
                            parameters, and *outproc* is used to decode the procedure's
                            results; *in* is the address of the procedure's argument, and
                            out is the address of where to place the result(s). This
                            routine returns 0 if it succeeds, or the value of enum
                            clnt_stat cast to an integer if it fails. The routine
                            clnt_perrno() is handy for translating failure statuses
                            into messages. See rpc_clnt_calls(3NSL).

                            You do not have control of timeouts or authentication
                            using this routine. This routine exists for backward
                            compatibility only, and is made obsolete by rpc_call().
                            See rpc_clnt_calls(3NSL).

clnt_stat_clnt_broadcast() Like callrpc(), except the call message is broadcast to all
                            locally connected broadcast nets. Each time the caller
                            receives a response, this routine calls eachresult(),
                            whose form is:

                            eachresult(char *out, struct sockaddr_in *addr);

                            where *out* is the same as *out* passed to clnt_broadcast(),
                            except that the remote procedure's output is decoded
                            there; *addr* points to the address of the machine that sent
                            the results. If eachresult() returns 0. clnt_broadcast()
                            waits for more replies; otherwise it returns with
                            appropriate status. If eachresult() is NULL,
                            clnt_broadcast() returns without waiting for any
                            replies.

Broadcast packets are limited in size to the maximum transfer unit of the transports involved. For Ethernet, the callers argument size is approximately 1500 bytes. Since the call message is sent to all connected networks, it may potentially lead to broadcast storms. `clnt_broadcast()` uses SB AUTH_SYS credentials by default. See rpc_clnt_auth(3NSL). This routine exists for backward compatibility only, and is made obsolete by `rpc_broadcast()`. See rpc_clnt_calls(3NSL).

clntraw_create()

This routine creates an internal, memory-based RPC client for the remote program *prognum*, version *versnum*. The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding RPC server should live in the same address space. See `svcraw_create()`. This allows simulation of RPC and acquisition of RPC overheads, such as round trip times, without any kernel interference. This routine returns NULL if it fails.

This routine exists for backward compatibility only. It has the same functionality as `clnt_raw_create()`. See rpc_clnt_create(3NSL), which obsoletes it.

clnttcp_create()

This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses TCP/IP as a transport. The remote program is located at Internet address *addr*. If *addr->sin_port* is 0, then it is set to the actual port that the remote program is listening on. The remote rpcbind service is consulted for this information. The parameter **fdp* is a file descriptor, which may be open and bound; if it is RPC_ANYSOCK, then this routine opens a new one and sets **fdp*. Refer to the `File Descriptor` section for more information. Since TCP-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*. Values of 0 choose suitable defaults. This routine returns NULL if it fails.

This routine exists for backward compatibility only. `clnt_create()`, `clnt_tli_create()`, or `clnt_vc_create()` should be used instead. See rpc_clnt_create(3NSL).

clntudp_bufcreate()	Create a client handle for the remote program *prognum*, on *versnum*; the client uses UDP/IP as the transport. The remote program is located at the Internet address *addr*. If *addr->sin_port* is `0`, it is set to port on which the remote program is listening on (the remote rpcbind service is consulted for this information). The parameter **fdp* is a file descriptor, which may be open and bound. If it is `RPC_ANYSOCK`, then this routine opens a new one and sets **fdp*. Refer to the `File Descriptor` section for more information. The UDP transport resends the call message in intervals of `wait` time until a response is received or until the call times out. The total time for the call to time out is specified by `clnt_call()`. See `rpc_clnt_calls(3NSL)`. If successful it returns a client handle, otherwise it returns NULL. The error can be printed using the `clnt_pcreateerror()` routine. See `rpc_clnt_create(3NSL)`.

The user can specify the maximum packet size for sending and receiving by using *sendsz* and *recvsz* arguments for UDP-based RPC messages.

If *addr->sin_port* is `0` and the requested version number *versnum* is not registered with the remote portmap service, it returns a handle if at least a version number for the given program number is registered. The version mismatch is discovered by a `clnt_call()` later (see `rpc_clnt_calls(3NSL)`).

This routine exists for backward compatibility only. `clnt_tli_create()` or `clnt_dg_create()` should be used instead. See `rpc_clnt_create(3NSL)`.

clntudp_create()	This routine creates an RPC client handle for the remote program *prognum*, version *versnum*; the client uses UDP/IP as a transport. The remote program is located at Internet address *addr*. If *addr->sin_port* is `0`, then it is set to actual port that the remote program is listening on. The remote rpcbind service is consulted for this information. The parameter **fdp* is a file descriptor, which may be open and bound; if it is `RPC_ANYSOCK`, then this routine opens a new one and sets **fdp*. Refer to the `File Descriptor` section for more information. The UDP transport resends the call message in intervals of `wait` time until a response is received or until the call times out. The total time for

the call to time out is specified by clnt_call(). See rpc_clnt_calls(3NSL). clntudp_create() returns a client handle on success, otherwise it returns NULL. The error can be printed using the clnt_pcreateerror() routine. See rpc_clnt_create(3NSL).

Since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

This routine exists for backward compatibility only. clnt_create(), clnt_tli_create(), or clnt_dg_create() should be used instead. See rpc_clnt_create(3NSL).

get_myaddress() Places the local system's IP address into *addr*, without consulting the library routines that deal with /etc/hosts. The port number is always set to htons(PMAPPORT).

This routine is only intended for use with the RPC library. It returns the local system's address in a form compatible with the RPC library, and should not be taken as the system's actual IP address. In fact, the *addr* buffer's host address part is actually zeroed. This address may have only local significance and should not be assumed to be an address that can be used to connect to the local system by remote systems or processes.

This routine remains for backward compatibility only. The routine netdir_getbyname() should be used with the name HOST_SELF to retrieve the local system's network address as a *netbuf* structure. See netdir(3NSL).

getrpcport() getrpcport() returns the port number for the version *versnum* of the RPC program *prognum* running on *host* and using protocol *proto*. getrpcport() returns 0 if the RPC system failed to contact the remote portmap service, the program associated with *prognum* is not registered, or there is no mapping between the program and a port.

This routine exists for backward compatibility only. Enhanced functionality is provided by rpcb_getaddr(). See rpcbind(3NSL).

pmaplist()	A user interface to the portmap service, which returns a list of the current RPC program-to-port mappings on the host located at IP address *addr*. This routine can return NULL . The command 'rpcinfo-p' uses this routine.
	This routine exists for backward compatibility only, enhanced functionality is provided by rpcb_getmaps(). See rpcbind(3NSL).
pmap_getport()	A user interface to the portmap service, which returns the port number on which waits a service that supports program *prognum*, version *versnum*, and speaks the transport protocol associated with *protocol*. The value of *protocol* is most likely IPPROTO_UDP or IPPROTO_TCP. A return value of 0 means that the mapping does not exist or that the RPC system failured to contact the remote portmap service. In the latter case, the global variable rpc_createerr contains the RPC status.
	This routine exists for backward compatibility only, enhanced functionality is provided by rpcb_getaddr(). See rpcbind(3NSL).
pmap_rmtcall()	Request that the portmap on the host at IP address **addr* make an RPC on the behalf of the caller to a procedure on that host. **portp* is modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in callrpc() and clnt_call(). See rpc_clnt_calls(3NSL).
	This procedure is only available for the UDP transport.
	If the requested remote procedure is not registered with the remote portmap then no error response is returned and the call times out. Also, no authentication is done.
	This routine exists for backward compatibility only, enhanced functionality is provided by rpcb_rmtcall(). See rpcbind(3NSL).
pmap_set()	A user interface to the portmap service, that establishes a mapping between the triple [*prognum*, *versnum*, *protocol*] and *port* on the machine's portmap service. The value of *protocol* may be IPPROTO_UDP or IPPROTO_TCP. Formerly, the routine failed if the requested *port* was found to be in use. Now, the routine only fails if it finds that *port* is still

	bound. If *port* is not bound, the routine completes the requested registration. This routine returns 1 if it succeeds, 0 otherwise. Automatically done by svc_register().
	This routine exists for backward compatibility only, enhanced functionality is provided by rpcb_set(). See rpcbind(3NSL).
pmap_unset()	A user interface to the portmap service, which destroys all mapping between the triple [*prognum*, *versnum*, *all-protocols*] and *port* on the machine's portmap service. This routine returns one if it succeeds, 0 otherwise.
	This routine exists for backward compatibility only, enhanced functionality is provided by rpcb_unset(). See rpcbind(3NSL).
svc_fds()	A global variable reflecting the RPC service side's read file descriptor bit mask; it is suitable as a parameter to the select() call. This is only of interest if a service implementor does not call svc_run(), but rather does his own asynchronous event processing. This variable is read-only , yet it may change after calls to svc_getreq() or any creation routines. Do not pass its address to select()! Similar to svc_fdset, but limited to 32 descriptors.
	This interface is made obsolete by svc_fdset. See rpc_svc_calls(3NSL).
svc_getcaller()	This routine returns the network address, represented as a struct sockaddr_in, of the caller of a procedure associated with the RPC service transport handle, *xprt*.
	This routine exists for backward compatibility only, and is obsolete. The preferred interface is svc_getrpccaller(). See rpc_svc_reg(3NSL), which returns the address as a struct netbuf.
svc_getreq()	This routine is only of interest if a service implementor does not call svc_run(), but instead implements custom asynchronous event processing. It is called when the select() call has determined that an RPC request has arrived on some RPC file descriptors; *rdfds* is the resultant read file descriptor bit mask. The routine returns when all

	file descriptors associated with the value of *rdfds* have been serviced. This routine is similar to svc_getreqset() but is limited to 32 descriptors.
	This interface is made obsolete by svc_getreqset()
svcfd_create()	Create a service on top of any open and bound descriptor. Typically, this descriptor is a connected file descriptor for a stream protocol. Refer to the File Descriptor section for more information. *sendsz* and *recvsz* indicate sizes for the send and receive buffers. If they are 0, a reasonable default is chosen.
	This interface is made obsolete by svc_fd_create() (see rpc_svc_create(3NSL)).
svcraw_create()	This routine creates an internal, memory-based RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; see clntraw_create(). This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel interference. This routine returns NULL if it fails.
	This routine exists for backward compatibility only, and has the same functionality of svc_raw_create(). See rpc_svc_create(3NSL), which obsoletes it.
svctcp_create()	This routine creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the file descriptor *fd*, which may be RPC_ANYSOCK, in which case a new file descriptor is created. If the file descriptor is not bound to a local TCP port, then this routine binds it to an arbitrary port. Refer to the File Descriptor section for more information. Upon completion, *xprt*->xp_fd is the transport's file descriptor, and *xprt*->xp_port is the transport's port number. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O, users may specify the size of buffers; values of 0 choose suitable defaults.
	This routine exists for backward compatibility only. svc_create(), svc_tli_create(), or svc_vc_create() should be used instead. See rpc_svc_create(3NSL).

svcudp_bufcreate()	This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the file descriptor *fd*. If *fd* is RPC_ANYSOCK then a new file descriptor is created. If the file descriptor is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, *xprt*xp_fd is the transport's file descriptor, and *xprt*->xp_port is the transport's port number. Refer to the File Descriptor section for more information. This routine returns NULL if it fails.

The user specifies the maximum packet size for sending and receiving UDP-based RPC messages by using the *sendsz* and *recvsz* parameters.

This routine exists for backward compatibility only. svc_tli_create(), or svc_dg_create() should be used instead. See rpc_svc_create(3NSL).

svcudp_create()	This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the file descriptor *fd*, which may be RPC_ANYSOCK, in which case a new file descriptor is created. If the file descriptor is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, *xprt*->xp_fd is the transport's file descriptor, and *xprt*->xp_port is the transport's port number. This routine returns NULL if it fails.

Since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

This routine exists for backward compatibility only. svc_create(), svc_tli_create(), or svc_dg_create() should be used instead. See rpc_svc_create(3NSL).

registerrpc()	Register program *prognum*, procedure *procname*, and version *versnum* with the RPC service package. If a request arrives for program *prognum*, version *versnum*, and procedure *procnum*, *procname* is called with a pointer to its parameter(s). *procname* should return a pointer to its static result(s). *inproc* is used to decode the parameters

while *outproc* is used to encode the results. This routine returns 0 if the registration succeeded, −1 otherwise.

svc_run() must be called after all the services are registered.

This routine exists for backward compatibility only, and it is made obsolete by rpc_reg().

svc_register()   Associates *prognum* and *versnum* with the service dispatch procedure, *dispatch*. If *protocol* is 0, the service is not registered with the portmap service. If *protocol* is non-zero, then a mapping of the triple [*prognum*, *versnum*, *protocol*] to *xprt*->xp_port is established with the local portmap service (generally *protocol* is 0, IPPROTO_UDP or IPPROTO_TCP). The procedure *dispatch* has the following form:

```
dispatch(struct svc_req *request, SVCXPRT *xprt);
```

The svc_register() routine returns one if it succeeds, and 0 otherwise.

This routine exists for backward compatibility only. Enhanced functionality is provided by svc_reg().

svc_unregister()   Remove all mapping of the double [*prognum*, *versnum*] to dispatch routines, and of the triple [*prognum*, *versnum*, *all-protocols*] to port number from portmap.

This routine exists for backward compatibility. Enhanced functionality is provided by svc_unreg().

xdr_authunix_parms()   Used for describing UNIX credentials. This routine is useful for users who wish to generate these credentials without using the RPC authentication package.

This routine exists for backward compatibility only, and is made obsolete by xdr_authsys_parms(). See rpc_xdr(3NSL).

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also   keyserv(1M), rpcbind(1M), rpcinfo(1M), netdir(3NSL), netdir_getbyname(3NSL), rpc(3NSL), rpc_clnt_auth(3NSL), rpc_clnt_calls(3NSL), rpc_clnt_create(3NSL), rpc_svc_calls(3NSL), rpc_svc_create(3NSL), rpc_svc_err(3NSL), rpc_svc_reg(3NSL), rpc_xdr(3NSL), rpcbind(3NSL), secure_rpc(3NSL), select(3C), xdr_authsys_parms(3NSL), libnsl(3LIB), attributes(5)

Notes   These interfaces are unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

Name rpc_svc_calls, svc_dg_enablecache, svc_done, svc_exit, svc_fdset, svc_freeargs, svc_getargs, svc_getreq_common, svc_getreq_poll, svc_getreqset, svc_getrpccaller, svc_max_pollfd, svc_pollfd, svc_run, svc_sendreply, svc_getcallerucred, svc_fd_negotiate_ucred – library routines for RPC servers

Synopsis cc [*flag*...] *file*... -lnsl [*library*...]
```
#include <rpc/rpc.h>

int svc_dg_enablecache(SVCXPRT *xprt, const uint_t cache_size);

int svc_done(SVCXPRT *xprt);

void svc_exit(void);

void svc_fd_negotiate_ucred(int fd);

bool_t svc_freeargs(const SVCXPRT *xprt, const txdrproc_t inproc,
     caddr_t in);

bool_t svc_getargs(const SVCXPRT *xprt, const xdrproc_t inproc,
     caddr_t in);

int svc_getcallerucred(const SVCXPRT *xprt, ucred_t **ucred);

void svc_getreq_common(const int fd);

void svc_getreqset(fd_set *rdfds);

void svc_getreq_poll(struct pollfd *pfdp, const int pollretval);

struct netbuf *svc_getrpccaller(const SVCXPRT *xprt);

void svc_run(void);

bool_t svc_sendreply(const SVCXPRT *xprt, const xdrproc_t outproc,
     caddr_t outint svc_max_pollfd;
fd_set svc_fdset;
pollfd_t *svc_pollfd;
```

Description These routines are part of the RPC library which allows C language programs to make procedure calls on other machines across the network.

These routines are associated with the server side of the RPC mechanism. Some of them are called by the server side dispatch function. Others, such as svc_run(), are called when the server is initiated.

Because the service transport handle SVCXPRT contains a single data area for decoding arguments and encoding results, the structure cannot freely be shared between threads that call functions to decode arguments and encode results. When a server is operating in the Automatic or User MT modes, however, a copy of this structure is passed to the service dispatch procedure in order to enable concurrent request processing. Under these circumstances, some routines which would otherwise be Unsafe, become Safe. These are

marked as such. Also marked are routines that are Unsafe for multithreaded applications, and are not to be used by such applications. See rpc(3NSL) for the definition of the SVCXPRT data structure.

The svc_dg_enablecache() function allocates a duplicate request cache for the service endpoint *xprt*, large enough to hold *cache_size* entries. Once enabled, there is no way to disable caching. The function returns 1 if space necessary for a cache of the given size was successfully allocated, and 0 otherwise. This function is Safe in multithreaded applications.

The svc_done() function frees resources allocated to service a client request directed to the service endpoint *xprt*. This call pertains only to servers executing in the User MT mode. In the User MT mode, service procedures must invoke this call before returning, either after a client request has been serviced, or after an error or abnormal condition that prevents a reply from being sent. After svc_done() is invoked, the service endpoint *xprt* should not be referenced by the service procedure. Server multithreading modes and parameters can be set using the rpc_control() call. This function is Safe in multithreaded applications. It will have no effect if invoked in modes other than the User MT mode.

The svc_exit() function when called by any of the RPC server procedures or otherwise, destroys all services registered by the server and causes svc_run() to return. If RPC server activity is to be resumed, services must be reregistered with the RPC library either through one of the rpc_svc_create(3NSL) functions, or using xprt_register(3NSL). The svc_exit() function has global scope and ends all RPC server activity.

The svc_freeargs() function macro frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using svc_getargs(). This routine returns TRUE if the results were successfully freed, and FALSE otherwise. This function macro is Safe in multithreaded applications utilizing the Automatic or User MT modes.

The svc_getargs() function macro decodes the arguments of an RPC request associated with the RPC service transport handle *xprt*. The parameter *in* is the address where the arguments will be placed; *inproc* is the XDR routine used to decode the arguments. This routine returns TRUE if decoding succeeds, and FALSE otherwise. This function macro is Safe in multithreaded applications utilizing the Automatic or User MT modes.

The svc_getreq_common() function is called to handle a request on a file descriptor.

The svc_getreq_poll() function is only of interest if a service implementor does not call svc_run(), but instead implements custom asynchronous event processing. It is called when poll(2) has determined that an RPC request has arrived on some RPC file descriptors; *pollretval* is the return value from poll(2) and *pfdp* is the array of *pollfd* structures on which the poll(2) was done. It is assumed to be an array large enough to contain the maximal number of descriptors allowed. The svc_getreq_poll() function macro is Unsafe in multithreaded applications.

The svc_getreqset() function is only of interest if a service implementor does not call svc_run(), but instead implements custom asynchronous event processing. It is called when

select(3C) has determined that an RPC request has arrived on some RPC file descriptors; *rdfds* is the resultant read file descriptor bit mask. The routine returns when all file descriptors associated with the value of *rdfds* have been serviced. This function macro is Unsafe in multithreaded applications.

The svc_getrpccaller() function is the approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle *xprt*. This function macro is Safe in multithreaded applications.

The svc_run() function never returns. In single-threaded mode, the function waits for RPC requests to arrive. When an RPC request arrives, the svc_run() function calls the appropriate service procedure. This procedure is usually waiting for the poll(2) library call to return.

Applications that execute in the Automatic or the User MT mode should invoke the svc_run() function exactly once. In the Automatic MT mode, the svc_run() function creates threads to service client requests. In the User MT mode, the function provides a framework for service developers to create and manage their own threads for servicing client requests.

The svc_fdset global variable reflects the RPC server's read file descriptor bit mask. This is only of interest if service implementors do not call svc_run(), but rather do their own asynchronous event processing. This variable is read-only may change after calls to svc_getreqset() or after any creation routine. Do not pass its address to select(3C). Instead, pass the address of a copy. multithreaded applications executing in either the Automatic MT mode or the user MT mode should never read this variable. They should use auxiliary threads to do asynchronous event processing. The svc_fdset variable is limited to 1024 file descriptors and is considered obsolete. Use of svc_pollfd is recommended instead.

The svc_pollfd global variable points to an array of pollfd_t structures that reflect the RPC server's read file descriptor array. This is only of interest if service service implementors do not call svc_run() but rather do their own asynchronous event processing. This variable is read-only, and it may change after calls to svc_getreg_poll() or any creation routines. Do no pass its address to poll(2). Instead, pass the address of a copy. By default, svc_pollfd is limited to 1024 entries. Use rpc_control(3NSL) to remove this limitation. multithreaded applications executing in either the Automatic MT mode or the user MT mode should never be read this variable. They should use auxiliary threads to do asynchronous event processing.

The svc_max_pollfd global variable contains the maximum length of the svc_pollfd array. This variable is read-only, and it may change after calls to svc_getreg_poll() or any creation routines.

The svc_sendreply() function is called by an RPC service dispatch routine to send the results of a remote procedure call. The *xprt* parameter is the transport handle of the request. The *outproc* parameter is the XDR routine used to encode the results. The *out* parameter is the address of the results. This routine returns TRUE if it succeeds, FALSE otherwise. The svc_sendreply() function macro is Safe in multithreaded applications that use the Automatic or the User MT mode.

The svc_fd_negotiate_ucred() function is called by an RPC server to inform the underlying transport that the function wishes to receive ucreds for local calls, including those over IP transports.

The svc_getcallerucred() function attempts to retrieve the ucred_t associated with the caller. The function returns 0 when successful and -1 when not.

When successful, the svc_getcallerucred() function stores the pointer to a freshly allocated ucred_t in the memory location pointed to by the *ucred* argument if that memory location contains the null pointer. If the memory location is non-null, the function reuses the existing ucred_t. When *ucred* is no longer needed, a credential allocated by svc_getcallerucred() should be freed with ucred_free(3C).

Attributes See attributes(5) for descriptions of attribute types and values.

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	See below.

The svc_fd_negotiate_ucred(), svc_dg_enablecache(), svc_getrpccaller(), and svc_getcallerucred() functions are Safe in multithreaded applications. The svc_freeargs(), svc_getargs(), and svc_sendreply() functions are Safe in multithreaded applications that use the Automatic or the User MT mode. The svc_getreq_common(), svc_getreqset(), and svc_getreq_poll() functions are Unsafe in multithreaded applications and should be called only from the main thread.

See Also rpcgen(1), poll(2), getpeerucred(3C), rpc(3NSL), rpc_control(3NSL), rpc_svc_create(3NSL), rpc_svc_err(3NSL), rpc_svc_reg(3NSL), select(3C), ucred_free(3C), xprt_register(3NSL), attributes(5)

Name rpc_svc_create, svc_control, svc_create, svc_destroy, svc_dg_create, svc_fd_create, svc_raw_create, svc_tli_create, svc_tp_create, svc_vc_create, svc_door_create – server handle creation routines

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

```
bool_t svc_control(SVCXPRT *svc, const uint_t req, void *info);

int svc_create(const void (*dispatch)const struct svc_req *,
    const SVCXPRT *, const rpcprog_t prognum, const rpcvers_t versnum,
    const char *nettype);

void svc_destroy(SVCXPRT *xprt);

SVCXPRT *svc_dg_create(const int fildes, const uint_t sendsz,
    const uint_t recvsz);

SVCXPRT *svc_fd_create(const int fildes, const uint_t sendsz,
    const uint_t recvsz);

SVCXPRT *svc_raw_create(void)

SVCXPRT *svc_tli_create(const int fildes, const struct netconfig *netconf,
    const struct t_bind *bind_addr, const uint_t sendsz,
    const uint_t recvsz);

SVCXPRT *svc_tp_create(const void (*dispatch)
    const struct svc_req *, const SVCXPRT *), const rpcprog_t prognum,
    const rpcvers_t versnum, const struct netconfig *netconf);

SVCXPRT *svc_vc_create(const int fildes, const uint_t sendsz,
    const uint_t recvsz);

SVCXPRT *svc_door_create(void (*dispatch)(struct svc_req *, SVCXPRT *),
    const rpcprog_t prognum, const rpcvers_t versnum,
    const uint_t sendsz);
```

Description These routines are part of the RPC library which allows C language programs to make procedure calls on servers across the network. These routines deal with the creation of service handles. Once the handle is created, the server can be invoked by calling svc_run().

Routines See [rpc(3NSL)](#) for the definition of the SVCXPRT data structure.

svc_control()
    A function to change or retrieve information about a service object. *req* indicates the type of operation and *info* is a pointer to the information. The supported values of *req*, their argument types, and what they do are:

SVCGET_VERSQUIET
    If a request is received for a program number served by this server but the version number is outside the range registered with the server, an RPC_PROGVERSMISMATCH error will normally be returned. *info* should be a pointer to an integer. Upon successful completion of the SVCGET_VERSQUIET request, *info* contains an integer which describes

the server's current behavior: 0 indicates normal server behavior, that is, an RPC_PROGVERSMISMATCH error will be returned. 1 indicates that the out of range request will be silently ignored.

SVCSET_VERSQUIET

If a request is received for a program number served by this server but the version number is outside the range registered with the server, an RPC_PROGVERSMISMATCH error will normally be returned. It is sometimes desirable to change this behavior. *info* should be a pointer to an integer which is either 0, indicating normal server behavior and an RPC_PROGVERSMISMATCH error will be returned, or 1, indicating that the out of range request should be silently ignored.

SVCGET_XID

Returns the transaction ID of connection−oriented and connectionless transport service calls. The transaction ID assists in uniquely identifying client requests for a given RPC version, program number, procedure, and client. The transaction ID is extracted from the service transport handle *svc*. *info* must be a pointer to an unsigned long. Upon successful completion of the SVCGET_XID request, *info* contains the transaction ID. Note that rendezvous and raw service handles do not define a transaction ID. Thus, if the service handle is of rendezvous or raw type, and the request is of type SVCGET_XID, svc_control() will return FALSE. Note also that the transaction ID read by the server can be set by the client through the suboption CLSET_XID in clnt_control(). See clnt_create(3NSL)

SVCSET_RECVERRHANDLER

Attaches or detaches a disconnection handler to the service handle, *svc*, that will be called when a transport error arrives during the reception of a request or when the server is waiting for a request and the connection shuts down. This handler is only useful for a connection oriented service handle.

info contains the address of the error handler to attach, or NULL to detach a previously defined one. The error handler has two arguments. It has a pointer to the erroneous service handle. It also has an integer that indicates if the full service is closed (when equal to zero), or that only one connection on this service is closed (when not equal to zero).

```
void handler (const SVCXPRT *svc, const bool_t isAConnection);
```

With the service handle address, *svc*, the error handler is able to detect which connection has failed and to begin an error recovery process. The error handler can be called by multiple threads and should be implemented in an MT-safe way.

SVCGET_RECVERRHANDLER

Upon successful completion of the SVCGET_RECVERRHANDLER request, *info* contains the address of the handler for receiving errors. Upon failure, *info* contains NULL.

SVCSET_CONNMAXREC
>   Set the maximum record size (in bytes) and enable non-blocking mode for this service
>   handle. Value can be set and read for both connection and non-connection oriented
>   transports, but is silently ignored for the non-connection oriented case. The *info*
>   argument should be a pointer to an int.

SVCGET_CONNMAXREC
>   Get the maximum record size for this service handle. Zero means no maximum in effect
>   and the connection is in blocking mode. The result is not significant for non-connection
>   oriented transports. The *info* argument should be a pointer to an int.

This routine returns TRUE if the operation was successful. Otherwise, it returns false.

svc_create()
>   svc_create() creates server handles for all the transports belonging to the class *nettype*.
>
>   *nettype* defines a class of transports which can be used for a particular application. The
>   transports are tried in left to right order in NETPATH variable or in top to bottom order in the
>   netconfig database. If *nettype* is NULL, it defaults to netpath.
>
>   svc_create() registers itself with the rpcbind service (see rpcbind(1M)). *dispatch* is
>   called when there is a remote procedure call for the given *prognum* and *versnum*; this
>   requires calling svc_run() (see svc_run() in rpc_svc_reg(3NSL)). If svc_create()
>   succeeds, it returns the number of server handles it created, otherwise it returns 0 and an
>   error message is logged.

svc_destroy()
>   A function macro that destroys the RPC service handle *xprt*. Destruction usually involves
>   deallocation of private data structures, including *xprt* itself. Use of *xprt* is undefined after
>   calling this routine.

svc_dg_create()
>   This routine creates a connectionless RPC service handle, and returns a pointer to it. This
>   routine returns NULL if it fails, and an error message is logged. *sendsz* and *recvsz* are
>   parameters used to specify the size of the buffers. If they are 0, suitable defaults are chosen.
>   The file descriptor *fildes* should be open and bound. The server is not registered with
>   rpcbind(1M).
>
>   Warning: since connectionless-based RPC messages can only hold limited amount of
>   encoded data, this transport cannot be used for procedures that take large arguments or
>   return huge results.

svc_fd_create()
>   This routine creates a service on top of an open and bound file descriptor, and returns the
>   handle to it. Typically, this descriptor is a connected file descriptor for a
>   connection-oriented transport. *sendsz* and *recvsz* indicate sizes for the send and receive
>   buffers. If they are 0, reasonable defaults are chosen. This routine returns NULL if it fails, and
>   an error message is logged.

svc_raw_create()
This routine creates an RPC service handle and returns a pointer to it. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; (see clnt_raw_create() in rpc_clnt_create(3NSL)). This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel and networking interference. This routine returns NULL if it fails, and an error message is logged.

Note: svc_run() should not be called when the raw interface is being used.

svc_tli_create()
This routine creates an RPC server handle, and returns a pointer to it. *fildes* is the file descriptor on which the service is listening. If *fildes* is RPC_ANYFD, it opens a file descriptor on the transport specified by *netconf*. If the file descriptor is unbound and *bindaddr* is non-null *fildes* is bound to the address specified by *bindaddr*, otherwise *fildes* is bound to a default address chosen by the transport. In the case where the default address is chosen, the number of outstanding connect requests is set to 8 for connection-oriented transports. The user may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of 0 choose suitable defaults. This routine returns NULL if it fails, and an error message is logged. The server is not registered with the rpcbind(1M) service.

svc_tp_create()
svc_tp_create() creates a server handle for the network specified by *netconf*, and registers itself with the rpcbind service. *dispatch* is called when there is a remote procedure call for the given *prognum* and *versnum*; this requires calling svc_run(). svc_tp_create() returns the service handle if it succeeds, otherwise a NULL is returned and an error message is logged.

svc_vc_create()
This routine creates a connection-oriented RPC service and returns a pointer to it. This routine returns NULL if it fails, and an error message is logged. The users may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of 0 choose suitable defaults. The file descriptor *fildes* should be open and bound. The server is not registered with the rpcbind(1M) service.

svc_door_create()
This routine creates an RPC server handle over doors and returns a pointer to it. Doors is a transport mechanism that facilitates fast data transfer between processes on the same machine. for the given program The user may set the size of the send buffer with the parameter *sendsz*. If *sendsz* is 0, the corresponding default buffer size is 16 Kbyte. If successful, the svc_door_create() routine returns the service handle. Otherwise it returns NULL and sets a value for rpc_createerr. The server is not registered with rpcbind(1M). The SVCSET_CONNMAXREC and SVCGET_CONNMAXREC svc_control() requests can be used to set and change the maximum allowed request size for the doors transport.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	All
Availability	system/library
Interface Stability	Committed
MT-Level	MT-Safe

See Also   rpcbind(1M), rpc(3NSL), rpc_clnt_create(3NSL), rpc_svc_calls(3NSL), rpc_svc_err(3NSL), rpc_svc_reg(3NSL), attributes(5)

Name rpc_svc_err, svcerr_auth, svcerr_decode, svcerr_noproc, svcerr_noprog, svcerr_progvers, svcerr_systemerr, svcerr_weakauth – library routines for server side remote procedure call errors

Description These routines are part of the RPC library which allows C language programs to make procedure calls on other machines across the network.

These routines can be called by the server side dispatch function if there is any error in the transaction with the client.

Routines See rpc(3NSL) for the definition of the SVCXPRT data structure.

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

void svcerr_auth(const SVCXPRT *xprt, const enum auth_stat why);
　Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

void svcerr_decode(const SVCXPRT *xprt);
　Called by a service dispatch routine that cannot successfully decode the remote parameters (see svc_getargs() in rpc_svc_reg(3NSL)).

void svcerr_noproc(const SVCXPRT *xprt);
　Called by a service dispatch routine that does not implement the procedure number that the caller requests.

void svcerr_noprog(const SVCXPRT *xprt);
　Called when the desired program is not registered with the RPC package. Service implementors usually do not need this routine.

void svcerr_progvers(const SVCXPRT *xprt, const rpcvers_t low_vers, const rpcvers_t high_vers);
　Called when the desired version of a program is not registered with the RPC package. low_vers is the lowest version number, and high_vers is the highest version number. Service implementors usually do not need this routine.

void svcerr_systemerr(const SVCXPRT *xprt);
　Called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

void svcerr_weakauth(const SVCXPRT *xprt);
　Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient (but correct) authentication parameters. The routine calls svcerr_auth(xprt, AUTH_TOOWEAK).

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also rpc(3NSL), rpc_svc_calls(3NSL), rpc_svc_create(3NSL), rpc_svc_reg(3NSL), attributes(5)

Name rpc_svc_input, svc_add_input, svc_remove_input – declare or remove a callback on a file descriptor

Synopsis #include <rpc/rpc.h>

```
typedef void (*svc_callback_t)(svc_input_id_t id, int fd,
    unsigned int events, void *cookie);

svc_input_id_t svc_add_input(int fd, unsigned int revents,
    svc_callback_t callback, void *cookie);

int svc_remove_input(svc_input_t id);
```

Description The following RPC routines are used to declare or remove a callback on a file descriptor.

Routines See rpc(3NSL) for the definition of the SVCXPRT data structure.

svc_add_input()   This function is used to register a *callback* function on a file
                              descriptor, *fd*. The file descriptor, *fd*, is the first parameter to be
                              passed to svc_add_input(). This *callback* function will be
                              automatically called if any of the events specified in the *events*
                              parameter occur on this descriptor. The *events* parameter is used to
                              specify when the callback is invoked. This parameter is a mask of
                              poll events to which the user wants to listen. See poll(2) for further
                              details of the events that can be specified.

                              The callback to be invoked is specified using the *callback* parameter.
                              The *cookie* parameter can be used to pass any data to the *callback*
                              function. This parameter is a user-defined value which is passed as
                              an argument to the *callback* function, and it is not used by the Sun
                              RPC library itself.

                              Several callbacks can be registered on the same file descriptor as
                              long as each callback registration specifies a separate set of event
                              flags.

                              The *callback* function is called with the registration *id*, the *fd* file
                              descriptor, an *revents* value, which is a bitmask of all events
                              concerning the file descriptor, and the *cookie* user-defined value.

                              Upon successful completion, the function returns a unique
                              identifier for this registration, that can be used later to remove this
                              callback. Upon failure, -1 is returned and errno is set to indicate
                              the error.

                              The svc_add_input() function will fail if:

                              EINVAL   The *fd* or *events* parameters are invalid.

	EEXIST	A callback is already registered to the file descriptor with one of the specified events.
	ENOMEM	Memory is exhausted.

svc_remove_input()   This function is used to unregister a callback function on a file descriptor, *fd*. The *id* parameter specifies the registration to be removed.

Upon successful completion, the function returns zero. Upon failure, -1 is returned and errno is set to indicate the error.

The svc_remove_input() function will fail if:

EINVAL   The *id* parameter is invalid.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	All
Availability	system/library
Interface Stability	Committed
MT-Level	MT-Safe

See Also   poll(2), rpc(3NSL), attributes(5)

Name rpc_svc_reg, rpc_reg, svc_reg, svc_unreg, svc_auth_reg, xprt_register, xprt_unregister –
library routines for registering servers

Description These routines are a part of the RPC library which allows the RPC servers to register
themselves with rpcbind() (see rpcbind(1M)), and associate the given program and version
number with the dispatch function. When the RPC server receives a RPC request, the library
invokes the dispatch routine with the appropriate arguments.

Routines See rpc(3NSL) for the definition of the SVCXPRT data structure.

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

bool_t rpc_reg(const rpcprog_t *prognum*, const rpcvers_t *versnum*, const rpcproc_t *procnum*,
char * (*procname*)( ), const xdrproc_t *inproc*, const xdrproc_t *outproc*, const char *nettype*);
  Register program *prognum*, procedure *procname*, and version *versnum* with the RPC
  service package. If a request arrives for program *prognum*, version *versnum*, and procedure
  procnum, *procname* is called with a pointer to its parameter(s); *procname* should return a
  pointer to its static result(s). The *arg* parameter to *procname* is a pointer to the (decoded)
  procedure argument. *inproc* is the XDR function used to decode the parameters while
  outproc is the XDR function used to encode the results. Procedures are registered on all
  available transports of the class *nettype*. See rpc(3NSL). This routine returns 0 if the
  registration succeeded, −1 otherwise.

int svc_reg(const SVCXPRT *xprt*, const rpcprog_t *prognum*, const rpcvers_t *versnum*, const
void (*dispatch*)( ), const struct netconfig *netconf*);
  Associates *prognum* and *versnum* with the service dispatch procedure, *dispatch*. If *netconf*
  is NULL, the service is not registered with the rpcbind service. For example, if a service has
  already been registered using some other means, such as inetd (see inetd(1M)), it will not
  need to be registered again. If *netconf* is non-zero, then a mapping of the triple [*prognum*,
  versnum, *netconf->*] to *xprt-> xp_ltaddr* is established with the local rpcbind service.

  The svc_reg() routine returns 1 if it succeeds, and 0 otherwise.

void svc_unreg(const rpcprog_t *prognum*, const rpcvers_t *versnum*);
  Remove from the rpcbind service, all mappings of the triple [*prognum*, *versnum*,
  all-transports] to network address and all mappings within the RPC service package of the
  double [*prognum*, *versnum*] to dispatch routines.

int svc_auth_reg(const int *cred_flavor*, const enum auth_stat (*handler)( ));
  Registers the service authentication routine *handler* with the dispatch mechanism so that it
  can be invoked to authenticate RPC requests received with authentication type *cred_flavor*.
  This interface allows developers to add new authentication types to their RPC applications
  without needing to modify the libraries. Service implementors usually do not need this
  routine.

  Typical service application would call svc_auth_reg() after registering the service and
  prior to calling svc_run(). When needed to process an RPC credential of type *cred_flavor*,
  the *handler* procedure will be called with two parameters (struct svc_req *rqst*, struct

rpc_msg *_msg_) and is expected to return a valid enum auth_stat value. There is no provision to change or delete an authentication handler once registered.

The svc_auth_reg() routine returns 0 if the registration is successful, 1 if _cred_flavor_ already has an authentication handler registered for it, and −1 otherwise.

void xprt_register(const SVCXPRT *_xprt_);
After RPC service transport handle _xprt_ is created, it is registered with the RPC service package. This routine modifies the global variable svc_fdset (see rpc_svc_calls(3NSL)). Service implementors usually do not need this routine.

void xprt_unregister(const SVCXPRT *_xprt_);
Before an RPC service transport handle _xprt_ is destroyed, it unregisters itself with the RPC service package. This routine modifies the global variable svc_fdset (see rpc_svc_calls(3NSL)). Service implementors usually do not need this routine.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also inetd(1M), rpcbind(1M), rpc(3NSL), rpc_svc_calls(3NSL), rpc_svc_create(3NSL), rpc_svc_err(3NSL), rpcbind(3NSL), select(3C), attributes(5)

Name rpc_xdr, xdr_accepted_reply, xdr_authsys_parms, xdr_callhdr, xdr_callmsg, xdr_opaque_auth, xdr_rejected_reply, xdr_replymsg – XDR library routines for remote procedure calls

Synopsis bool_t xdr_accepted_reply(XDR *xdrs, const struct accepted_reply *ar);

bool_t xdr_authsys_parms(XDR *xdrs, struct authsys_parms *aupp);

void xdr_callhdr(XDR *xdrs, struct rpc_msg *chdr);

bool_t xdr_callmsg(XDR *xdrs, struct rpc_msg *cmsg);

bool_t xdr_opaque_auth(XDR *xdrs, struct opaque_auth *ap);

bool_t xdr_rejected_reply(XDR *xdrs, const struct rejected_reply *rr);

bool_t xdr_replymsg(XDR *xdrs, const struct rpc_msg *rmsg);

Description These routines are used for describing the RPC messages in XDR language. They should normally be used by those who do not want to use the RPC package directly. These routines return TRUE if they succeed, FALSE otherwise.

Routines See rpc(3NSL) for the definition of the XDR data structure.

    #include <rpc/rpc.h>

xdr_accepted_reply()   Used to translate between RPC reply messages and their external representation. It includes the status of the RPC call in the XDR language format. In the case of success, it also includes the call results.

xdr_authsys_parms()   Used for describing UNIX operating system credentials. It includes machine-name, uid, gid list, etc.

xdr_callhdr()   Used for describing RPC call header messages. It encodes the static part of the call message header in the XDR language format. It includes information such as transaction ID, RPC version number, program and version number.

xdr_callmsg()   Used for describing RPC call messages. This includes all the RPC call information such as transaction ID, RPC version number, program number, version number, authentication information, etc. This is normally used by servers to determine information about the client RPC call.

xdr_opaque_auth()   Used for describing RPC opaque authentication information messages.

xdr_rejected_reply()   Used for describing RPC reply messages. It encodes the rejected RPC message in the XDR language format. The message could be rejected either because of version number mis-match or because of authentication errors.

xdr_replymsg()   Used for describing RPC reply messages. It translates between the RPC reply message and its external representation. This reply could be either an acceptance, rejection or NULL.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also rpc(3NSL), xdr(3NSL), attributes(5)

Name rstat, havedisk – get performance data from remote kernel

Synopsis cc [*flag* ...] *file* ... -lrpcsvc [*library* ...]
#include <rpc/rpc.h>
#include <rpcsvc/rstat.h>

enum clnt_stat rstat(char *_host_, struct statstime *_statp_);

int havedisk(char *_host_);

Protocol /usr/include/rpcsvc/rstat.x

Description These routines require that the rpc.rstatd(1M) daemon be configured and available on the remote system indicated by *host*. The rstat() protocol is used to gather statistics from remote kernel. Statistics will be available on items such as paging, swapping, and cpu utilization.

rstat() fills in the statstime structure *statp* for *host*. *statp* must point to an allocated statstime structure. rstat() returns RPC_SUCCESS if it was successful; otherwise a enum clnt_stat is returned which can be displayed using clnt_perrno(3NSL).

havedisk() returns 1 if *host* has disk, 0 if it does not, and −1 if this cannot be determined.

The following XDR routines are available in librpcsvc:

xdr_statstime
xdr_statsvar

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also rup(1), rpc.rstatd(1M), rpc_clnt_calls(3NSL), attributes(5)

Name rusers, rnusers – return information about users on remote machines

Synopsis cc [*flag* ...] *file* ... -lrpcsvc [*library* ...]
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

enum clnt_stat rusers(char *_host_, struct utmpidlearr *_up_);

int rnusers(char *_host_);

Protocol /usr/include/rpcsvc/rusers.x

Description These routines require that the rpc.rusersd(1M) daemon be configured and available on the remote system indicated by *host*. The rusers() protocol is used to retrieve information about users logged in on the remote system.

rusers() fills the utmpidlearr structure with data about *host*, and returns 0 if successful. *up* must point to an allocated utmpidlearr structure. If rusers() returns successful it will have allocated data structures within the *up* structure, which should be freed with xdr_free(3NSL) when you no longer need them:

xdr_free(xdr_utimpidlearr, up);

On error, the returned value can be interpreted as an enum clnt_stat and can be displayed with clnt_perror(3NSL) or clnt_sperrno(3NSL).

See the header <rpcsvc/rusers.h> for a definition of struct utmpidlearr.

rnusers() returns the number of users logged on to *host* (−1 if it cannot determine that number).

The following XDR routines are available in librpcsvc:

xdr_utmpidlearr

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also rusers(1), rpc.rusersd(1M), rpc_clnt_calls(3NSL), xdr_free(3NSL), attributes(5)

Name rwall – write to specified remote machines

Synopsis cc [*flag* ...] *file* ... -lrpcsvc [*library* ...]
#include <rpc/rpc.h>
#include <rpcsvc/rwall.h>

enum clnt_stat rwall(char *_host_, char *_msg_);

Protocol /usr/include/rpcsvc/rwall.x

Description These routines require that the rpc.rwalld(1M) daemon be configured and available on the remote system indicated by *host*.

rwall() executes wall(1M) on *host*. The rpc.rwalld process on *host* prints *msg* to all users logged on to that system. rwall() returns RPC_SUCCESS if it was successful; otherwise a enum clnt_stat is returned which can be displayed using clnt_perrno(3NSL).

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also rpc.rwalld(1M), wall(1M), rpc_clnt_calls(3NSL), attributes(5)

Name sasl_authorize_t – the SASL authorization callback

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/sasl.h>

int sasl_authorize_t(sasl_conn_t *conn*, const char *requested_user*,
    unsigned *alen*, const char* *auth_identity*, unsigned *rlen*,
    const char *def_realm*, unsigned *urlen*, struct propctx *propctx*);

Description sasl_authorize_t() is a typedef function prototype that defines the interface associated with the SASL_CB_PROXY_POLICY callback.

Use the sasl_authorize_t() interface to check whether the authorized user *auth_identity* can act as the user *requested_user*. For example, the user root may want to authenticate with root's credentials but as the user tmartin, with all of tmartin's rights, not root's. A server application should be very careful when it determines which users may proxy as other users.

Parameters *conn* The SASL connection context.

requested_user The identity or username to authorize. *requested_user* is null-terminated.

rlen The length of *requested_user*.

auth_identity The identity associated with the secret. *auth_identity* is null-terminated.

alen The length of *auth_identity*.

default_realm The default user realm as passed to sasl_server_new(3SASL).

ulren The length of the default realm

propctx Auxiliary properties

Return Values Like other SASL callback functions, sasl_authorize_t() returns an integer that corresponds to a SASL error code. See <sasl.h> for a complete list of SASL error codes.

Errors SASL_OK The call to sasl_authorize_t() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also   sasl_errors(3SASL), sasl_server_new(3SASL), attributes(5)

Name   sasl_auxprop, prop_new, prop_dup, prop_request, prop_get, prop_getnames, prop_clear, prop_erase, prop_dispose, prop_format, prop_set, prop_setvals – SASL auxilliary properties

Synopsis   cc [*flag* ...] *file* ... -lsasl   [*library* ...]
```
#include <sasl/prop.h>

struct propctx *prop_new(unsigned estimate);

int prop_dup(struct propctx *src_ctx, struct propctx *dst_ctx

int prop_request(struct propctx *ctx, const char **names

const struct propval *prop_get(struct propctx *ctx

int prop_getnames(struct propctx *ctx, const char **names,
    struct propval *vals

void prop_clear(struct propctx *ctx, int requests

void prop_erase(struct propctx *ctx, const char *name

void prop_dispose(struct propctx *ctx

int prop_format(struct propctx *ctx, const char *sep, int seplen,
    char *outbuf, unsigned outmax, unsigned *outlen

int prop_set(struct propctx *ctx, const char *name, const char *value,
    int vallen

int prop_setvals(struct propctx *ctx, const char *name,
    const char **values
```

Description   The SASL auxilliary properties are used to obtain properties from external sources during the authentication process. For example, a mechanizm might need to query an LDAP server to obtain the authentication secret. The application probably needs other information from the LDAP server as well, such as the home directory of the UID. The auxilliary property interface allows the two to cooperate and results in only a single query against the property sources.

Property lookups take place directly after user canonicalization occurs. Therefore, all request should be registered with the context before user canonicalization occurs. Requests can calso be registered by using the sasl_auxprop_request(3SASL) function. Most of the auxilliary property functions require a property context that can be obtained by calling sasl_auxprop_getctx(3SASL).

prop_new()   The prop_new() function creates a new property context. It is unlikely that application developers will use this call.

prop_dup()   The prop_dup() function duplicates a given property context.

prop_request()   The prop_request() function adds properties to the request list of a given context.

prop_get()	The prop_get() function returns a null-terminated array of struct propval from the given context.
prop_getnames()	The prop_getnames() function fills in an array of struct propval based on a list of property names. The vals array is at least as long as the names array. The values that are filled in by this call persist until the next call on the context to prop_request(), prop_clear(), or prop_dispose(). If a name specified was never requested, then its associated values entry will be set to NULL.
	The prop_getnames() function returns the number of matching properties that were found or a SASL error code.
prop_clear()	The prop_clear() function clears *values* and *requests* from a property context. If the value of *requests* is 1, then *requests* is cleared. Otherwise, the value of *requests* is 0.
prop_erase()	The prop_erase() function securely erases the value of a property. *name* is the name of the property to erase.
prop_dispose()	The prop_dispose() function disposes of a property context and nullifies the pointer.
prop_format()	The prop_format() function formats the requested property names into a string. The prop_format() function is not intended to be used by the application. The function is used only by auxprop plug-ins.
prop_set()	The prop_set() functions adds a property value to the context. The prop_set() function is used only by auxprop plug-ins.
prop_setvals()	The prop_setvals() function adds multiple values to a single property. The prop_setvals() function is used only by auxprop plug-ins.
Parameters *conn*	The sasl_conn_t for which the request is being made
ctx	The property context.
estimate	The estimate of the total storage needed for requests and responses. The library default is implied by a value of 0.
names	The null-terminated array of property names. *names* must persist until the requests are cleared or the context is disposed of with a call to prop_dispose().
name	The name of the property.
	For prop_set(), *name* is the named of the property to receive the new value, or NULL. The value will be added to the same property as the last call to either prop_set() or prop_setvals().
outbuf	The caller-allocated buffer of length *outmax* that the resulting string, including the NULL terminator, will be placed in.
outlen	If non-NULL, contains the length of the resulting sting, excluding the NULL terminator.

outmax   The maximum length of the output buffer, including the NULL terminator.

requests   The request list for a given context.

sep   The separator to use for the string.

seplen   The length of the separator. The the values is less than 0, then strlen will be used
                as *sep*.

vallen   The length of the property.

vals   The value string.

value   A value for the property of length *vallen*.

values   A null-terminated array of values to be added to the property.

Errors The sasl_auxprop() functions that return an int will return a SASL error code. See
sasl_errors(3SASL). Those sasl_auxprop() functions that return a pointer will return a
valid pointer upon success and return NULL upon failure.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT-Safe

See Also sasl_auxprop_getctx(3SASL), sasl_auxprop_request(3SASL), sasl_errors(3SASL),
attributes(5)

Name sasl_auxprop_add_plugin – add a SASL auxiliary property plug-in

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/saslplug.h>

int sasl_auxprop_add_plugin(const char *plugname,
    sasl_auxprop_plug_init_t *cplugfunc);

Description Use the sasl_auxprop_add_plugin() interface to add a auxiliary property plug-in to the current list of auxiliary property plug-ins in the SASL library.

Parameters *plugname*   The name of the auxiliary property plug-in.

cplugfunc   The value of *cplugfunc* is filled in by the sasl_auxprop_plug_init_t structure.

Return Values sasl_auxprop_add_plugin() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The call to sasl_client_add_plugin() was successful.

SASL_BADVERS   Version mismatch with plug-in.

SASL_NOMEM   Memory shortage failure.

See sasl_errors(3SASL) for information on other SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), attributes(5)

Name sasl_auxprop_getctx – acquire an auxiliary property context

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

struct propctx *sasl_auxprop_getctx(sasl_conn_t **conn*);

Description The sasl_auxprop_getctx() interface returns an auxiliary property context for the given sasl_conn_t on which the sasl auxiliary property functions can operate. See sasl_auxprop(3SASL).

Parameters *conn*   The sasl_conn_t for which the request is being made

Return Values sasl_auxprop_getctx() returns a pointer to the context, upon success.
sasl_auxprop_getctx() returns NULL upon failure.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	Safe

See Also attributes(5)

Name sasl_auxprop_request – request auxialliary properties from SASL

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/sasl.h>

int sasl_auxprop_request(sasl_conn_t *conn, const char **propnames);

Description The sasl_auxprop_request() interface requests that the SASL library obtain properties form
any auxialiary property plugins that might be installed, for example, the user's home directory
from an LDAP server. The lookup occurs just after username canonicalization is complete.
Therefore, the request should be made before the call to sasl_server_start(3SASL), but
after the call to sasl_server_new(3SASL).

Parameters *conn*   The sasl_conn_t for which the request is being made

propnames A null-terminated array of property names to request. This array must persist
until a call to sasl_dispose(3SASL) on the sasl_conn_t.

Errors sasl_auxprop_request() returns SASL_OK upon success. See sasl_errors(3SASL) for a
discussion of other SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	Safe

See Also sasl_dispose(3SASL), sasl_errors(3SASL), sasl_server_new(3SASL),
sasl_server_start(3SASL), attributes(5)

Name sasl_canonuser_add_plugin – add a SASL user canonicalization plug-in

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/saslplug.h>

int sasl_canonuser_add_plugin(const char **plugname*,
    sasl_canonuser_plug_init_t **cplugfunc*);

Description Use the sasl_canonuser_add_plugin() interface to add a user canonicalization plug-in to the current list of user canonicalization plug-ins in the SASL library.

Parameters *plugname*   The name of the user canonicalization plug-in.

cplugfunc   The value of *cplugfunc* is filled in by the sasl_canonuser_plug_init_t structure.

Return Values sasl_server_add_plugin() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The call to sasl_client_add_plugin() was successful.

SASL_BADVERS   Version mismatch with plug-in.

SASL_NOMEM   Memory shortage failure.

See sasl_errors(3SASL) for information on other SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), attributes(5)

Name sasl_canon_user_t – the canon user callback

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/sasl.h>

int sasl_canon_user_t(sasl_conn_t *_conn_, void *_context_, const char *_user_,
    unsigned *ulen*, unsigned *flags*, const char *_user_realm_, char *_out_user_,
    unsigned *_out_umax_, unsigned *_out_ulen_);

Description The sasl_canon_user_t() interface is the callback function for an application-supplied user
canonical function. This function is subject to the requirements of all canonical functions. It
must copy the result into the output buffers, but the output buffers and the input buffers can
be the same.

Parameters *conn*   The SASL connection context.

context   The context from the callback record.

user   User name. The form of *user* is not canonical.

ulen   Length of *user*. The form of *ulen* is not canonical.

flags   One of the following values, or a bitwise OR of both:

SASL_CU_AUTHID   Indicates the authentication ID is canonical

SASL_CU_AUTHZID   Indicates the authorization ID is canonical

user_realm   Realm of authentication.

out_user   The output buffer for the user name.

out_max   The maximum length for the user name.

out_len   The actual length for the user name.

Return Values Like other SASL callback functions, sasl_canon_user_t() returns an integer that
corresponds to a SASL error code. See <sasl.h> for a complete list of SASL error codes.

Errors SASL_OK   The call to sasl_canon_user_t() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also   sasl_errors(3SASL), sasl_server_new(3SASL), attributes(5)

Name sasl_chalprompt_t – prompt for input in response to a challenge

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

int sasl_chalprompt_t(void **context*, int *id*, const char **challenge*,
    const char **prompt*, const char **defresult*, const char **result*,
    unsigned *len*);

Description Use the sasl_chalprompt_t() callback interface to prompt for input in response to a server
challenge.

Parameters *context*   The context from the callback record.

id   The callback id. *id* can have a value of SASL_CB_ECHOPROMPT or
              SASL_CB_NOECHOPROMPT

challenge   The server's challenge.

prompt   A prompt for the user.

defresult   The default result. The value of *defresult* can be NULL

result   The user's response. *result* is a null-terminated string.

len   The length of the user's response.

Return Values Like other SASL callback functions, sasl_chalprompt_t() returns an integer that
corresponds to a SASL error code. See <sasl.h> for a complete list of SASL error codes.

Errors SASL_OK   The call to sasl_chalprompt_t() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), sasl_server_new(3SASL), attributes(5)

Name sasl_checkapop – check an APOP challenge or response

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/sasl.h>

int sasl_checkapop(sasl_conn_t **conn*, const char **challenge*,
    unsigned *challen*, const char **response*, unsigned *resplen*);

Description The sasl_checkapop() interface checks an APOP challenge or response. APOP is an option POP3 authentication command that uses a shared secret password. See *RFC 1939*.

If sasl_checkapop() is called with a NULL challenge, sasl_checkapop() will check to see if the APOP mechanism is enabled.

Parameters *conn*   The sasl_conn_t for which the request is being made

challenge The challenge sent to the client

challen   The length of *challenge*

response   The client response

resplens   The length of *response*

Return Values sasl_checkapop() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   Indicates that the authentication is complete

All other error codes indicate an error situation that must be handled, or the authentication session should be quit. See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Obsolete
MT-Level	Safe

See Also sasl_errors(3SASL), attributes(5)

Meyers, J. and Rose, M. *RFC 1939, Post Office Protocol – Version 3*. Network Working Group. May 1996.

Name sasl_checkpass – check a plaintext password

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/sasl.h>

int sasl_checkpass(sasl_conn_t *conn, const char *user, unsigned *userlen*,
    const char *pass, unsigned *passlen*);

Description The sasl_checkpass() interface checks a plaintext password. The sasl_checkpass() interface is used for protocols that had a login method before SASL, for example, the LOGIN command in IMAP. The password is checked with the pwcheck_method.

The sasl_checkpass() interface is a server interface. You cannot use it to check passwords from a client.

The sasl_checkpass() interface checks the posible repositories until it succeeds or there are no more repositories. If sasl_server_userdb_checkpass_t is registered, sasl_checkpass() tries it first.

Use the pwcheck_method SASL option to specify which pwcheck methods to use.

The sasl_checkpass() interface supports the transition of passwords if the SASL option auto_transition is on.

If *user* is NULL, check is plaintext passwords are enabled.

Parameters *conn*   The sasl_conn_t for which the request is being made

pass   Plaintext password to check

passlen   The length of *pass*

user   User to query in current user_domain

userlen   The length of username.

Return Values sasl_checkpass() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   Indicates that the authentication is complete

All other error codes indicate an error situation that must be handled, or the authentication session should be quit. See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also sasl_errors(3SASL), attributes(5)

Name sasl_client_add_plugin – add a SASL client plug-in

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/saslplug.h>

int sasl_client_add_plugin(const char *plugname*,
    sasl_client_plug_init_t **cplugfunc*);

Description Use the sasl_client_add_plugin() interface to add a client plug-in to the current list of client plug-ins in the SASL library.

Parameters *plugname*   The name of the client plug-in.

cplugfunc   The value of *cplugfunc* is filled in by the sasl_client_plug_init_t structure.

Return Values sasl_client_add_plugin() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The call to sasl_client_add_plugin() was successful.

SASL_BADVERS   Version mismatch with plug-in.

SASL_NOMEM   Memory shortage failure.

See sasl_errors(3SASL) for information on other SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), attributes(5)

Name sasl_client_init – initialize SASL client authentication

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

int sasl_client_init(const sasl_callback_t **callbacks*);

Description Use the sasl_client_init() interface to initialize SASL. The sasl_client_init() interface must be called before any calls to sasl_client_start(3SASL). The call to sasl_client_init() initiallizes all SASL client drivers, for example, authentication mechanisms. SASL client drivers are usually found in the /usr/lib/sasl directory.

Parameters *callbacks*   Specifies the base callbacks for all client connections.

Return Values sasl_client_init() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The call to sasl_client_init() was successful.

SASL_BADVERS   There is a mismatch in the mechanism version.

SASL_BADPARAM   There is an error in the configuration file.

SASL_NOMEM   There is not enough memory to complete the operation.

All other error codes indicate an error situation that must be handled, or the authentication session should be quit. See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	Unsafe

See Also sasl_errors(3SASL), attributes(5)

Notes While most of libsasl is MT-Safe, no other libsasl function should be called until this function completes.

Name sasl_client_new – create a new client authentication object

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/sasl.h>

int sasl_client_new(const char **service*, const char **serverFQDN*,
    const char **iplocalport*, const char **ipremoteport*,
    const sasl_callback_t **prompt_supp*, unsigned *flags*,
    sasl_conn_t **pconn*);

Description Use the sasl_client_new() interface to create a new SASL context. This SASL context will be
used for all SASL calls for one connection. The context handles both authentication and the
integrity and encryption layers after authentication.

Parameters *service*   The registered name of the service that uses SASL, usually the protocol
name, for example, IMAP.

serverFQDN   The fully qualified domain name of the server, for example,
serverhost.cmu.edu.

iplocalport

The IP and port of the local side of the connection, or NULL. If *iplocalport* is
NULL, mechanisms that require IP address information are disabled. The
iplocalport string must be in one of the following formats:

- a.b.c.d:port (IPv6)
- [e:f:g:h:i:j:k:l]:port (IPv6)
- [e:f:g:h:i:j:a.b.c.d]:port (IPv6)
- a.b.c.d;port (IPv4)
- e:f:g:h:i:j:k:l;port (IPv6)
- e:f:g:h:i:j:a.b.c.d;port (IPv6)

ipremoteport   The IP and port of the remote side of the connection, or NULL.

prompt_supp   A list of the client interactions supported that are unique to this connection.
If this parameter is NULL, the global callbacks specified in
sasl_client_init(3SASL) are used.

flags   Usage flags. For clients, the flag SASL_NEED_PROXY is available.

pconn   The connection context allocated by the library. The *pconn* structure is used
for all future SASL calls for this connection.

Return Values sasl_client_new() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The call to sasl_client_new() was successful.

SASL_NOMECH   No mechanishm meets the requested properties.

SASL_BADPARAM   There is an error in the configuration file or passed parameters.

SASL_NOMEM   There is not enough memory to complete the operation.

All other error codes indicate an error situation that must be handled, or the authentication session should be quit. See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	Safe

See Also sasl_client_init(3SASL), sasl_errors(3SASL), attributes(5)

Name sasl_client_plug_init_t – client plug-in entry point

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/saslplug.h>

int sasl_client_plug_init_t(const sasl_utils_t **utils*, int *max_version*,
    int *out_version*, sasl_client_plug_t **pluglist*, int *plugcount*);

Description The sasl_client_plug_init_t() callback function is the client plug-in entry point.

Parameters *utils*   The utility callback functions.

max_version   The highest client plug-in version supported.

out_version   The client plug-in version of the result..

pluglist   The list of client mechanism plug-ins.

plugcount   The number of client mechanism plug-ins.

Return Values Like other SASL callback functions, sasl_client_plug_init_t() returns an integer that corresponds to a SASL error code. See <sasl.h> for a complete list of SASL error codes.

Errors SASL_OK   The call to sasl_client_plug_init_t() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), attributes(5)

Name sasl_client_start – perform a step in the authentication negotiation

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

int sasl_client_start(sasl_conn_t **conn*, const char **mechlist*,
    sasl_interact_t **prompt_need*, const char **clientout*,
    unsigned **clientoutlen*, const char **mech*);

Description Use the sasl_client_start() interface to select a mechanism for authentication and start
the authentication session. The *mechlist* parameter holds the list of mechanisms that the client
might like to use. The mechanisms in the list are not necessarily supported by the client, nor
are the mechanisms necessarily valid. SASL determines which of the mechanisms to use based
upon the security preferences specified earlier. The list of mechanisms is typically a list of
mechanisms that the server supports, acquired from a capability request.

If SASL_INTERACT is returned, the library needs some values to be filled in before it can
proceed. The *prompt_need* structure is filled in with requests. The application fullfills these
requests and calls sasl_client_start() again with identical parameters. The *prompt_need*
parameter is the same pointer as before, but it is filled in by the application.

Parameters *conn*   The SASL connection context.

mechlist   A list of mechanism that the server has available. Punctuation is ignored.

prompt_need   A list of prompts that are needed to continue, if necessary.

clientout
clientoutlen *clientout* and *clientoutlen* are created. They contain the initial client response
                to send to the server. It is the job of the client to send them over the network to
                the server. Any protocol specific encodingthat is necessary, for example
                base64 encoding, must be done by the client.

                If the protocol lacks client-send-first capability, then set *clientout* to NULL. If
                there is no initial client-send, then *clientout* will be set to NULL on return.

mech   Contains the name of the chosen SASL mechanism, upon success.

Return Values sasl_client_start() returns an integer that corresponds to a SASL error code.

Errors SASL_CONTINUE   The call to sasl_client_start() was successful, and more steps are
                          needed in the authentication.

All other error codes indicate an error situation that must be handled, or the authentication
session should be quit. See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	Safe

See Also   sasl_errors(3SASL), attributes(5)

Name sasl_client_step – acquire an auxiliary property context

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/sasl.h>

int sasl_client_step(sasl_conn_t **conn*, const char *serverin*,
    sasl_interact_t **unsigned *serverinlen*, *prompt_need*,
    const char **clientout*, sasl_interact_t **unsigned *clientoutlen*);

Description Use the sasl_client_step() interface performs a step in the authentication negotiation.
sasl_client_step() returns SASL_OK if the complete negotiation is successful. If the
negotiation on step is completed successfuly, but at least one more step is required,
sasl_client_step() returns SASL_CONTINUE. A client should not assume an authentication
negotiaion is successful because the server signaled success through the protocol. For
example, if the server signaled OK Authentication succeeded in IMAP,
sasl_client_step() should be called one more time with a *serverinlen* of zero.

If a call to sasl_client_step() returns SASL_INTERACT, the library requires some values
before sasl_client_step() can proceed. The *prompt_need* structure will be filled with the
requests. The application should fulfull these requests and call sasl_client_step() again
with identical parameters. The *prompt_need* parameter will be the same pointer as before, but
it will have been filled in by the application.

Parameters *conn*   The SASL connection context.

serverin   The data given by the server. The data is decoded if the protocol encodes
requests sent over the wire.

serverinlen   The length of the *serverin*.

clientout
clientoutlen *clientout* and *clientoutlen* are created. They contain the initial client response
to send to the server. It is the job of the client to send them over the network to
the server. Any protocol specific encodingthat is necessary, for example
base64 encoding, must be done by the client.

prompt_need   A list of prompts that are needed to continue, if necessary.

Return Values sasl_client_step() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The call to sasl_client_start() was successful. Authentication is
complete.

SASL_CONTINUE   The call to sasl_client_start() was successful, but at least one more
step is required for authentication.

SASL_INTERACT   The library requires some values before sasl_client_step() can
proceed.

All other error codes indicate an error situation that must be handled, or the authentication session should be quit. See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	Safe

See Also sasl_errors(3SASL), attributes(5)

Name sasl_decode – decode data received

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

int sasl_decode(sasl_conn_t *conn, const char *input, unsigned *inputlen*,
     const char **output, unsigned *outputlen*);

Description Use the sasl_decode() interface to decode data received. After authentication, call this function on all data received. The data is decoded from encrypted or signed form to plain data. If no security lay is negotiated, the output is identical to the input.

Do not give sasl_decode() more data than the negotiated maxbufsize. See sasl_getprop(3SASL).

sasl_decode() can complete successfully although the value of *outputlen* is zero. If this is the case, wait for more data and call sasl_decode() again.

Parameters *conn*   The SASL connection context.

input   Data received.

inputlen   The length of *input*

output   The decoded data. *output* must be allocated or freed by the library.

outputlen   The length of *output*.

Return Values sasl_decode() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The call to sasl_decode() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	Safe

See Also sasl_errors(3SASL), sasl_getprop(3SASL), attributes(5)

Name sasl_decode64 – decode base64 string

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/saslutil.h>

int sasl_decode64(const char *in*, unsigned *inlen*, char *out*,
    unsigned *outmax*,unsigned *outlen*);

Description Use the sasl_decode64() interface to decode a base64 encoded buffer.

Parameters *in*   Input data.

inlen   The length of the input data.

out   The output data. The value of *out* can be the same as *in*. However, there must be
            enough space.

outlen   The length of the actual output.

outmax   The maximum size of the output buffer.

Return Values sasl_decode64() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The call to sasl_decode64() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), attributes(5)

Name   sasl_dispose – dispose of a SASL connection object

Synopsis   cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

void sasl_dispose(sasl_conn_t **pconn*);

Description   Use the sasl_dispose() interface when a SASL connection object is no longer needed.
Generally, the SASL connection object is no longer needed when the protocol session is
completed, not when authentication is completed, as a security layer may have been
negotiated.

Parameters   *pconn*   The SASL connection context

Return Values   sasl_dispose() has no return values.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	Safe

See Also   attributes(5)

Name sasl_done – dispose of all SASL plug-ins

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

void sasl_encode(void)

Description Make a call to the sasl_done() interface when the application is completely done with the
SASL library. You must call sasl_dispose(3SASL) before you make a call to sasl_done().

Return Values sasl_done() has no return values.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	Safe

See Also sasl_dispose(3SASL), attributes(5)

Name sasl_encode, sasl_encodev – encode data for transport to an authenticated host

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
```
#include <sasl/sasl.h>
```

```
int sasl_encode(sasl_conn_t *conn, const char *input, unsigned inputlen,
     const char **output, unsigned *outputlen);
```

```
int sasl_encodev(sasl_conn_t *conn, const struct iovec *invec,
     unsigned numiov, const char *outputlen);
```

Description The sasl_encode() interface encodes data to be sent to a remote host for which there has been a successful authentication session. If there is a negotiated security, the data is signed or encrypted, and the output is sent without modification to the remote host. If there is no security layer, the output is identical to the input.

The sasl_encodev() interface functions the same as the sasl_encode() interface, but operates on a struct iovec instead of a character buffer.

Parameters *conn*   The SASL connection context.

input   Data.

inputlen   *input* length.

output   The encoded data. *output* must be allocated or freed by the library.

outputlen The length of *output*.

invec   A pointer to set of iovec structures.

numiov   The number of iovec structures in the *invec* set.

Return Values sasl_encode() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The call to sasl_encode() or sasl_encodev()was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	Safe

See Also attributes(5)

Name sasl_encode64 – encode base64 string

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/saslutil.h>

int sasl_encode64(const char **in*, unsigned *inlen*, char **out*,
    unsigned *outmax*, unsigned **outlen*);

Description Use the sasl_encode64() interface to convert an octet string into a base64 string. This
routine is useful for SASL profiles that use base64, such as the IMAP (IMAP4) and POP
(POP_AUTH) profiles. The output is null-terminated. If *outlen* is non-NULL, the length is
placed in the *outlen*.

Parameters *in*   Input data.

inlen   The length of the input data.

out   The output data. The value of *out* can be the same as *in*. However, there must be
          enough space.

outlen   The length of the actual output.

outmax   The maximum size of the output buffer.

Return Values sasl_encode64() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The call to sasl_encode64() was successful.

SASL_BUFOVER   The output buffer was too small.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), attributes(5)

Name sasl_erasebuffer – erase buffer

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/saslutil.h>

void sasl_erasebuffer(char **pass*, unsigned *len*);

Description Use the sasl_erasebuffer() interface to erase a security sensitive buffer or password. The implementation may use recovery-resistant erase logic.

Parameters *pass*   A password

len   The length of the password

Return Values The sasl_erasebuffer() interface returns no return values.

Errors None.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also attributes(5)

Name sasl_errdetail – retrieve detailed information about an error

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/sasl.h>

const char * sasl_errdetail(sasl_conn_t *conn);

Description The sasl_errdetail() interface returns an internationalized string that is a message that describes the error that occurred on a SASL connection. The sasl_errdetail() interface provides a more user friendly error message than the SASL error code returned when SASL indicates that an error has occurred on a connection. See sasl_errors(3SASL).

Parameters *conn*   The SASL connection context for which the inquiry is made.

Return Values sasl_errdetail() returns the string that describes the error that occurred, or NULL, if there was an error retrieving it.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	Safe

See Also sasl_errors(3SASL), sasl_seterror(3SASL), attributes(5)

Name	sasl_errors – SASL error codes	

Synopsis `#include <sasl/sasl.h>`

Description This man page describes the general error codes that can be returned by calls into the SASL library. The meaning of the error code can vary slightly based upon the context of the call from which it is returned.

Errors

Common Result Codes	SASL_OK	The call was successful.
	SASL_CONTINUE	Another step is required for authentication.
	SASL_FAILURE	Generic failure.
	SASL_NOMEM	Memory shortage failure.
	SASL_BUFOVER	Overflowed buffer.
	SASL_NOMECH	The mechanism was not supported, or no mechanisms matched the requirements.
	SASL_BADPROT	The protocol was bad, invalid or cancelled.
	SASL_NOT DONE	Cannot request information. Not applicable until later in the exchange.
	SASL_BADPARAM	An invalid parameter was supplied.
	SASL_TRYAGAIN	Transient failure, for example, a weak key.
	SASL_BADMAC	Integrity check failed.
	SASL_NOTINIT	SASL library not initialized.
Client Only Result Codes	SASL_INTERACT	Needs user interaction.
	SASL_BADSERV	Server failed mutual authentication step.
	SASL_WRONGMECH	Mechanism does not support the requested feature.
Server Only Result Codes	SASL_BADAUTH	Authentication failure.
	SASL_NOAUTHZ	Authorization failure.
	SASL_TOOWEAK	The mechanism is too weak for this user.
	SASL_ENCRYPT	Encryption is needed to use this mechanism.
	SASL_TRANS	One time use of a plaintext password will enable requested mechanism for user.
	SASL_EXPIRED	The passphrase expired and must be reset.
	SASL_DISABLED	Account disabled.

	SASL_NOUSER	User not found.
	SASL_BADVERS	Version mismatch with plug-in.
	SASL_NOVERIFY	The user exists, but there is no verifier for the user.
Password Setting Result Codes	SASL_PWLOCK	Passphrase locked.
	SASL_NOCHANGE	The requested change was not needed.
	SASL_WEAKPASS	The passphrase is too weak for security policy.
	SASL_NOUSERPASS	User supplied passwords are not permitted.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	Safe

See Also attributes(5)

Name sasl_errstring – translate a SASL return code to a human-readable form

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/sasl.h>

const char * sasl_errstring(int *saslerr*, const char *langlist*,
    const char **outlang*);

Description The sasl_errstring() interface is called to convert a SASL return code from an integer into a human readable string.

You should not used the sasl_errstring() interface to extract error code information from SASL. Applications should use sasl_errdetail(3SASL) instead, which contains this error information and more.

The sasl_errstring() interface supports only i-default and i-local at this time.

Parameters *saslerr* The error number to be translated.

langlist A comma-separated list of languages. See *RFC 1766*. If the*langlist* parameter has aNULL value, the default language,i-default, is used.

outlang The language actually used. The *outlang* parameter can be NULL. The returned error string is in UTF-8.

Return Values sasl_errstring() returns the string that describes the error that occurred, or NULL, if there was an error retrieving it.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	Safe

See Also sasl_errors(3SASL), sasl_seterror(3SASL), attributes(5)

Alvestrand, H. *RFC 1766, Tags for the Identification of Languages*. Network Working Group. November 1995.

Name sasl_getcallback_t – callback function to lookup a sasl_callback_t for a connection

Synopsis cc [*flag ...*] *file ...* -lsasl   [*library ...*]
#include <sasl/saslplug.h>

int sasl_getcallback_t(sasl_conn_t *conn*, unsigned long *callbacknum*,
    int *(**proc)*( ), void **pcontext*);

Description The sasl_getcallback_t() function is a callback to lookup a sasl_callback_t for a connection.

Parameters *conn*   The connection to lookup a callback for.

callbacknum   The number of the callback.

proc   Pointer to the callback function. The value of *proc* is set to NULL upon failure.

pcontext   Pointer to the callback context. The value of *pcontext* is set to NULL upon failure.

Return Values Like other SASL callback functions, sasl_getcallback_t() returns an integer that corresponds to a SASL error code. See <sasl.h> for a complete list of SASL error codes.

Errors SASL_OK   The call to sasl_getcallback_t() was successful.

SASL_FAIL   Unable to find a callback of the requested type.

SASL_INTERACT   The caller must use interaction to get data.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), attributes(5)

Name sasl_getopt_t – the SASL get option callback function

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

int sasl_getopt_t(void *context, const char *plugin_name,
    const char *option, const char **result, unsigned *len);

Description The sasl_getopt_t() function allows a SASL configuration to be encapsulated in the caller's configuration system. Some implementations may use default configuration file(s) if this function is omitted. Configuration items are arbitrary strings and are plug-in specific.

Parameters
context   The option context from the callback record.

plugin_name   The name of the plug-in. If the value of *plugin_name* is NULL, the the plug-in is a general SASL option.

option   The name of the option.

result   The value of *result* is set and persists until the next call to sasl_getopt_t() in the same thread. The value of *result* is unchanged if *option* is not found.

len   The length of *result*. The value of *result* can be NULL.

Return Values Like other SASL callback functions, sasl_getopt_t() returns an integer that corresponds to a SASL error code. See <sasl.h> for a complete list of SASL error codes.

Errors SASL_OK   The call to sasl_getopt_t() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), attributes(5)

Name   sasl_getpath_t – the SASL callback function to indicate location of the security mechanism drivers

Synopsis   cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

int sasl_getpath_t(void **context*, char **path*);

Description   Use the sasl_getpath_t() function to enable the application to use a different location for the SASL security mechanism drivers, which are shared library files. If the sasl_getpath_t() callback is not used, SASL uses /usr/lib/sasl by default.

Parameters   *context*   The getpath context from the callback record

path   The path(s) for the location of the SASL security mechanism drivers. The values for *path* are colon-separated.

Return Values   Like other SASL callback functions, sasl_getpath_t() returns an integer that corresponds to a SASL error code. See <sasl.h> for a complete list of SASL error codes.

Errors   SASL_OK   The call to sasl_getpath_t() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also   sasl_errors(3SASL), attributes(5)

Name sasl_getprop – get a SASL property

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

int sasl_getprop(sasl_conn_t **conn*, int *propnum*, const void **pvalue*);

Description Use the sasl_getprop() interface to get the value of a SASL property. For example, after
successful authentication, a server may want to know the authorization name. Similarly, a
client application may want to know the strength of the security level that was negotiated.

Parameters *conn*   The SASL connection context.

propnum   The identifier for the property requested.

pvalue   The value of the SASL property. This value is filled in upon a successful call.
              Possible SASL values include:

SASL_USERNAME	A pointer to a null-terminated user name.
SASL_SSF	The security layer security strength factor. If the value of SASL_SSF is 0, a call to sasl_encode() or sasl_decode() is unnecessary.
SASL_MAXOUTBUF	The maximum size of output buffer returned by the selected security mechanism
SASL_DEFUSERREALM	Server authentication realm used.
SASL_GETOPTCTX	The context for getopt() callback.
SASL_IPLOCALPORT	Local address string.
SASL_IPREMOTEPORT	Remote address string.
SASL_SERVICE	Service passed on to sasl_*_new().
SASL_SERVERFQDN	Server FQDN passed on to sasl_*_new().
SASL_AUTHSOURCE	Name of authentication source last used. Useful for failed authentication tracking.
SASL_MECHNAME	Active mechanism name, if any.
SASL_PLUGERR	Similar to sasl_errdetail().

Errors SASL_OK   The call to sasl_getprop() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also   sasl_errors(3SASL), attributes(5)

Name sasl_getrealm_t – the realm acquisition callback function

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

int sasl_getrealm_t(void **context*, int *id*, const char ***availrealms*,
     const char ***result*);

Description Use the sasl_getrealm_t() functionwhen there is an interaction with SASL_CB_GETREALM as the type.

If a mechanism would use this callback, but it is not present, then the first realm listed is automatically selected. A mechanism can still force the existence of a getrealm callback by SASL_CB_GETREALM to its required_prompts list.

Parameters *context*   The context from the callback record

id   The callback ID (SASL_CB_GETREALM)

availrealms   A string list of the vailable realms. *availrealms* is a null-terminated sting that can be empty.

result   The chosen realm. *result* is a null-terminated string.

Return Values Like other SASL callback functions, sasl_getrealm_t() returns an integer that corresponds to a SASL error code. See <sasl.h> for a complete list of SASL error codes.

Errors SASL_OK   The call to sasl_getrealm_t() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), attributes(5)

Name sasl_getsecret_t – the SASL callback function for secrets (passwords)

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

int sasl_getsecret_t(sasl_conn_t **conn*, void **context*,
    int *id*, sasl_secret_t **psecret*);

Description Use the sasl_getsecret_t() function to retrieve the secret from the application. Allocate a
sasl_secret_t to length sizeof(sasl_secret_t)+<length of secret>. sasl_secret_t
has two fields of *len* which contain the length of *secret* in bytes and the data contained in *secret*.
The *secret* string does not need to be null-terminated.

Parameters *conn*   The connection context

context   The context from the callback structure

id   The callback ID

psecret   To cancel, set the value of *psecret* to NULL. Otherwise, set the value to the password
            structure. The structure must persist until the next call to sasl_getsecret_t() in
            the same connection. Middleware erases password data when it is done with it.

Return Values Like other SASL callback functions, sasl_getsecret_t() returns an integer that corresponds
to a SASL error code. See <sasl.h> for a complete list of SASL error codes.

Errors SASL_OK   The call to sasl_getsecret_t() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), attributes(5)

Name sasl_getsimple_t – the SASL callback function for username, authname and realm

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

int sasl_getsimple_t(void **context*, int *id*, const char **result*,
    unsigned **len*);

Description Use the sasl_getsimple_t() callback function to retrieve simple data from the application such as the authentication name, the authorization name, and the realm. The *id* parameter indicates which value is requested.

Parameters *context*   The context from the callback structure.

id   The callback ID. Possible values for *id* include:

SASL_CB_USER   Client user identity for login.

SASL_CB_AUTHNAME   Client authentication name.

SASL_CB_LANGUAGE   Comma-separated list of languages pursuant to *RFC 1766*.

SASL_CB_CNONCE   The client-nonce. This value is used primarily for testing.

result   To cancel user, set the value of *result* with a null-terminated string. If the value of *result* is NULL, then the user is cancelled.

len   The length of *result*.

Return Values Like other SASL callback functions, sasl_getsimple_t() returns an integer that corresponds to a SASL error code. See <sasl.h> for a complete list of SASL error codes.

Errors SASL_OK   The call to sasl_getsimple_t() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), attributes(5)

Alvestrand, H. *RFC 1766, Tags for the Identification of Languages*. Network Working Group. November 1995.

Name sasl_global_listmech – retrieve a list of the supported SASL mechanisms

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

const char ** sasl_global_listmech( );

Description The sasl_global_listmech() interface to returns a null-terminated array of strings that lists all of the mechanisms that are loaded by either the client or server side of the library.

Return Values A successful call to sasl_global_listmech() returns a pointer the array. On failure, NULL is returned. The SASL library is uninitialized.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Obsolete
MT-Level	MT–Safe

See Also attributes(5)

Name sasl_idle – perform precalculations during an idle period

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

int sasl_idle(sasl_conn_t *conn);

Description Use the sasl_idle() interface during an idle period to allow the SASL library or any
mechanisms to perform any necessary precalculation.

Parameters *conn*   The SASL connection context. The value of *conn* can be NULL in order to complete a
precalculation before the connection takes place.

Return Values sasl_idle() returns the following values:

1   Indicates action was taken

0   Indicates no action was taken

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also attributes(5)

Name sasl_listmech – retrieve a list of the supported SASL mechanisms

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

int sasl_listmech(sasl_conn_t *conn*, const char *user*, const char *prefix*,
    const char *sep*, const char *suffix*, const char **result*,
    unsigned *plen*,int *pcount*);

Description The sasl_listmech() interface returns a string listing the SASL names of all the mechanisms available to the specified user. This call is typically given to the client through a capability command or initial server response. Client applications need this list so that they know what mechanisms the server supports.

Parameters *conn*   The SASL context for this connection user restricts the mechanism list to those mechanisms available to the user. This parameter is optional.

user   Restricts security mechanisms to those available to that user. The value of *user* may be NULL, and it is not used if called by the client application.

prefix   Appended to the beginning of *result*.

sep   Appended between mechanisms.

suffix   Appended to the end of *result*.

result   A null-terminated result string. *result* must be allocated or freed by the library.

plen   The length of the result filled in by the library. The value of *plen* may be NULL.

pcount   The number of mechanisms available. The value of *pcount* is filled in by the library. The value of *pcount* may be NULL

Return Values sasl_listmech() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The call to sasl_listmech() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), attributes(5)

Name sasl_log_t – the SASL logging callback function

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/sasl.h>

int sasl_log_t(void **context*, int *level*, const char **message*);

Description Use the sasl_log_t() function to log warning and error messages from the SASL library.
syslog(3C) is used, unless another logging function is specified.

Parameters *context*   The logging context from the callback record.

level   The logging level. Possible values for *level* include:

SASL_LOG_NONE	Do not log anything.
SASL_LOG_ERR	Log unusual errors. This is the default log level.
SASL_LOG_FAIL	Log all authentication failures.
SASL_LOG_WARN	Log non-fatal warnings.
SASL_LOG_NOTE	Log non-fatal warnings (more verbose than SASL_LOG_WARN).
SASL_LOG_DEBUG	Log non-fatal warnings (more verbose than SASL_LOG_NOTE).
SASL_LOG_TRACE	Log traces of internal protocols.
SASL_LOG_PASS	Log traces of internal protocols, including passwords.

message   The message to log

Return Values Like other SASL callback functions, sasl_log_t() returns an integer that corresponds to a
SASL error code. See <sasl.h> for a complete list of SASL error codes.

Errors SASL_OK   The call to sasl_log_t() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also   sasl_errors(3SASL), syslog(3C), attributes(5)

Name sasl_server_add_plugin – add a SASL server plug-in

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/saslplug.h>

int sasl_server_add_plugin(const char *_plugname_,
    sasl_server_plug_init_t *_cplugfunc_);

Description Use the sasl_server_add_plugin() interface to add a server plug-in to the current list of client plug-ins in the SASL library.

Parameters *plugname*   The name of the server plug-in.

cplugfunc   The value of *cplugfunc* is filled in by the sasl_server_plug_init_t structure.

Return Values sasl_server_add_plugin() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The call to sasl_client_add_plugin() was successful.

SASL_BADVERS   Version mismatch with plug-in.

SASL_NOMEM   Memory shortage failure.

See sasl_errors(3SASL) for information on other SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), attributes(5)

Name sasl_server_init – SASL server authentication initialization

Synopsis
```
cc [ flag ... ] file ... -lsasl   [ library ... ]
#include <sasl/sasl.h>

int sasl_server_init(const sasl_callback *callbacks, const char *appname);
```

Description Use the sasl_server_init() interface to initialize SASL. You must call sasl_server_init() before you make a call to sasl_server_start(). sasl_server_init() may be called only once per process. A call to sasl_server_init() initializes all SASL mechanism drivers, that is, the authentication mechanisms. The SASL mechanism drivers are usually found in the /usr/lib/sasl directory.

Parameters
callbacks   Specifies the base callbacks for all client connections.

appname   The name of the application for lower level logging. For example, the sendmail server calls *appname* this way:

```
sasl_server_init(srvcallbacks, "Sendmail")
```

Return Values sasl_server_init() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The call to sasl_server_init() was successful.

All other error codes indicate an error situation that must be handled, or the authentication session should be quit. See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	Unsafe

See Also sasl_errors(3SASL), attributes(5)

Notes While most of libsasl is MT-Safe, no other libsasl function should be called until this function completes.

Name sasl_server_new – create a new server authentication object

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/sasl.h>

int sasl_server_new(const char *service, const char *serverFQDN,
    const char *user_realm, const char *iplocalport,
    const char *ipremoteport, const sasl_callback_t *callbacks,
    unsigned *flags*, sasl_conn_t **pconn);

Description Use the sasl_server_new() interface to create a new SASL context. This context will be used
for all SASL calls for one connection. The new SASL context handles both authentication and
integrity or encryption layers after authentication.

Parameters *service*   The registered name of the service that uses SASL. The registered name is
                   usually the protocol name, for example, IMAP.

serverFQDN   The fully-qualified server domain name. If the value of *serverFQDN* is NULL,
                   use gethostname(3C). The *serverFQDN* parameter is useful for
                   multi-homed servers.

user_realm   The domain of the user agent. The *user_realm* is usually not necessary. The
                   default value of *user_realm* is NULL.

iplocalport

                   The IP address and port of the local side of the connection. The value of
                   iplocalport may be NULL. If *iplocalport* is NULL, mechanisms that require IP
                   address information are disabled. The *iplocalport* string must be in one of
                   the following formats:

                   ■ a.b.c.d:port (IPv4)
                   ■ [e:f:g:h:i:j:k:l]:port (IPv6)
                   ■ [e:f:g:h:i:j:a.b.c.d]:port (IPv6)

                   The following older formats are also supported:

                   ■ a.b.c.d;port (IPv4)
                   ■ e:f:g:h:i:j:k:l;port (IPv6)
                   ■ e:f:g:h:i:j:a.b.c.d;port (IPv6)

ipremoteport   The IP address and port of the remote side of the connection. The value of
                   ipremoteport may be NULL. See *iplocalport*.

callbacks   Callbacks, for example: authorization, lang, and new getopt context.

flags   Usage flags. For servers, the flags SASL_NEED_PROXY and
                   SASL_SUCCESS_DATA are available.

pconn   A pointer to the connection context allocated by the library. This structure
                   will be used for all future SASL calls for this connection.

Return Values sasl_server_new() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The call to sasl_server_new() was successful.

All other error codes indicate an error situation that must be handled, or the authentication session should be quit. See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also gethostname(3C), sasl_errors(3SASL), attributes(5)

Name sasl_server_plug_init_t – server plug-in entry point

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/saslplug.h>

int sasl_server_plug_init_t(const sasl_utils_t *utils, int *max_version*,
    int *out_version*, sasl_client_plug_t **pluglist*, int *plugcount*);

Description The sasl_server_plug_init_t() callback function is the server plug-in entry point.

Parameters *utils*   The utility callback functions.

max_version   The highest server plug-in version supported.

out_version   The server plug-in version of the result.

pluglist   The list of server mechanism plug-ins.

plugcount   The number of server mechanism plug-ins.

Return Values Like other SASL callback functions, sasl_server_plug_init_t() returns an integer that corresponds to a SASL error code. See <sasl.h> for a complete list of SASL error codes.

Errors SASL_OK   The call to sasl_server_plug_init_t() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), attributes(5)

Name sasl_server_start – create a new server authentication object

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

int sasl_server_start(sasl_conn_t *conn, const char *mech,
    const char *clientin, unsigned *clientinlen,
    const char **serverout, unsigned *serveroutlen);

Description The sasl_server_start() interface begins the authentication with the mechanism specified by the *mech* parameter. sasl_server_start() fails if the mechanism is not supported.

Parameters
conn	The SASL context for this connection.
mech	The mechanism name that the client requested.
clientin	The initial response from the client. The value of *clientin* is NULL if the protocol lacks support for the client-send-first or if the other end did not have an initial send. No initial client send is distinct from an initial send of a null string. The protocol must account for this difference.
clientinlen	The length of the initial response.
serverout	Created by the plugin library. The value of *serverout* is the initial server response to send to the client. *serverout* is allocated or freed by the library. It is the job of the client to send it over the network to the server. Protocol specific encoding, for example base64 encoding, must be done by the server.
serveroutlen	The length of the initial server challenge.

Return Values sasl_server_start() returns an integer that corresponds to a SASL error code.

Errors
SASL_OK	Authentication completed successfully.
SASL_CONTINUE	The call to sasl_server_start() was successful, and more steps are needed in the authentication.

All other error codes indicate an error situation that must be handled, or the authentication session should be quit. See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT−Safe

See Also   gethostname(3C), sasl_errors(3SASL), attributes(5)

Name sasl_server_step – perform a step in the server authentication negotiation

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

int sasl_server_step(sasl_conn_t *conn, const char *clientin,
    unsigned *clientinlen*, const char **serverout*,
    unsigned *serveroutlen*);

Description The sasl_server_step() performs a step in the authentication negotiation.

Parameters *conn*   The SASL context for this connection.

clientin   The data given by the client. The data is decoded if the protocol encodes
               requests that are sent over the wire.

clientinlen   The length of *clientin*.

serverout
serveroutlen Set by the library and sent to the client.

Return Values sasl_server_step() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The whole authentication completed successfully.

SASL_CONTINUE   The call to sasl_server_step() was successful, and at least one more
                  step is needed for the authentication.

All other error codes indicate an error situation that you must handle, or you should quit the
authentication session. See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), attributes(5)

Name sasl_server_userdb_checkpass_t – plaintext password verification callback function

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/sasl.h>

int sasl_sasl_server_userdb_checkpass_t(sasl_conn_t *conn*, void *context*,
    const char *user*, const char *pass*, unsigned *passlen*, struct propctx *propctx*);

Description Use the sasl_sasl_server_userdb_checkpass_t() callback function to verify a plaintext
password against the callback supplier's user database. Verification allows additional ways to
encode the userPassword property.

Parameters *conn*   The SASL connection context.

context   The context from the callback record.

user   A null-terminated user name with user@realm syntax.

pass   The password to check. This string cannot be null-terminated.

passlen   The length of *pass*.

propctx   The property context to fill in with userPassword.

Return Values Like other SASL callback functions, sasl_server_userdb_checkpass_t() returns an integer
that corresponds to a SASL error code. See <sasl.h> for a complete list of SASL error codes.

Errors SASL_OK   The call to sasl_server_userdb_checkpass_t() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), attributes(5)

Name sasl_server_userdb_setpass_t – user database plaintext password setting callback function

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

int sasl_server_userdb_setpass_t(sasl_conn_t *conn, void *context,
    const char *user, const char *pass, unsigned *passlen*, struct propctx *propctx,
    unsigned *flags*);

Description Use the sasl_server_userdb_setpass_t() callback function to store or change a a plaintext password in the callback supplier's user database.

Parameters *conn* The SASL connection context.

context The context from the callback record.

user A null-terminated user name with user@realm syntax.

pass The password to check. This string cannot be null-terminated.

passlen The length of *pass*.

propctx Auxiliary properties. The value of *propctx* is not stored.

flags See sasl_setpass(3SASL). sasl_server_userdb_setpass_t() uses the same *flags* that are passed to sasl_setpass().

Return Values Like other SASL callback functions, sasl_server_userdb_setpass_t() returns an integer that corresponds to a SASL error code. See <sasl.h> for a complete list of SASL error codes.

Errors SASL_OK The call to sasl_server_userdb_setpass_t() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), sasl_setpass(3SASL), attributes(5)

Name sasl_set_alloc – set the memory allocation functions used by the SASL library

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/sasl.h>

void sasl_set_alloc(sasl_malloc_t *m, sasl_calloc_t *c, sasl_realloc_t *r,
    sasl_free_t *f);

Description Use the sasl_set_alloc() interface to set the memory allocation routines that the SASL library and plug-ins will use.

Parameters *c* A pointer to a calloc() function

f A pointer to a free() function

m A pointer to a malloc() function

r A pointer to a realloc() function

Return Values sasl_set_alloc() has no return values.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Obsolete
MT-Level	Unsafe

See Also attributes(5)

Notes While most of libsasl is MT-Safe, sasl_set_* modifies the global state and should be considered Unsafe.

Name sasl_seterror – set the error string

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

void sasl_seterror(sasl_conn_t *conn*, unsigned *flags*,
    const char *fmt*, ...);

Description The sasl_seterror() interface sets the error string that will be returned by
sasl_errdetail(3SASL). Use syslog(3C) style formatting, that is, use printf()—style with
%m as the most recent errno error.

The sasl_seterror() interface is primarily used by server callback functions and internal
plug-ins, for example, with the sasl_authorize_t callback. The sasl_seterror() interface
triggers a call to the SASL logging callback, if any, with a level of SASL_LOG_FAIL, unless the
SASL_NOLOG flag is set.

Make the message string sensitive to the current language setting. If there is no
SASL_CB_LANGUAGE callback, message strings must be i-default. Otherwise, UTF-8 is used.
Use of *RFC 2482* for mixed-language text is encouraged.

If the value of *conn* is NULL, the sasl_seterror() interface fails.

Parameters *conn*   The sasl_conn_t for which the call to sasl_seterror() applies.

flags   If set to SASL_NOLOG, the call to sasl_seterror() is not logged.

fmt   A syslog(3C) style format string.

Return Values sasl_seterror() has no return values.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errdetail(3SASL), syslog(3C), attributes(5)

Whistler, K. and Adams, G. *RFC 2482, Language Tagging in Unicode Plain Text.* Network
Working Group. January 1999.

Name sasl_set_mutex – set the mutex lock functions used by the SASL library

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/sasl.h>

void sasl_set_mutex(sasl_mutex_alloc_t *a, sasl_mutex_lock_t *l,
    sasl_mutex_unlock_t *u, sasl_mutex_free_t *f);

Description Use the sasl_set_mutex() interface to set the mutex lock routines that the SASL library and plug-ins will use.

Parameters *a* A pointer to the mutex lock allocation function

f A pointer to the mutex free or destroy function

l A pointer to the mutex lock function

u A pointer to the mutex unlock function

Return Values sasl_set_mutex() has no return values.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Obsolete
MT-Level	Unsafe

See Also attributes(5)

Notes While most of libsasl is MT-Safe, sasl_set_* modifies the global state and should be considered Unsafe.

Name sasl_setpass – set the password for a user

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

int sasl_setpass(sasl_conn_t *conn, const char *user, const char *pass,
    unsigned *passlen*, const char *oldpass, unsigned *oldpasslen*,
    unsigned *flags*);

Description Use the sasl_setpass() interface to set passwords. sasl_setpass() uses the
SASL_CB_SERVER_USERDB_SETPASS callback, if one is supplied. Additionally, if any server
mechanism plugins supply a setpass callback, the setpass callback would be called. None of the
server mechanism plugins currently supply a setpass callback.

Parameters *conn*   The SASL connection context

user   The username for which the password is set

pass   The password to set

passlen   The length of *pass*

oldpass   The old password, which is optional

oldpasslen   The length of *oldpass*, which is optional

flags   Refers to flags, including, SASL_SET_CREATE and SASL_SET_DISABLE. Use these
flags to create and disable accounts.

Return Values sasl_setpass() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The call to sasl_setpass() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), sasl_getprop(3SASL), attributes(5)

Name sasl_setprop – set a SASL property

Synopsis cc [*flag* ...] *file* ... -lsasl [*library* ...]
#include <sasl/sasl.h>

int sasl_setprop(sasl_conn_t *conn*, int *propnum*, const void *pvalue*);

Description Use the sasl_setprop() interface to set the value of a SASL property. For example, an
application can use sasl_setprop() to tell the SASL liabrary about any external negotiated
security layer like TLS.

sasl_setprop() uses the following flags.

SASL_AUTH_EXTERNAL   External authentication ID that is a pointer of type const char

SASL_SSF_EXTERNAL   External SSF active of type sasl_ssf_t

SASL_DEFUSERREALM   User realm that is a pointer of type const char

SASL_SEC_PROPS   sasl_security_properties_t, that can be freed after the call

SASL_IPLOCALPORT   A string that describes the local ip and port in the form a.b.c.d:p
                       or [e:f:g:h:i:j:k:l]:port or one of the older forms, a.b.c.d;p
                       or e:f:g:j:i:j:k:l;port

SASL_IPREMOTEPORT   A string that describes the remote ip and port in the form
                       a.b.c.d:p or [e:f:g:h:i:j:k:l]:port or one of the older forms,
                       a.b.c.d;p or e:f:g:j:i:j:k:l;port

Parameters *conn*   The SASL connection context

propnum   The identifier for the property requested

pvalue   Contains a pointer to the data. The application must ensure that the data type is
            correct, or the application can crash.

Return Values sasl_setprop() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The call to sasl_setprop() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also   sasl_errors(3SASL), attributes(5)

Name sasl_utf8verify – encode base64 string

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/saslutil.h>

int sasl_utf8verify(const char *str, unsigned *len*);

Description Use the sasl_utf8verify() interface to verify that a string is valid UTF-8 and does not contain NULL, a carriage return, or a linefeed. If *len* ==0, strlen(*str*) will be used.

Parameters *str*   A string

len   The length of the string

Return Values sasl_utf8verify() returns an integer that corresponds to a SASL error code.

Errors SASL_OK   The call to sasl_utf8verify() was successful.

SASL_BADPROT   There was invalid UTF-8, or an error was found.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also attributes(5)

Name sasl_verifyfile_t – the SASL file verification callback function

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
```
#include <sasl/sasl.h>

typedef enum {
        SASL_VRFY_PLUGIN,      /* a DLL/shared library plugin */
        SASL_VRFY_CONF,        /* a configuration file */
        SASL_VRFY_PASSWD,      /* a password storage file */
        SASL_VRFY_OTHER        /* some other file type */
} sasl_verify_tyep_t

int sasl_verifyfile_t(void *context, const char *file,
    sasl_verifyfile_t type);
```

Description Use the sasl_verifyfile_t() callback function check whether a given file can be used by the SASL library. Applications use sasl_verifyfile_t() to check the environment to ensure that plugins or configuration files cannot be written to.

Parameters *context*   The context from the callback record

file   The full path of the file to verify

type   The type of the file

Return Values Like other SASL callback functions, sasl_verifyfile_t() returns an integer that corresponds to a SASL error code. See <sasl.h> for a complete list of SASL error codes.

Errors SASL_OK   The call to sasl_verifyfile_t() was successful.

See sasl_errors(3SASL) for information on SASL error codes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also sasl_errors(3SASL), attributes(5)

Name sasl_version – get SASL library version information

Synopsis cc [*flag* ...] *file* ... -lsasl   [*library* ...]
#include <sasl/sasl.h>

void sasl_version(const char ***implementation*, int **version*);

Description Use the sasl_version() interface to obtain the version of the SASL library.

Parameters *implementation*   A vendor-defined string that describes the implementation. The value of
*implementation*returned is Sun SASL.

version   A vendor-defined represetation of the version number.

Return Values The sasl_version() interface has no return values.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/library/security/libsasl
Interface Stability	Committed
MT-Level	MT–Safe

See Also attributes(5)

Name sctp_bindx – add or remove IP addresses to or from an SCTP socket

Synopsis cc [*flag*...] *file*... -lsocket -lnsl -lsctp [*library*...]
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

int sctp_bindx(int *sock*, void **addrs*, int *addrcnt*, int *flags*);

Description The sctp_bindx() function adds or removes addresses to or from an SCTP socket. If *sock* is
an Internet Protocol Version 4 (IPv4) socket, *addrs* should be an array of sockaddr_in
structures containing IPv4 addresses. If *sock* is an Internet Protocol Version 6 (IPv6) socket,
addrs should be an array of sockaddr_in6 structures containing IPv6 or IPv4-mapped IPv6
addresses. The *addrcnt* is the number of array elements in *addrs*. The family of the address
type is used with *addrcnt* to determine the size of the array.

The *flags* parameter is a bitmask that indicates whether addresses are to be added or removed
from a socket. The *flags* parameter is formed by bitwise OR of zero or more of the following
flags:

SCTP_BINDX_ADD_ADDR Indicates that addresses from *addrs* should be added to the SCTP
socket.

SCTP_BINDX_REM_ADDR Indicates that addresses from *addrs* should be removed from the
SCTP socket.

These two flags are mutually exclusive. If *flags* is formed by a bitwise OR of both
SCTP_BINDX_ADD_ADDR and SCTP_BINDX_REM_ADDR, the sctp_bindx() function will fail.

Prior to calling sctp_bindx() on an SCTP endpoint, the endpoint should be bound using
bind(3SOCKET). On a listening socket, a special INADDR_ANY value for IP or an unspecified
address of all zeros for IPv6 can be used in *addrs* to add all IPv4 or IPv6 addresses on the
system to the socket. The sctp_bindx() function can also be used to add or remove addresses
to or from an established association. In such a case, messages are exchanged between the
SCTP endpoints to update the address lists for that association if both endpoints support
dynamic address reconfiguration.

Return Values Upon successful completion, the sctp_bindx() function returns 0. Otherwise, the function
returns -1 and sets errno to indicate the error.

Errors The sctp_bindx() call fails under the following conditions.

EBADF The *sock* argument is an invalid file descriptor.

ENOTSOCK The *sock* argument is not a socket.

EINVAL One or more of the IPv4 or IPv6 addresses is invalid.

EINVAL The endpoint is not bound.

EINVAL   The last address is requested to be removed from an established association.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   bind(3SOCKET), in.h(3HEAD), libsctp(3LIB), listen(3SOCKET), sctp_freeladdrs(3SOCKET), sctp_freepaddrs(3SOCKET), sctp_getladdrs(3SOCKET), sctp_getpaddrs(3SOCKET), socket(3SOCKET), inet(7P), inet6(7P), ip(7P), ip6(7P), sctp(7P)

Notes   IPv4-mapped addresses are not recommended.

Name sctp_connectx – connect an SCTP oscket

Synopsis cc [flag...] file... -lsocket -lnsl -lsctp [library...]
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

int sctp_connectx(int *sd*, struct sockaddr **addrs*, int *addrcnt*,
     sctp_assoc_t **aid*);

Parameters *sd*   The socket descriptor.

addrs   If *sd* is an IPv4 socket, *addrs* should be an array of sockaddr_in structures
              containing IPv4 addresses. If *sd* is an IPv6 socket, *addrs* should be an array of
              sockaddr_in6 structures containing IPv6 or IPv4-mapped IPv6 addresses.

addrcnt   The number of addresses in the array *addrs*.

aid   If the function returns successfully, the association identifier for the newly created
              association is returned in *aid*. This parameter is applicable only to one-to-many
              style SCTP sockets.

Description The sctp_connectx() function requests an SCTP association to be made on a socket. This is
similar to connect(3SOCKET) except that an array of peer addresses can be given.

Much like sctp_bindx(3SOCKET), this function allows a caller to specify multiple addresses
at which a peer can be reached. The SCTP stack tries each addresses in the array in a round
robin fashion to set up the association. Note that the list of addresses passed in is only used for
setting up the association. It does not necessarily equal the set of addresses the peer uses for the
resulting association. If the caller wants to find out the set of peer addresses, it must use
sctp_getpaddrs(3SOCKET) to retrieve them after the association has been set up.

Return Values Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to
indicate the error.

Errors The sctp_connectx() function will fail if:

EADDRINUSE   The address is already in use.

EADDRNOTAVAIL   No local address is available for this operation.

EAFNOSUPPORT   Addresses in the specified address family cannot be used with this socket.

EALREADY   The socket is non-blocking and a previous connection attempt has not yet
                    been completed.

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

ECONNREFUSED	The attempt to connect was forcefully rejected. The calling program should use connect(3SOCKET) to close the socket descriptor, and issue another socket(3SOCKET) call to obtain a new descriptor before making another attempt.
EFAULT	A parameter can not be accessed.
EINTR	The connect attempt was interrupted before it is completed. The attempt will be established asynchronously.
EINVAL	A parameter provided is invalid for this operation.
EISCONN	The socket is already connected.
ENETUNREACH	The network is not reachable from this host.
ENOBUFS	Insufficient memory is available to complete the operation.
EOPNOTSUPP	The operation is not supported in this type of socket.
ETIMEDOUT	The attempt timed out.
EWOULDBLOCK	The socket is marked as non-blocking and the requested operation would block.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   connect(3SOCKET), libsctp(3LIB), sctp_bindx(3SOCKET), sctp_getpaddrs(3SOCKET), socket(3SOCKET), attributes(5), sctp(7P)

Name sctp_getladdrs, sctp_freeladdrs – returns all locally bound addresses on an SCTP socket

Synopsis cc [*flag*...] *file*... -lsocket -lnsl -lsctp [*library*...]
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

int sctp_getladdrs(int *sock*, sctp_assoc_t *id*, void ***addrs*);

void sctp_freeladdrs(void *addrs);

Description The sctp_getladdrs() function queries addresses to which an SCTP socket is bound. The
sctp_freeladdrs() function releases resources that are allocated to hold the addresses.

The sctp_getladdrs() function returns all the locally bound addresses on the SCTP socket
sock. On completion *addrs* points to a dynamically allocated array of sockaddr_in structures
for an Internet Protocol (IPv4) socket or an array of sockaddr_in6 structures for an Internet
Protocol Version 6 (IPv6) socket. The *addrs* parameter must not be NULL. For an IPv4 SCTP
socket, the addresses returned in the sockaddr_in structures are IPv4 addresses. For an IPv6
SCTP socket, the addresses in the sockaddr_in6 structures can be IPv6 addresses or
IPv4-mapped IPv6 addresses.

If *sock* is a one-to-many style SCTP socket, *id* specifies the association of interest. A value of 0
to *id* returns locally-bound addresses regardless of a particular association. If *sock* is a
one-to-one style SCTP socket, *id* is ignored.

The sctp_freeladdrs() function frees the resources allocated by sctp_getladdrs(). The
addrs parameter is the array of addresses allocated by sctp_getladdrs().

Return Values Upon successful completion, the sctp_getladdrs() function returns the number of
addresses in the *addrs* array. Otherwise, the function returns -1 and sets errno to indicate the
error.

Errors The sctp_getladdrs() call fails under the following conditions.

EBADF The *sock* argument is an invalid file descriptor.

ENOTSOCK The *sock* argument is not a socket.

EINVAL The *addrs* argument is NULL.

EINVAL The *id* argument is an invalid socket.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   bind(3SOCKET), in.h(3HEAD), libsctp(3LIB), sctp_freepaddrs(3SOCKET),
sctp_getpaddrs(3SOCKET), socket(3SOCKET), attributes(5), inet(7P), inet6(7P),
ip(7P), ip6(7P), sctp(7P)

Name sctp_getpaddrs, sctp_freepaddrs – returns all peer addresses on an SCTP association

Synopsis cc [*flag*...] *file*... -lsocket -lnsl -lsctp [*library*...]
```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

int sctp_getpaddrs(int sock, sctp_assoc_t id, void **addrs);

void sctp_freepaddrs(void *addrs);
```

Description The sctp_getpaddrs() queries the peer addresses in an SCTP association. The sctp_freepaddrs() function releases resources that are allocated to hold the addresses.

The sctp_getpaddrs() function returns all the peer addresses in the SCTP association identified by *sock*. On completion *addrs* points to a dynamically allocated array of sockaddr_in structures for an Internet Protocol (IPv4) socket or an array of sockaddr_in6 structures for an Internet Protocol Version 6 (IPv6) socket. The *addrs* parameter must not be NULL. For an IPv4 SCTP socket, the addresses returned in the sockaddr_in structures are IPv4 addresses. For an IPv6 SCTP socket, the addresses in the sockaddr_in6 structures can be IPv6 addresses or IPv4-mapped IPv6 addresses.

If *sock* is a one-to-many style SCTP socket, *id* specifies the association of interest. If *sock* is a one-to-one style SCTP socket, *id* is ignored.

The sctp_freepaddrs() function frees the resources allocated by sctp_getpaddrs(). The *addrs* parameter is the array of addresses allocated by sctp_getpaddrs().

Return Values Upon successful completion, the sctp_getpaddrs() function returns the number of addresses in the *addrs* array. Otherwise, the function returns -1 and sets errno to indicate the error.

Errors The sctp_getpaddrs() succeeds unless one of the following conditions exist.

EBADF   The *sock* argument is an invalid file descriptor.

ENOTSOCK The *sock* argument is not a socket.

EINVAL   The *addrs* argument is NULL.

EINVAL   The *id* argument is an invalid association identifier for a one-to-many style STP socket.

ENOTCONN The specified socket is not connected.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   bind(3SOCKET), in.h(3HEAD), libsctp(3LIB), sctp_freeladdrs(3SOCKET), sctp_getladdrs(3SOCKET), socket(3SOCKET), attributes(5), inet(7P), inet6(7P), ip(7P), ip6(7P), sctp(7P)

Name sctp_opt_info – examine SCTP level options for an SCTP endpoint

Synopsis cc [*flag*...] *file*... -lsocket -lnsl -lsctp [*library*...]
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

```
int sctp_opt_info(int sock, sctp_assoc_t id, int opt, void *arg,
    socklen_t *len);
```

Description The sctp_opt_info() returns SCTP level options associated with the SCTP socket *sock*. If *sock* is a one-to-many style socket, *id* refers to the association of interest. If *sock* is a one-to-one socket or if *sock* is a branched-off one-to-many style socket, *id* is ignored. The *opt* parameter specifies the SCTP option to get. The *arg* structure is an option-specific structure buffer allocated by the caller. The *len* parameter is the length of the option specified.

Following are the currently supported values for the *opt* parameter. When one of the options below specifies an association *id*, the *id* is relevant for only one-to-many style SCTP sockets. The associatation *id* can be ignored for one-to-one style or branched-off one-to-many style SCTP sockets.

SCTP_RTOINFO
Returns the protocol parameters used to initialize and bind retransmission timeout (RTO) tunable. The following structure is used to access these parameters:

```
struct sctp_rtoinfo {
        sctp_assoc_t          srto_assoc_id;
        uint32_t              srto_initial;
        uint32_t              srto_max;
        uint32_t              srto_min;
};
where:
        srto_assoc_id         Association ID specified by the caller
        srto_initial          Initial RTO value
        srto_max              Maximum value for the RTO
        srto_min              Minimum value for the RTO
```

SSCTP_ASSOCINFO
Returns association-specific parameters. The following structure is used to access the parameters:

```
struct sctp_assocparams {
        sctp_assoc_t          sasoc_assoc_id;
        uint16_t              sasoc_asocmaxrxt;
        uint16_t              sasoc_number_peer_destinations;
        uint32_t              sasoc_peer_rwnd;
        uint32_t              sasoc_local_rwnd;
        uint32_t              sasoc_cookie_life;
};
where:
```

```
               srto_assoc_id          Association ID specified by the caller
               sasoc_asocmaxrxt       Maximum retransmission count for
                                      the association
               sasoc_number_peer_destinations
                                      Number of addresses the peer has
               sasoc_peer_rwnd        Current value of the peer's
                                      receive window
               sasoc_local_rwnd       Last reported receive window sent
                                      to the peer
               sasoc_cookie_life      Association cookie lifetime used
                                      when issuing cookies
```

All parameters with time values are in milliseconds.

SCTP_DEFAULT_SEND_PARAM

Returns the default set of parameters used by the sendto() function on this association.
The following structure is used to access the parameters:

```
struct sctp_sndrcvinfo {
        uint16_t              sinfo_stream;
        uint16_t              sinfo_ssn;
        uint16_t              sinfo_flags;
        uint32_t              sinfo_ppid;
        uint32_t              sinfo_context;
        uint32_t              sinfo_timetolive;
        uint32_t              sinfo_tsn;
        uint32_t              sinfo_cumtsn;
        sctp_assoc_t          sinfo_assoc_id;
};
where:
        sinfo_stream          Default stream for sendmsg()
        sinfo_ssn             Always returned as 0
        sinfo_flags           Default flags for sendmsg()
                              that include the following:
                              MSG_UNORDERED
                              MSG_ADDR_OVER
                              MSG_ABORT
                              MSG_EOF
                              MSG_PR_SCTP
        sinfo_ppid            Default payload protocol identifier
                              for sendmsg()
        sinfo_context         Default context for sendmsg()
        sinfo_timetolive      Time to live in milliseconds for a
                              message on the sending side.
                              The message expires if the sending
                              side does not start the first
                              transmission for the message within
                              the specified time period. If the
                              sending side starts the first
```

```
                               transmission before the time period
                               expires, the message is sent as a
                               normal reliable message. A value of
                               0 indicates that the message does not
                               expire. When MSG_PR_SCTP is set in
                               sinfo_flags, the message expires if
                               it is not acknowledged within the
                               time period.
        sinfo_tsn              Always returned as 0
        sinfo_cumtsn           Always returned as 0
        sinfo_assoc_id         Association ID specified by the caller
```

SCTP_PEER_ADDR_PARAMS

Returns the parameters for a specified peer address of the association. The following structure is used to access the parameters:

```
struct sctp_paddrparams {
        sctp_assoc_t            spp_assoc_id;
        struct sockaddr_storage spp_address;
        uint32_t                spp_hbinterval;
        uint16_t                spp_pathmaxrxt;
        uint32_t                spp_pathmtu;
        uint32_t                spp_flags;
        uint32_t                spp_ipv6_flowlabel;
        uint8_t                 spp_ipv4_tos;
};
```

where:

spp_assoc_id
   Association ID specified by the caller.

spp_address
   Peer's address.

spp_hbinterval
   Heartbeat interval in milliseconds.

spp_pathmaxrxt
   Maximum number of retransmissions to an address before it is considered unreachable.

spp_pathmtu
   The current path MTU of the peer address. It is the number of bytes available in an SCTP packet for chunks. Providing a value of 0 does not change the current setting. If a positive value is provided and SPP_PMTUD_DISABLE is set in *spp_flags*, the given value is used as the path MTU. If SPP_PMTUD_ENABLE is set in *spp_flags*, the *spp_pathmtu* field is ignored.

spp_ipv6_flowlabel
   This field is used in conjunction with the SPP_IPV6_FLOWLABEL flag. This setting has precedence over any IPv6 layer setting.

spp_ipv4_tos

This field is used in conjunction with the `SPP_IPV4_TOS` flag. This setting has precedence over any IPv4 layer setting.

spp_flags

These flags are used to control various features on an association. The flag field is a bit mask which may contain zero or more of the following options:

`SPP_HB_ENABLE`

Enable heartbeats on the specified address.

`SPP_HB_DISABLE`

Disable heartbeats on the specified address. Note that `SPP_HB_ENABLE` and `SPP_HB_DISABLE` are mutually exclusive, only one of these two should be specified. Enabling both fields will have undetermined results.

`SPP_HB_DEMAND`

Request a user initiated heartbeat to be made immediately. This must not be used in conjunction with a wildcard address.

`SPP_HB_TIME_IS_ZERO`

Specifies that the time for heartbeat delay is to be set to the value of 0 milliseconds.

`SPP_PMTUD_ENABLE`

This field will enable PMTU discovery upon the specified address.

`SPP_PMTUD_DISABLE`

This field will disable PMTU discovery upon the specified address. Note that if the address field is empty then all addresses on the association are affected. Note also that `SPP_PMTUD_ENABLE` and `SPP_PMTUD_DISABLE` are mutually exclusive. Enabling both will have undetermined results.

`SPP_IPV6_FLOWLABEL`

Setting this flag enables the setting of the IPV6 flowlabel value. The value is obtained in the *spp_ipv6_flowlabel* field.

Upon retrieval, this flag will be set to indicate that the *spp_ipv6_flowlabel* field has a valid value returned. If a specific destination address is set (in the *spp_address* field), the value returned is that of the address. If just an association is specified (and no address), then the association's default flowlabel is returned. If neither an association nor a destination is specified, then the socket's default flowlabel is returned. For non IPv6 sockets, this flag will be left cleared.

`SPP_IPV4_TOS`

Setting this flag enables the setting of the IPV4 TOS value associated with either the association or a specific address. The value is obtained in the *spp_ipv4_tos* field.

Upon retrieval, this flag will be set to indicate that the *spp_ipv4_tos* field has a valid value returned. If a specific destination address is set when called (in the *spp_address* field) then that specific destination address' TOS value is returned. If just an

association is specified then the association default TOS is returned. If neither an association nor a destination is specified, then the sockets default TOS is returned.

SCTP_STATUS

Returns the current status information about the association. The following structure is used to access the parameters:

```
struct sctp_status {
        sctp_assoc_t            sstat_assoc_id;
        int32_t                 sstat_state;
        uint32_t                sstat_rwnd;
        uint16_t                sstat_unackdata;
        uint16_t                sstat_penddata;
        uint16_t                sstat_instrms;
        uint16_t                sstat_outstrms;
        uint16_t                sstat_fragmentation_point;
        struct sctp_paddrinfo   sstat_primary;
};
where:

        sstat_assoc_id          Association ID specifed by the caller
        sstat_state             Current state of the association
                                which might be one of the following:

                                SCTP_CLOSED
                                SCTP_BOUND
                                SCTP_LISTEN
                                SCTP_COOKIE_WAIT
                                SCTP_COOKIE_ECHOED
                                SCTP_ESTABLISHED
                                SCTP_SHUTDOWN_PENDING
                                SCTP_SHUTDOWN_SENT
                                SCTP_SHUTDOWN_RECEIVED
                                SCTP_SHUTDOWN_ACK_SENT
        sstat_rwnd              Current receive window of the
                                association peer
        sstat_unackdata         Number of unacked DATA chunks
        sstat_penddata          Number of DATA chunks pending
                                receipt
        sstat_instrms           Number of inbound streams
        sstat_outstrms          Number of outbound streams
        sstat_fragmentation_point
                                Size at which SCTP fragmentation occurs
        sstat_primary           Information about the primary
                                peer address

        sstat_primary has the following structure
                        struct sctp_paddrinfo {
                                sctp_assoc_t            spinfo_assoc_id;
```

```
                             struct sockaddr_storage spinfo_address;
                             int32_t                 spinfo_state;
                             uint32_t                spinfo cwnd;
                             uint32_t                spinfo_srtt;
                             uint32_t                spinfo_rto;
                             uint32_t                spinfo_mtu;
                      };
                      where:
                      spinfo_assoc_id        Association ID
                                             specified by
                                             the caller
                      spinfo_address         Primary peer
                                             address
                      spinfo_state           State of the peer
                                             address:
                                             SCTP_ACTIVE or
                                             SCTP_INACTIVE
                      spinfo_cwnd            Congestion window
                                             of the peer
                                             address
                      spinfo_srtt            Smoothed round-trip
                                             time calculation of
                                             the peer address
                      spinfo_rto             Current retransmission
                                             timeout value of the
                                             peer address in
                                             milliseconds
                      spinfo_mtu             P-MTU of the address
```

Return Values Upon successful completion, the sctp_opt_info() function returns 0. Otherwise, the function returns -1 and sets errno to indicate the error.

Errors The sctp_opt_info() call fails under the following conditions.

EBADF   The *sock* argument is an invalid file descriptor.

ENOTSOCK   The *sock* argument is not a socket.

EINVAL   The association *id* is invalid for a one-to-many style SCTP socket.

EINVAL   The input buffer length is insufficient for the option specified.

EINVAL   The peer address is invalid or does not belong to the association.

EAFNOSUPPORT   The address family for the peer's address is other than AF_INET or AF_INET6.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   in.h(3HEAD), libsctp(3LIB), getsockopt(3SOCKET), setsockopt(3SOCKET), socket(3SOCKET), inet(7P), inet6(7P), ip(7P), ip6(7P), sctp(7P)

Name sctp_peeloff – branch off existing association from a one-to-many SCTP socket to create a one-to-one STP socket

Synopsis cc [*flag*...] *file*... -lsocket -lnsl -lsctp [*library*...]
```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

int sctp_peeloff(int sock, sctp_assoc_t id);
```

Description The sctp_peeloff() function branches off an existing association from a one-to-many style SCTP socket into a separate socket file descriptor. The resulting branched-off socket is a one-to-one style SCTP socket and is confined to operations allowed on a one-to-one style SCTP socket.

The *sock* argument is a one-to-many socket. The association specified by the *id* argument is branched off *sock*.

Return Values Upon successful completion, the sctp_peeloff() function returns the file descriptor that references the branched-off socket. The function returns -1 if an error occurs.

Errors The sctp_peeloff() function fails under the following conditions.

EOPTNOTSUPP   The *sock* argument is not a one-to-many style SCTP socket.

EINVAL   The *id* is 0 or greater than the maximum number of associations for *sock*.

EMFILE   Failure to create a new user file descriptor or file structure.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also in.h(3HEAD), libsctp(3LIB), socket(3SOCKET), sctp(7P)

Name sctp_recvmsg – receive message from an SCTP socket

Synopsis cc [*flag*...] *file*... -lsocket -lnsl -lsctp [*library*...]
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

ssize_t sctp_recvmsg(int *s*, void **msg*, size_t *len*, struct sockaddr **from*,
    socklen_t **fromlen*, struct sctp_sndrcvinfo **sinfo*, int **msg_flags*);

Description The sctp_recvmsg() function receives a message from the SCTP endpoint *s*.

In addition to specifying the message buffer *msg* and the length *len* of the buffer, the following parameters can be set:

from   Pointer to an address, filled in with the sender's address

fromlen   Size of the buffer associated with the *from* parameter

sinfo   Pointer to an sctp_sndrcvinfo structure, filled in upon the receipt of the message

msg_flags   Message flags such as MSG_CTRUNC, MSG_NOTIFICATION, MSG_EOR

The *sinfo* parameter is filled in only when the caller has enabled sctp_data_io_events by calling setsockopt() with the socket option SCTP_EVENTS.

Return Values Upon successful completion, the sctp_recvmsg() function returns the number of bytes received. The function returns -1 if an error occurs.

Errors The sctp_recvmsg() function fails under the following conditions.

EBADF   The *s* argument is an invalid file descriptor.

ENOTSOCK   The *s* argument is not a socket.

EOPNOTSUPP   MSG_OOB is set as a flag.

ENOTCONN   There is no established association.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also accept(3SOCKET), bind(3SOCKET), connect(3SOCKET), in.h(3HEAD), libsctp(3LIB), listen(3SOCKET), recvmsg(3SOCKET), sctp_opt_info(3SOCKET), setsockopt(3SOCKET), socket(3SOCKET), socket.h(3HEAD), sctp(7P)

Name   sctp_recvv – receive a message from an SCTP socket

Synopsis   cc [*flag*...] *file*... -lsocket -lnsl -lsctp [*library*...]
```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

ssize_t sctp_recvv(int sd, const struct iovec *iov, int iovlen,
     struct sockaddr *from, socklen_t *fromlen, void *info,
     socklen_t *infolen, unsigned int *infotype, int *flags);
```

Parameters   *sd*   The socket descriptor.

iov   The scatter buffer containing the received message.

iovlen   The number of elements in *iov*.

from   A pointer to a buffer to be filled with the sender address of the received message.

fromlen   The size of the from buffer. Upon return, it is set to the actual size of the sender's address.

info   A pointer to the buffer to hold the attributes of the received message. The structure type of info is determined by the *infotype* parameter.

infolen   The size of the info buffer. Upon return, it is set to the actual size of returned info buffer.

infotype   The type of the info buffer. The defined values are:

SCTP_RECVV_NOINFO
  If both SCTP_RECVRCVINFO and SCTP_RECVNXTINFO options are not enabled, no attribute will be returned. If only the SCTP_RECVNXTINFO option is enabled but there is no next message in the buffer, there will also no attribute be returned. In these cases, *infotype* will be set to SCTP_RECVV_NOINFO.

SCTP_RECVV_RCVINFO
  The type of info is struct sctp_rcvinfo and the attribute is about the received message.

SCTP_RECVV_NXTINFO
  The type of info is struct sctp_nxtinfo and the attribute is about the next message in receive buffer. This is the case when only the SCTP_RECVNXTINFO option is enabled and there is a next message in the buffer.

SCTP_RECVV_RN
  The type of info is struct sctp_recvv_rn. The *recvv_rcvinfo* field is the attribute of the received message and the *recvv_nxtinfo* field is the attribute of the next message in buffer. This is the case when both SCTP_RECVRCVINFO and SCTP_RECVNXTINFO options are enabled and there is a next message in the receive buffer.

flags   Flag for receive as in recvmsg(3SOCKET). On return, its value will be different from what was set in to the call. It has the same value as *rcv_flags*.

Description The sctp_recvv() function provides an extensible way for the SCTP stack to pass up different SCTP attributes associated with a received message to an application.

There are two types of attributes which can be returned by this call: the attribute of the received message and the attribute of the next message in receive buffer. The caller enables the SCTP_RECVRCVINFO and SCTP_RECVNXTINFO socket option to receive these attributes respectively. Attributes of the received message are returned in struct sctp_rcvinfo and attributes of the next message are returned in struct sctp_nxtinfo. If both options are enabled, both attributes are returned using the following structure.

```
struct sctp_recvv_rn {
    struct sctp_rcvinfo    recvv_rcvinfo;
    struct sctp_nxtinfo    recvv_nxtinfo;
};
```

The sctp_rcvinfo structure is defined as:

```
struct sctp_rcvinfo {
    uint16_t       rcv_sid;
    uint16_t       rcv_ssn;
    uint16_t       rcv_flags;
    uint32_t       rcv_ppid;
    uint32_t       rcv_tsn;
    uint32_t       rcv_cumtsn;
    uint32_t       rcv_context;
    sctp_assoc_t   rcv_assoc_id;
};
```

rcv_sid   The stream number of the received message.

rcv_ssn   The stream sequence number that the peer endpoint assigned to the DATA chunk of this message. For fragmented messages, this is the same number for all deliveries of the message (if more than one sctp_recvv()) is needed to read the message).

rcv_flags   This field may be set to following values:

               SCTP_UNORDERED   This flag is set when the message was sent unordered.

rcv_ppid   This value is the same information that is passed by the peer socket to its SCTP stack. The SCTP stack performs no byte order modification of this field.

rcv_tsn   The transmission sequence number that the peer endpoint assigned to the received message.

rcv_cumtsn	The current cumulative transmission sequence number of the association known to the SCTP stack.
rcv_assoc_id	The association identifier of the association of the received message. This field applies only to a one-to-many style socket.
rcv_context	This value is an opaque 32 bit context datum that was set by the caller with the SCTP_CONTEXT socket option. This value is passed back to the upper layer if an error occurs on the transmission of a message and is retrieved with each undelivered message.

The sctp_nxtinfo structure is defined as follows:

```
struct sctp_nxtinfo {
    uint16_t      nxt_sid;
    uint16_t      nxt_flags;
    uint32_t      nxt_ppid;
    size_t        nxt_length;
    sctp_assoc_t  nxt_assoc_id;
};
```

nxt_sid	The stream number of the next message.
nxt_flags	This field can contain any of the following flags and is composed of a bitwise OR of the following values:

> SCTP_UNORDERED
> The next message was sent unordered.

> SCTP_COMPLETE
> The entire message has been received and is in the socket buffer. This flag has special implications with respect to the *nxt_length* field.

> SCTP_NOTIFICATION
> The next message is not a user message but instead is a notification.

nxt_ppid	This value is the same information that was passed by the peer socket to its SCTP stack when sending the next message. The SCTP stack performs no byte order modification of this field.
nxt_length	The length of the message currently received in the socket buffer. This might not be the entire length of the next message since a partial delivery may be in progress. Only if the flag SCTP_COMPLETE is set in the *nxt_flags* field does this field represent the entire next message size.
nxt_assoc_id	The association identifier of the association of the next message. This field applies only to a one-to-many style socket.

New structures can be defined to hold new types of attributes. The new structures do not need to be based on struct sctp_recvv_rn or struct sctp_rcvinfo.

Return Values Upon successful completion, the sctp_recvv() function returns the number of bytes received. The function returns -1 if an error occurs and *errno* is set to indicate the error.

Errors The sctp_recvv() function will fail if:

EBADF	The *sd* parameter is not a valid file descriptor.
EFAULT	A parameter can not be accessed.
EINTR	The operation was interrupted by delivery of a signal before any data could be buffered to be sent.
	The operation was interrupted by delivery of a signal before any data is available to be received.
EINVAL	A parameter provided is invalid for this operation.
ENOBUFS	Insufficient memory is available to complete the operation.
EWOULDBLOCK	The socket is marked as non-blocking, and the requested operation would block.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libsctp(3LIB), recvmsg(3SOCKET), setsockopt(3SOCKET), socket(3SOCKET), attributes(5), sctp(7P)

Name sctp_send – send message from an SCTP socket

Synopsis cc [*flag*...] *file*... -lsocket -lnsl -lsctp [*library*...]
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

ssize_t sctp_send(int *s*, const void **msg*, size_t **len*,
    const struct sctp_sndrcvinfo **sinfo*, int *flags*);

Description The sctp_send() function sends messages from one-to-one and one-to-many style SCTP
endpoints. The following parameters can be set:

s   Socket created by socket(3SOCKET)

msg   Message to be sent

len   Size of the message to be sent in bytes

The caller completes the *sinfo* parameter with values used to send a message. Such values
might include the stream number, payload protocol identifier, time to live, and the SCTP
message flag and context. For a one-to-many socket, the association ID can be specified in the
sinfo parameter to send a message to the association represented in the ID.

Flags supported for sctp_send() are reserved for future use.

Return Values Upon successful completion, the sctp_send() function returns the number of bytes sent. The
function returns -1 if an error occurs.

Errors The sctp_send() function fails under the following conditions.

EBADF   The *s* argument is an invalid file descriptor.

ENOTSOCK   The *s* argument is not a socket.

EOPNOTSUPP   MSG_ABORT or MSG_EOF is set in the sinfo_flags field of sinfo for a
                 one-to-one style SCTP socket.

EPIPE   The socket is shutting down and no more writes are allowed.

EAGAIN   The socket is non-blocking and the transmit queue is full.

ENOTCONN   There is no established association.

EINVAL   Control message length is incorrect.

EINVAL   Specified destination address does not belong to the association.

EINVAL   The stream_no is outside the number of outbound streams supported by
                 the association.

EAFNOSUPPORT   Address family of the specified destination address is other than AF_INET
                 or AF_INET6.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   accept(3SOCKET), bind(3SOCKET), connect(3SOCKET), in.h(3HEAD), libsctp(3LIB), listen(3SOCKET), sctp_sendmsg(3SOCKET), sendmsg(3SOCKET), socket(3SOCKET), socket.h(3HEAD), sctp(7P)

Name sctp_sendmsg – send message from an SCTP socket

Synopsis cc [*flag*...] *file*... -lsocket -lnsl -lsctp [*library*...]
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

ssize_t sctp_sendmsg(int *s*, const void **msg*, size_t *len*,
    const struct sockaddr **to*, socklen_t *tolen*, uint32_t *ppid*,
    uint32_t *flags*, uint16_t *stream_no*, uint32_t *timetolive*,
    uint32_t *context*);

Description The sctp_sendmsg() function sends a message from the SCTP endpoint *s*.

In addition to specifying *msg* as the message buffer and *len* as the length of the buffer, the following parameters can be set:

to	Destination address
tolen	Length of the destination address
ppid	Application-specified payload protocol identifier
stream_no	Target stream for the message
timetolive	Time period in milliseconds after which the message expires if transmission for the message has not been started. A value of 0 indicates that the message does not expire. When the MSG_PR_SCTP flag is set the message expires, even if transmission has started, unless the entire message is transmitted within the *timetolive* period.
context	Value returned when an error occurs in sending a message

The *flags* parameter is formed from the bitwise OR of zero or more of the following flags:

MSG_UNORDERED	This flag requests un-ordered delivery of the message. If this flag is clear the message is considered an ordered send.
MSG_ABORT	When set, this flag causes the specified association to abort by sending an ABORT to the peer. The flag is used only for one-to-many style SCTP socket associations.
MSG_EOF	When set, this flag invokes a graceful shutdown on a specified association. The flag is used only for one-to-many style SCTP socket associations.
MSG_PR_SCTP	This flag indicates that the message is treated as partially reliable. The message expires unless the entire message is successfully transmitted within the time period specified in the *timetolive* parameter.

MSG_PR_SCTP implements *timed reliability* service for SCTP messages. As yet, no common standard has been defined for the service and the interface is considered unstable.

The initial call to sctp_sendmsg() can be used to create an association, but it cannot be used subsequently on an existing association. Since sctp_sendmsg() always uses 0 internally as the association ID, it is not suitable for use on one-to-many sockets.

Return Values Upon successful completion, the sctp_sendmsg() function returns the number of bytes sent. The function returns -1 if an error occurs.

Errors The sctp_sendmsg() function will fail if:

EBADF   The *s* argument is an invalid file descriptor.

ENOTSOCK   The *s* argument is not a socket.

EOPNOTSUPP   MSG_OOB is set as a *flag*.

EOPNOTSUPP   MSG_ABORT or MSG_EOF is set on a one-to-one style SCTP socket.

EPIPE   The socket is shutting down and no more writes are allowed.

EAGAIN   The socket is non-blocking and the transmit queue is full.

ENOTCONN   There is no established association.

EINVAL   Control message length is incorrect.

EINVAL   Specified destination address does not belong to the association.

EAFNOSUPPORT   Address family of the specified destination address is other than AF_INET or AF_INET6.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also accept(3SOCKET), bind(3SOCKET), connect(3SOCKET), in.h(3HEAD), libsctp(3LIB), listen(3SOCKET), sendmsg(3SOCKET), socket(3SOCKET), socket.h(3HEAD), attributes(5), sctp(7P)

Name sctp_sendv – send a message to an SCTP socket

Synopsis cc [flag...] file... -lsocket -lnsl -lsctp [library...]
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

ssize_t sctp_sendv(int sd, const struct iovec *iov, int iovcnt,
    struct sockaddr *addrs, int addrcnt, void *info, socklen_t infolen,
    unsigned int infotype, int flags);

Parameters *sd*   The socket descriptor.

iov   The message to be sent. The data in the buffer are treated as one single user message.

iovcnt   The number of elements in *iov*.

addrs   An array of addresses to be used to set up an association or one single address to be used to send the message. Pass in NULL if the caller does not want to set up an association nor want to send the message to a specific address.

addrcnt   The number of addresses in the *addrs* array.

info   A pointer to the buffer containing the attribute associated with the message to be sent. The type is indicated by *info_type* parameter.

infolen   The length in bytes of info.

infotype Identifies the type of the information provided in *info*. The defined values are:

SCTP_SENDV_SNDINFO
    The *type* of info is struct sctp_sndinfo.

SCTP_SENDV_PRINFO
    The *type* of info is struct sctp_prinfo.

SCTP_SENDV_AUTHINFO
    The *type* of info is struct sctp_authinfo (not supported).

SCTP_SENDV_SPA
    The type of *info* is struct sctp_sendv_spa.

flags   The same flags as used by sendmsg(3SOCKET) (for example, MSG_DONTROUTE).

Description The sctp_sendv() function provides an extensible way for an application to communicate different send attributes to the SCTP stack when sending a message. This function can also be used to set up an association. The *addrs* array is similar to the *addrs* array used by sctp_connectx(3SOCKET).

There are three types of attributes which can be used to describe a message to be sent. They are represented by struct sctp_sndinfo, struct sctp_prinfo, and struct sctp_authinfo

(currently not supported). The following structure sctp_sendv_spa is defined to be used when more than one of the above attributes are needed to describe a message to be sent.

```
struct sctp_sendv_spa {
    uint32_t            sendv_flags;
    struct sctp_sndinfo   sendv_sndinfo;
    struct sctp_prinfo    sendv_prinfo;
    struct sctp_authinfo  sendv_authinfo;
};
```

The *sendv_flags* field holds a bitwise OR of SCTP_SEND_SNDINFO_VALID, SCTP_SEND_PRINFO_VALID, and SCTP_SEND_AUTHINFO_VALID, indicating whether the *sendv_sndinfo*, *sendv_prinfo*, and *sendv_authinfo* fields contain valid information.

The sctp_sndinfo structure is defined as:

```
struct sctp_sndinfo {
    uint16_t      snd_sid;
    uint16_t      snd_flags;
    uint32_t      snd_ppid;
    uint32_t      snd_context;
    sctp_assoc_t  snd_assoc_id;
};
```

snd_sid   This value holds the stream number to send the message to. If a sender specifies an invalid stream number, an error value is returned and the call fails.

snd_flags   This field is a bit wise OR of the following flags:

SCTP_UNORDERED
This flag requests the unordered delivery of the message.

SCTP_ADDR_OVER
This flag requests the SCTP stack to override the primary destination address and send the message to the given address in *addrs*. Only one address can be given is this case. If this flag is not specified and *addrs* is not NULL, this call is treated as a connect request. This flag is applicable to one-to-many style sockets only.

SCTP_ABORT
Setting this flag causes the specified association to be aborted by sending an ABORT message to the peer. The ABORT message will contain an error cause 'User Initiated Abort' with cause code 12. The specific information the cause of this error is provided in *msg_iov*.

SCTP_EOF
Setting this flag invokes the SCTP graceful shutdown procedures on the specified association. Graceful shutdown assures that all data queued by both endpoints is successfully transmitted before closing the association.

SCTP_SENDALL
This flag requests that the message is sent to all associations that are currently established on the socket. This flag is applicable to one-to-many style sockets only.

snd_ppid   An unsigned integer that is passed to the remote end in each user message (SCTP DATA chunk). The SCTP stack performs no byte order modification of this field. For example, if the DATA chunk has to contain a given value in network byte order, the SCTP user has to perform the htonl(3SOCKET) computation.

snd_context   This value is an opaque 32 bit context datum. It is passed back to the caller if an error occurs on the transmission of the message and is retrieved with each undelivered message.

snd_assoc_id When sending a message, this holds the identifier for the association which the message is sent to. When this call is used to set up an association, the association identifier of the newly created association is returned in this field. This field is applicable to one-to-many style sockets only.

The sctp_prinfo structure is defined as:

```
struct sctp_prinfo {
    uint16_t pr_policy;
    uint32_t pr_value;
};
```

pr_policy   This field specifies which partial reliability (PR-SCTP) policy is used to send the message. If it is SCTP_PR_SCTP_NONE, the message is sent reliably (the default is normal send). If it is SCTP_PR_SCTP_TTL, "timed reliability" as defined in RFC 3758 is used. In this case, the lifetime is provided in *pr_value*.

pr_value   The meaning of this field depends on the PR-SCTP policy specified by the *pr_policy* field. It is ignored when SCTP_PR_SCTP_NONE is specified. In case of SCTP_PR_SCTP_TTL, this field specifies the lifetime in milliseconds of the message.

When new send attributes are needed, new structures can be defined. Those new structures do not need to be based on any of the above defined structures.

The struct sctp_sndinfo attribute for one-to-many style sockets must always be used in order to specify the association the message is to be sent to. The only case where it is not needed is when this call is used to set up a new association.

The caller provides a list of addresses in the *addrs* parameter to set up an association. This function will behave like calling sctp_connectx(), first using the list of addresses, and then calling sendmsg() with the given message and attributes. For an one-to-many style socket, if a struct sctp_sndinfo attribute is provided, the *snd_assoc_id* field must be 0. When this

function returns, the *snd_assoc_id* field will contain the association identifier of the newly established association. The struct sctp_sndinfo attribute is not required to set up an association for one-to-many style sockets. If this attribute is not provided, the caller can enable the SCTP_ASSOC_CHANGE notification and use the SCTP_COMM_UP message to find out the association identifier.

If the caller wants to send the message to a specific peer address (overriding the primary address), it can provide the specific address in the *addrs* parameter and provide a struct sctp_sndinfo attribute with the *snd_flags* field set to SCTP_ADDR_OVER.

This function can also be used to terminate an association. The caller provides an sctp_sndinfo attribute with the *snd_flags* set to SCTP_EOF. In this case, the length of the message would be zero.

Sending a message using sctp_sendv() is atomic unless explicit EOR marking is enabled on the socket specified by *sd*.

Return Values   Upon successful completion, the number of bytes sent is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

Errors   The sctp_sendv() function will fail if:

EADDRINUSE	The address is already in use.
EADDRNOTAVAIL	No local address is available for this operation.
EAFNOSUPPORT	Addresses in the specified address family cannot be used with this socket.
EBADF	The *sd* parameter is not a valid file descriptor.
ECONNREFUSED	The attempt to connect was forcefully rejected. The calling program should close the socket descriptor using close(2) and issue another socket(3SOCKET) call to obtain a new descriptor before making another attempt.
EFAULT	A parameter can not be accessed.
EINTR	The operation was interrupted by delivery of a signal before any data could be buffered to be sent.
EINVAL	A parameter provided is invalid for this operation.
EMSGSIZE	The message is too large to be sent all at once.
ENETUNREACH	The network is not reachable from this host.
ENOBUFS	Insufficient memory is available to complete the operation.
EOPNOTSUPP	Operation not supported in this type of socket.
EPIPE	The peer end point has shutdown the association.

ETIMEDOUT   Attempt timed out.

EWOULDBLOCK   The socket is marked as non-blocking, and the requested operation
                   would block.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also htonl(3SOCKET), libsctp(3LIB), sctp_connectx(3SOCKET), sendmsg(3SOCKET),
socket(3SOCKET), attributes(5), sctp(7P)

Name sdp_add_origin, sdp_add_name, sdp_add_information, sdp_add_uri, sdp_add_email, sdp_add_phone, sdp_add_connection, sdp_add_bandwidth, sdp_add_repeat, sdp_add_time, sdp_add_zone, sdp_add_key, sdp_add_attribute, sdp_add_media – add specific SDP fields to the SDP session structure

Synopsis cc [*flag*...] *file*... -lcommputil [*library*...]
```
#include <sdp.h>

int sdp_add_origin(sdp_session_t *session, const char *name,
    uint64_t id, uint64_t ver, const char *nettype,
    const char *addrtype, const char *address);

int sdp_add_name(sdp_session_t *session, const char *name);

int sdp_add_information(char **information, const char *value);

int sdp_add_uri(sdp_session_t *session, const char *uri);

int sdp_add_email(sdp_session_t *session, const char *email);

int sdp_add_phone(sdp_session_t *session, const char *phone);

int sdp_add_connection(sdp_conn_t **conn, const char *nettype,
    const char *addrtype, const char *address, uint8_t ttl,
    int addrcount);

int sdp_add_bandwidth(sdp_bandwidth_t **bw, const char *type,
    uint64_t value);

int sdp_add_repeat(sdp_time_t *time, uint64_t interval,
    uint64_t duration, const char *offset);

int sdp_add_time(sdp_session_t *session, uint64_t starttime,
    uint64_t stoptime, sdp_time_t **time);

int sdp_add_zone(sdp_session_t *session, uint64_t time,
    const char *offset);

int sdp_add_key(sdp_key_t **key, const char *method,
    const char *enckey);

int sdp_add_attribute(sdp_attr_t **attr, const char *name,
    const char *value);

int sdp_add_media(sdp_session_t *session, const char *name,
    uint_t port, int portcount, const char *protocol,
    const char *format, sdp_media_t **media);
```

Description The caller has to first call sdp_new_session(3COMMPUTIL) and get pointer to a session structure. Then that pointer is used as argument in the following functions and the session structure is constructed. Once the structure is built the caller converts it to a string representation using sdp_session_to_str(3COMMPUTIL).

The `sdp_add_origin()` function adds ORIGIN (o=) SDP field to the session structure (`sdp_session_t`) using *name*, *id*, *ver*, *nettype*, *addrtype*, and *address*.

The `sdp_add_name()` function adds NAME (s=) SDP field to the session structure (`sdp_session_t`) using *name*.

The `sdp_add_information()` function adds INFO (i=) SDP field to the session structure (`sdp_session_t`) or media structure (`sdp_media_t`) using value. Since this field can be either in the media section or the session section of an SDP description the caller has to pass &*session*→s_info or &*media*→m_info as the first argument.

The `sdp_add_uri()` function adds URI (u=) SDP field to the session structure (`sdp_session_t`) using *uri*.

The `sdp_add_email()` function adds EMAIL (e=) SDP field to the session structure (`sdp_session_t`) using *email*.

The `sdp_add_phone()` function adds PHONE (p=) SDP field to the session structure (`sdp_session_t`) using *phone*.

The `sdp_add_connection()` function adds CONNECTION (c=) SDP field to the session structure (`sdp_session_t`) or the media structure (`sdp_media_t`) using *nettype*, *addrtype*, *address*, *ttl*, and *addrcount*. While adding an IP4 or IP6 unicast address the *ttl* and *addrcount* should be set to 0. For multicast address the *ttl* should be set a reasonable value (0 - 255) and *addrcount* cannot be 0. Also since this field can be either in the media section or the session section of an SDP description, the caller has to pass &*session*→s_conn or &*media*→m_conn as the first argument.

The `sdp_add_bandwidth()` function adds BANDWIDTH (b=) SDP field to the session structure (`sdp_session_t`) or the media structure (`sdp_media_t`) using *type* and *value*. Since this field can be either in the media section or the session section of an SDP description, the caller has to pass &session→s_bw or &media→m_bw as the first argument.

The `sdp_add_time()` function adds the TIME (t=) SDP field to the session structure using *startime* and *stoptime*. The pointer to the newly created time structure is returned in *time*. This pointer is then used in `sdp_add_repeat()` function.

The `sdp_add_repeat()` function adds the REPEAT (r=) SDP field to the session structure using interval, duration and offset. Here, offset is a string holding one or more offset values, for example "60" or "60 1d 3h".

The `sdp_add_zone()` function adds the ZONE (z=) SDP field to the session structure using *time* and *offset*. To add multiple time and offset values in a single zone field, call this function once for each pair. See the example below.

The `sdp_add_key()` function adds the KEY (k=) SDP field to the session structure (`sdp_session_t`) or media structure (`sdp_media_t`) using *method* and *enckey*. Since this field can be either in the media section or the session section of an SDP description, the caller has to pass &*session*→s_key or &*media*→m_key as the first argument.

The sdp_add_attribute() function adds the ATTRIBUTE (a=) SDP field to the session structure (sdp_session_t) or media structure (sdp_media_t) using *name* and *value*. Since this field can be either in the media section or the session section of an SDP description, the caller has to pass &*session*→s_attr or &*media*→m_attr as the first argument.

The sdp_add_media() function adds the MEDIA (m=) SDP field to the session structure (sdp_session_t) using *name*, *port*, *portcount*, *protocol*, and *format*. Here, *format* is a string holding possibly more than one value, for example, "0 31 32 97". The pointer to the newly created media structure is returned in *media*. This pointer is then used to add SDP fields specific to that media section.

Return Values These functions return 0 on success and the appropriate error value on failure. The value of errno is not changed by these calls in the event of an error.

Errors These functions will fail if:

EINVAL   Mandatory parameters are not provided (they are null).

ENOMEM   The allocation of memory failed.

Examples EXAMPLE 1   Build an SDP session structure

In the following example we see how to build an SDP session structure using the functions described on this manual page. We first get a pointer to sdp_session_t structure by calling sdp_new_session(). Then to this newly created structure we add various SDP fields. Once the structure is built we obtain a string representation of the structure using sdp_session_to_str() function. Since its caller responsibility to free the session we call sdp_free_session() towards the end.

```
/* SDP Message we will be building
"v=0\r\n\
o=Alice 2890844526 2890842807 IN IP4 10.47.16.5\r\n\
s=-\r\n\
i=A Seminar on the session description protocol\r\n\
u=http://www.example.com/seminars/sdp.pdf\r\n\
e=alice@example.com (Alice Smith)\r\n\
p=+1 911-345-1160\r\n\
c=IN IP4 10.47.16.5\r\n\
b=CT:1024\r\n\
t=2854678930 2854679000\r\n\
r=604800 3600 0 90000\r\n\
z=2882844526 -1h 2898848070 0h\r\n\
a=recvonly\r\n\
m=audio 49170 RTP/AVP 0\r\n\
i=audio media\r\n\
b=CT:1000\r\n\
k=prompt\r\n\
m=video 51372 RTP/AVP 99 90\r\n\
i=video media\r\n\
```

EXAMPLE 1 Build an SDP session structure   *(Continued)*

```
a=rtpmap:99 h232-199/90000\r\n\
a=rtpmap:90 h263-1998/90000\r\n"
*/

#include stdio.h>
#include string.h>
#include errno.h>
#include sdp.h>

int main ()
{
    sdp_session_t        *my_sess;
    sdp_media_t          *my_media;
    sdp_time_t           *my_time;
    char *b_sdp;

    my_sess = sdp_new_session();
    if (my_sess == NULL) {
        return (ENOMEM);
    }
    my_sess->version = 0;
    if (sdp_add_name(my_sess, "-") != 0)
        goto err_ret;
    if (sdp_add_origin(my_sess, "Alice", 2890844526ULL, 2890842807ULL,
         "IN", "IP4", "10.47.16.5") != 0)
        goto err_ret;
    if (sdp_add_information(&my_sess->s_info, "A Seminar on the session"
                 "description protocol") != 0)
        goto err_ret;
    if (sdp_add_uri (my_sess, "http://www.example.com/seminars/sdp.pdf")
                 != 0)
        goto err_ret;
    if (sdp_add_email(my_sess, "alice@example.com (Alice smith)") != 0)
        goto err_ret;
    if (sdp_add_phone(my_sess, "+1 911-345-1160") != 0)
        goto err_ret;
    if (sdp_add_connection(&my_sess->s_conn, "IN", "IP4", "10.47.16.5",
                 0, 0) != 0)
        goto err_ret;
    if (sdp_add_bandwidth(&my_sess->s_bw, "CT", 1024) != 0)
        goto err_ret;
    if (sdp_add_time(my_sess, 2854678930ULL, 2854679000ULL, &my_time)
                 != 0)
        goto err_ret;
    if (sdp_add_repeat(my_time, 604800ULL, 3600ULL, "0 90000") != 0)
```

EXAMPLE 1   Build an SDP session structure   *(Continued)*

```
        goto err_ret;
    if (sdp_add_zone(my_sess, 2882844526ULL, "-1h") != 0)
        goto err_ret;
    if (sdp_add_zone(my_sess, 2898848070ULL, "0h") != 0)
        goto err_ret;
    if (sdp_add_attribute(&my_sess->s_attr, "sendrecv", NULL) != 0)
        goto err_ret;
    if (sdp_add_media(my_sess, "audio", 49170, 1, "RTP/AVP",
                        "0", &my_media) != 0)
        goto err_ret;
    if (sdp_add_information(&my_media->m_info, "audio media") != 0)
        goto err_ret;
    if (sdp_add_bandwidth(&my_media->m_bw, "CT", 1000) != 0)
        goto err_ret;
    if (sdp_add_key(&my_media->m_key, "prompt", NULL) != 0)
        goto err_ret;
    if (sdp_add_media(my_sess, "video", 51732, 1, "RTP/AVP",
                "99 90", &my_media) != 0)
        goto err_ret;
    if (sdp_add_information(&my_media->m_info, "video media") != 0)
        goto err_ret;
    if (sdp_add_attribute(&my_media->m_attr, "rtpmap",
                "99 h232-199/90000") != 0)
        goto err_ret;
    if (sdp_add_attribute(&my_media->m_attr, "rtpmap",
                "90 h263-1998/90000") != 0)
        goto err_ret;
    b_sdp = sdp_session_to_str(my_sess, &error);

    /*
     * b_sdp is the string representation of my_sess structure
     */

    free(b_sdp);
    sdp_free_session(my_sess);
    return (0);
err_ret:
    free(b_sdp);
    sdp_free_session(my_sess);
    return (1);
}
```

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   libcommputil(3LIB), sdp_new_session(3COMMPUTIL), sdp_parse(3COMMPUTIL), sdp_session_to_str(3COMMPUTIL), attributes(5)

Name sdp_clone_session – clone an SDP session structure

Synopsis cc [*flag*...] *file*... -lcommputil [*library*...]
#include <sdp.h>

sdp_session_t *sdp_clone_session(const sdp_session_t *session);

Description The sdp_clone_session() function clones the input SDP session structure and returns the cloned structure. The resulting cloned structure has all the SDP fields from the input structure. The caller is responsible for freeing the returned cloned structure using sdp_free_session(), described on the sdp_new_session(3COMMPUTIL) manual page.

Return Values The sdp_clone_session() function returns the cloned structure on success and NULL on failure.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libcommputil(3LIB), sdp_new_session(3COMMPUTIL), attributes(5)

Name   sdp_delete_all_field, sdp_delete_all_media_field – delete all SDP fields

Synopsis   cc [*flag*...] *file*... -lcommputil [*library*...]
#include <sdp.h>

int sdp_delete_all_field(sdp_session_t *session,
    const char *field*);

int sdp_delete_all_media_field(sdp_media_t *media,
    const char *field*);

Description   The sdp_delete_all_field() function deletes all the occurrences of the specified SDP field from the session structure. For example, if the session structure has 3 bandwidth (b=) fields, then when this function is called with SDP_BANDWIDTH_FIELD, all the three bandwidth fields are deleted from the session structure.

The sdp_delete_all_media_field() function deletes all the occurrences of the specified SDP field from the specified media structure. For example, if the caller wants to delete all the attribute fields in a media structure, calling this function with SDP_ATTRIBUTE_FIELD argument would delete all the attribute fields in the media structure.

Return Values   Upon successful completion, these functions return 0. Otherwise, the appropriate error value is returned. The value of errno is not changed by these calls in the event of an error.

Errors   These functions will fail if:

EINVAL   The *session* or *media* argument is NULL or the field type is unknown.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   libcommputil(3LIB), attributes(5)

Name sdp_delete_media, sdp_delete_attribute – delete the specified media or attribute from the appropriate list

Synopsis cc [*flag*...] *file*... -lcommputil [*library*...]
#include <sdp.h>

int sdp_delete_media(sdp_media_t **l_media, sdp_media_t *media);

int sdp_delete_attribute(sdp_attr_t **l_attr, sdp_attr_t *attr);

Description The sdp_delete_media() function deletes the specified media from the media list. It is similar to deleting a node in a linked list. The function first finds the media that needs to be deleted using sdp_find_media(3COMMPUTIL). The found media is then passed to sdp_delete_media() to delete it. The function frees the memory allocated to media structure after deleting it.

The sdp_delete_attribute() function deletes the specified attribute from the attribute list. It is similar to deleting a node in a linked list. The function first finds the attribute that needs to be deleted using sdp_find_media_rtpmap(3COMMPUTIL) or sdp_find_attribute(3COMMPUTIL). The found attribute is then passed to sdp_delete_attribute() to delete it. The function frees the memory allocated to attribute structure after deleting it.

Return Values Upon successful completion, these functions return 0. Otherwise, the appropriate error value is returned. The value of errno is not changed by these calls in the event of an error.

Errors These functions will fail if:

EINVAL   The mandatory input parameters are not provided or are NULL.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libcommputil(3LIB), sdp_find_attribute(3COMMPUTIL), sdp_find_media(3COMMPUTIL), sdp_find_media_rtpmap(3COMMPUTIL), attributes(5)

Name sdp_find_attribute – find the attribute from the attribute list

Synopsis cc [*flag*...] *file*... -lcommputil [*library*...]
#include <sdp.h>

sdp_attr_t *sdp_find_attribute(sdp_attr_t **attr*, const char **name*);

Description The sdp_find_attribute() function searches the attribute list *attr* for the specified attribute *name*. If the attribute is found it returns the pointer to that attribute. Otherwise it returns NULL.

Return Values The sdp_find_attribute() function returns the attribute (sdp_attr_t *) on success and NULL when the search fails or when mandatory input parameters are NULL.

Examples EXAMPLE 1 An (incomplete) SDP description that contains one media section: audio.

```
m=audio 49170 RTP/AVP 0 8
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
a=sendonly
a=ptime:10000
a=maxptime:20000

/*
 * Assuming that above description is parsed using sdp_parse and that
 * the parsed structure is in "session" sdp_session_t structure.
 */

sdp_attr_t *ptime;
sdp_attr_t *max_ptime;
sdp_media_t *media = session->s_media;

if ((ptime = sdp_find_attribute(media->m_attr, "ptime")) == NULL)
        /* ptime attribute not present */
else if((max_ptime = sdp_find_attribute(media->m_attr,
        "maxptime")) == NULL)
        /* max_ptime attribute not present */
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libcommputil(3LIB), sdp_parse(3COMMPUTIL), attributes(5)

Name sdp_find_media – find the specified media from the media list

Synopsis cc [*flag*...] *file*... -lcommputil [*library*...]
#include <sdp.h>

sdp_media_t *sdp_find_media(sdp_media_t **media*, const char **name*);

Description The sdp_find_media() function searches the media list for the media specified by *name*. If
the media is found it returns the pointer to the media. Otherwise it returns NULL.

Return Values The sdp_find_media() function returns the media (sdp_media_t *) on success and NULL
when the search fails or the mandatory input parameters are NULL.

Examples EXAMPLE 1   An (incomplete) SDP description that contains two media sections: audio and video.

```
m=audio 49170 RTP/AVP 0 8
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
m=video 51372 RTP/AVP 31 32
a=rtpmap:31 H261/90000
a=rtpmap:32 MPV/90000

/*
 * Assuming that above description is parsed using sdp_parse() and that
 * the parsed structure is in "session" sdp_session_t structure.
 */

sdp_media_t      *my_media;
my_media = sdp_find_media(session->s_media, "video");

/*
 * my_media now points to the structure containg video media section
 * information
 */
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libcommputil(3LIB), sdp_parse(3COMMPUTIL), attributes(5)

Name sdp_find_media_rtpmap – find the rtpmap attribute in the specified media

Synopsis cc [*flag*...] *file*... -lcommputil [*library*...]
#include <sdp.h>

sdp_attr_t *sdp_find_media_rtpmap(sdp_media_t **media*,
    const char **format*);

Description The sdp_find_media_rtpmap() function searches the attribute list of the specified media structure, *media*, for the specified *format*. If the search is successful a pointer to that *rtpmap* attribute is returned. Otherwise it returns NULL.

Return Values The sdp_find_media_rtpmap() function returns the attribute (sdp_attr_t *) on success and NULL when the search fails or the mandatory input parameters are NULL.

Examples EXAMPLE 1 An (incomplete) SDP description that contains two media sections: audio and video.

```
m=audio 49170 RTP/AVP 0 8
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
m=video 51372 RTP/AVP 31 32
a=rtpmap:31 H261/90000
a=rtpmap:32 MPV/90000

/*
 * Assuming that above description is parsed using sdp_parse() and that
 * the parsed structure is in "session" sdp_session_t structure.
 */

sdp_media_t      *video;
sdp_attr_t       *mpv;

video = sdp_find_media(session->s_media, "video);
mpv = sdp_find_media_rtpmap(video, "32");

/*
 * Now the attribute structure sdp_attr_t, mpv will be having
 * values from the attribute field "a=rtpmap:32 MPV/90000"
 */
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   libcommputil(3LIB), sdp_parse(3COMMPUTIL), attributes(5)

Name sdp_new_session, sdp_free_session – allocate a new SDP session structure

Synopsis cc [*flag*...] *file*... -lcommputil [*library*...]
#include <sdp.h>

sdp_session_t *sdp_new_session();

void sdp_free_session(sdp_session_t *session);

Description The sdp_new_session() function allocates memory for an SDP session structure specified by *session*, assigns a version number to the session structure, and returns a new session structure. It is the responsibility of the user to free the memory allocated to the session structure using the sdp_free_session() function.

The sdp_free_session() function destroys the SDP session structure and frees the resources associated with it.

Return Values The sdp_new_session() function returns the newly allocated SDP session structure on success and NULL on failure.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libcommputil(3LIB), attributes(5)

Name sdp_parse – parse the SDP description

Synopsis cc [*flag*...] *file*... -lcommputil [*library*...]
#include <sdp.h>

int sdp_parse(const char *sdp_info, int *len*, int *flags*,
        sdp_session_t **session*, uint_t *p_error*);

Description The sdp_parse() function parses the SDP description present in *sdp_info* and populates the
sdp_session_t structure. The *len* argument specifies the length of the character buffer
sdp_info. The *flags* argument is not used, but must be set to 0, otherwise the call fails with the
error value of EINVAL and *session* set to NULL. The function allocates the memory required for
the sdp_session_t structure and hence the caller is responsible for freeing the parsed session
structure (sdp_session_t) using sdp_free_session(), described on the
sdp_new_session(3COMMPUTIL) manual page.

The *p_error* argument identifies any field that had a parsing error. It cannot be NULL and can
take any of the following values:

SDP_VERSION_ERROR	0x00000001
SDP_ORIGIN_ERROR	0x00000002
SDP_NAME_ERROR	0x00000004
SDP_INFO_ERROR	0x00000008
SDP_URI_ERROR	0x00000010
SDP_EMAIL_ERROR	0x00000020
SDP_PHONE_ERROR	0x00000040
SDP_CONNECTION_ERROR	0x00000080
SDP_BANDWIDTH_ERROR	0x00000100
SDP_TIME_ERROR	0x00000200
SDP_REPEAT_TIME_ERROR	0x00000400
SDP_ZONE_ERROR	0x00000800
SDP_KEY_ERROR	0x00001000
SDP_ATTRIBUTE_ERROR	0x00002000
SDP_MEDIA_ERROR	0x00004000
SDP_FIELDS_ORDER_ERROR	0x00008000
SDP_MISSING_FIELDS	0x00010000

RFC 4566 states that the fields in the SDP description need to be in a strict order. If the fields are not in the order specified in the RFC, SDP_FIELDS_ORDER_ERROR will be set.

RFC 4566 mandates certain fields to be present in SDP description. If those fields are missing then SDP_MISSING_FIELDS will be set.

Applications can check for presence of parsing error using the bit-wise operators.

If there was an error on a particular field, that field information will not be in the sdp_session_t structure. Also, parsing continues even if there was a field with a parsing error.

The sdp_session_t structure is defined in the header file <sdp.h> and contains the following members:

```
typedef    struct sdp_session {
    int             sdp_session_version; /* SDP session verstion */
    int             s_version;           /* SDP version field */
    sdp_origin_t    *s_origin;           /* SDP origin field */
    char            *s_name;             /* SDP name field */
    char            *s_info;             /* SDP info field */
    char            *s_uri;              /* SDP uri field */
    sdp_list_t      *s_email;            /* SDP email field */
    sdp_list_t      *s_phone;            /* SDP phone field */
    sdp_conn_t      *s_conn;             /* SDP connection field */
    sdp_bandwidth_t *s_bw;               /* SDP bandwidth field */
    sdp_time_t      *s_time;             /* SDP time field */
    sdp_zone_t      *s_zone;             /* SDP zone field */
    sdp_key_t       *s_key;              /* SDP key field */
    sdp_attr_t      *s_attr;             /* SDP attribute field */
    sdp_media_t     *s_media;            /* SDP media field */
} sdp_session_t;
```

The sdp_session_version member is used to track the version of the structure. Initially it is set to SDP_SESSION_VERSION_1 (= 1).

The sdp_origin_t structure contains the following members:

```
typedef struct sdp_origin {
    char        *o_username; /* username of the originating host */
    uint64_t    o_id;        /* session id */
    uint64_t    o_version;   /* version number of this session */
                             /* description */
    char        *o_nettype;  /* type of network */
    char        *o_addrtype; /* type of the address */
    char        *o_address;  /* address of the machine from which */
                             /* session was created */
} sdp_origin_t;
```

The sdp_conn_t structure contains the following members:

```
typedef struct sdp_conn {
    char            *c_nettype; /* type of network */
    char            *c_addrtype; /* type of the address */
    char            *c_address;  /* unicast-address or multicast */
                                 /* address */
    int             c_addrcount; /* number of addresses (case of */
                                 /* multicast address with layered */
                                 /* encodings */
    struct sdp_conn *c_next;     /* pointer to next connection */
                                 /* structure; there could be several */
                                 /* connection fields in SDP description */
    uint8_t         c_ttl;       /* TTL value for IPV4 multicast address */
} sdp_conn_t;
```

The sdp_bandwidth_t structure contains the following members:

```
typedef struct sdp_bandwidth {
    char                 *b_type; /* info needed to interpret b_value */
    uint64_t             b_value; /* bandwidth value */
    struct sdp_bandwidth *b_next; /* pointer to next bandwidth structure*/
                                  /* (there could be several bandwidth */
                                  /* fields in SDP description */
} sdp_bandwidth_t;
```

The sdp_list_t structure is a linked list of void pointers. This structure holds SDP fields like email and phone, in which case the void pointers point to character buffers. It to hold information in cases where the number of elements is not predefined (for example, offset (in repeat field) where void pointer holds integer values or format (in media field) where void pointers point to character buffers). The sdp_list_t structure is defined as:

```
typedef struct sdp_list {
    void            *value; /* string values in case of email, phone and */
                            /* format (in media field) or integer values */
                            /* in case of offset (in repeat field) */
    struct sdp_list *next;  /* pointer to the next node in the list */
} sdp_list_t;
```

The sdp_repeat_t structure contains the following members:

```
typedef struct sdp_repeat {
    uint64_t          r_interval; /* repeat interval, e.g. 86400 seconds */
                                  /* (1 day) */
    uint64_t          r_duration; /* duration of session, e.g. 3600 */
                                  /* seconds (1 hour) */
    sdp_list_t        *r_offset;  /* linked list of offset values; each */
                                  /* represents offset from start-time */
                                  /* in SDP time field */
    struct sdp_repeat *r_next;    /* pointer to next repeat structure; */
                                  /* there could be several repeat */
                                  /* fields in SDP description */
```

The sdp_repeat_t structure will always be part of the time structure sdp_time_t, since the repeat field does not appear alone in SDP description and is always associated with the time field.

The sdp_time_t structure contains the following members:

```
typedef struct sdp_time {
    uint64_t        t_start;  /* start-time for a session */
    uint64_t        t_stop;   /* end-time for a session */
    sdp_repeat_t    *t_repeat; /* points to the SDP repeat field */
    struct sdp_time *t_next;  /* pointer to next time field; there */
                              /* could there could be several time */
                              /* fields in SDP description */
} sdp_time_t;
```

The sdp_zone_t structure contains the following members:

```
typedef struct sdp_zone {
    uint64_t        z_time;   /* base time */
    char            *z_offset; /* offset added to z_time to determine */
                              /* session time; mainly used for daylight */
                              /* saving time conversions */
    struct sdp_zone *z_next;  /* pointer to next zone field; there */
                              /* could be several <adjustment-time> */
                              /* <offset> pairs within a zone field */
} sdp_zone_t;
```

The sdp_key_t structure contains the following members:

```
typedef struct sdp_key {
    char *k_method;  /* key type */
    char *k_enckey;  /* encryption key */
} sdp_key_t;
```

The sdp_attr_t structure contains the following members:

```
typedef struct sdp_attr {
    char            *a_name;  /* name of the attribute */
    char            *a_value; /* value of the attribute */
    struct sdp_attr *a_next;  /* pointer to the next attribute */
                              /* structure; there could be several */
                              /* attribute fields within SDP description */
} sdp_attr_t;
```

The sdp_media_t structure contains the following members:

```
typedef struct sdp_media {
    char            *m_name;     /* name of the media such as "audio", */
                                 /* "video", "message" */
    uint_t          m_port;      /* transport layer port information */
    int             m_portcount; /* number of ports in case of */
```

```
                                       /* hierarchically encoded streams */
        char              *m_proto;    /* transport protocol */
        sdp_list_t        *m_format;   /* media format description */
        char              *m_info;     /* media info field */
        sdp_conn_t        *m_conn;     /* media connection field */
        sdp_bandwidth_t   *m_bw;       /* media bandwidth field */
        sdp_key_t         *m_key;      /* media key field */
        sdp_attr_t        *m_attr;     /* media attribute field */
        struct sdp_media  *m_next;     /* pointer to next media structure; */
                                       /* there could be several media */
                                       /* sections in SDP description */
        sdp_session_t     *m_session;  /* pointer to the session structure */
} sdp_media_t;
```

Return Values The sdp_parse() function returns 0 on success and the appropriate error value on failure. The value of errno is not changed by these calls in the event of an error.

Errors The sdp_parse() function will fail if:

EINVAL   Arguments to the function were invalid.

ENOMEM   Memory allocation failed while parsing *sdp_info*.

Examples EXAMPLE 1   sdp_parse() example

If the SDP description was

```
v=0\r\n
o=jdoe 23423423 234234234 IN IP4 192.168.1.1\r\n
s=SDP seminar\r\n
i=A seminar on the session description protocol\r\n
e=test@host.com
c=IN IP4 156.78.90.1\r\n
t=2873397496 2873404696\r\n
```

then after call to sdp_parse() function the sdp_session_t structure would be

```
session {
        sdp_session_version = 1
        s_version = 0
        s_origin {
                o_username = "jdoe"
                o_id = 23423423ULL
                o_version = 234234234ULL
                o_nettype = "IN"
                o_addrtype = "IP4"
                o_address = "192.168.1.1"
        }
        s_name = "SDP seminar"
        s_info = "A seminar on the session description protocol"
```

EXAMPLE 1   sdp_parse() example   *(Continued)*

```
        s_uri =  (nil)
        s_email {
                value = "test@host.com"
                next = (nil)
        }
        s_phone = (nil)
        s_conn {
                c_nettype = "IN"
                c_addrtype = "IP4"
                c_address = "156.78.90.1"
                c_addrcount = 0
                c_ttl = 0
                c_next = (nil)
        }
        s_bw = (nil)
        s_time {
                t_start = 2873397496ULL
                t_stop = 2873404696ULL
                t_repeat = (nil)
                t_next = (nil)
        }
        s_zone = (nil)
        s_key = (nil)
        s_attr = (nil)
        s_media = (nil)
}
```

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   libcommputil(3LIB), sdp_new_session(3COMMPUTIL), attributes(5)

Name sdp_session_to_str – return a string representation of a session structure

Synopsis cc [*flag*...] *file*... -lcommputil [*library*...]
#include <sdp.h>

char *sdp_session_to_str(const sdp_session_t *session,
      int *error);

Description The sdp_session_to_str() function returns the string representation of the SDP session
structure *session*. The caller is responsible for freeing the returned string.

The function adds a CRLF at the end of each SDP field before appending that field to the
string.

Return Values The sdp_session_to_str() function returns the relevant string on success and NULL
otherwise.

If *error* is non-null, the location pointed by *error* is set to 0 on success or the error value on
failure. The value of errno is not changed by these calls in the event of an error.

Errors The sdp_session_to_str() function will fail if:

EINVAL   The input is null.

ENOMEM   A memory allocation failure occurred.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libcommputil(3LIB), attributes(5)

Name secure_rpc, authdes_getucred, authdes_seccreate, getnetname, host2netname, key_decryptsession, key_encryptsession, key_gendes, key_setsecret, key_secretkey_is_set, netname2host, netname2user, user2netname – library routines for secure remote procedure calls

Synopsis cc [*flag*...] *file*... -lnsl [*library*...]
```
#include <rpc/rpc.h>
#include <sys/types.h>
```

```
int authdes_getucred(const struct authdes_cred *adc, uid_t *uidp,
     gid_t *gidp, short *gidlenp, gid_t *gidlist);
```

```
AUTH *authdes_seccreate(const char *name, const uint_t window,
     const char *timehost, ckey);
```

```
int getnetname(char name [MAXNETNAMELEN+1]);
```

```
int host2netname(char name [MAXNETNAMELEN+1], const char *host,
     const char *domain);
```

```
int key_decryptsession(const char *remotename, des_block *deskey);
```

```
int key_encryptsession(const char *remotename, des_block *deskey);
```

```
int key_gendes(des_block *deskey);
```

```
int key_setsecret(const char *key);
```

```
int key_secretkey_is_set(void)
```

```
int netname2host(const char *name, char *host, const int hostlen);
```

```
int netname2user(const char *name, uid_t *uidp, gid_t *gidp, int *gidlenp,
     gid_t *gidlist [NGRPS]);
```

```
int user2netname(char name [MAXNETNAMELEN+1], const uid_t uid,
     const char *domain);
```

Description The RPC library functions allow C programs to make procedure calls on other machines across the network.

RPC supports various authentication flavors. Among them are:

AUTH_NONE   No authentication (none).

AUTH_SYS   Traditional UNIX-style authentication.

AUTH_DES   DES encryption-based authentication.

The authdes_getucred() and authdes_seccreate() functions implement the AUTH_DES authentication style. The keyserver daemon keyserv(1M) must be running for the AUTH_DES authentication system to work and keylogin(1) must have been run. The AUTH_DES style of

authentication is discussed here. For information about the AUTH_NONE and AUTH_SYS flavors of authentication, refer to rpc_clnt_auth(3NSL). See rpc(3NSL) for the definition of the AUTH data structure.

The following functions documented on this page are MT-Safe. For the MT-levels of other authentication styles, see relevant man pages.

authdes_getucred()   This is the first of two functions that interface to the RPC secure authentication system AUTH_DES. The second is the authdes_seccreate() function. The authdes_getucred() function is used on the server side to convert an AUTH_DES credential, which is operating system independent, to an AUTH_SYS credential. The authdes_getucred() function returns 1 if it succeeds, 0 if it fails.

The *uidp* parameter is set to the user's numerical ID associated with *adc*. The *gidp* parameter is set to the numerical ID of the user's group. The *gidlist* parameter contains the numerical IDs of the other groups to which the user belongs. The *gidlenp* parameter is set to the number of valid group ID entries specified by the *gidlist* parameter.

The authdes_getucred() function fails if the authdes_cred structure was created with the netname of a host. In such a case, netname2host() should be used to get the host name from the host netname in the authdes_cred structure.

authdes_seccreate()   The second of two AUTH_DES authentication functions, the authdes_seccreate() function is used on the client side to return an authentication handle that enables the use of the secure authentication system. The first field, *name*, specifies the network name *netname* of the owner of the server process. The field usually represents a hostname derived from the host2netname() utility, but the field might also represent a user name converted with the user2netname() utility.

The second field, *window*, specifies the validity of the client credential in seconds. If the difference in time between the client's clock and the server's clock exceeds *window*, the server rejects the client's credentials and the clock will have to be resynchronized. A small window is more secure than a large one, but choosing too small a window increases the frequency of resynchronization due to clock drift.

The third parameter, *timehost*, is the host's name and is optional. If *timehost* is NULL, the authentication system

assumes that the local clock is always in sync with the *timehost* clock and does not attempt resynchronization. If a timehost is supplied, the system consults the remote time service whenever resynchronization is required. The *timehost* parameter is usually the name of the host on which the server is running.

The final parameter, *ckey*, is also optional. If *ckey* is NULL, the authentication system generates a random DES key to be used for the encryption of credentials. If *ckey* is supplied, it is used for encryption.

If authdes_seccreate() fails, it returns NULL.

getnetname()   This function returns the unique, operating system independent netname of the caller in the fixed-length array *name*. The function returns 1 if it succeeds and 0 if it fails.

host2netname()   This function converts a domain-specific hostname *host* to an operating system independent netname. The function returns 1 if it succeeds and 0 if it fails. The host2netname() function is the inverse of the netname2host() function. If the *domain* is NULL, host2netname() uses the default domain name of the machine. If *host* is NULL, it defaults to that machine itself. If *domain* is NULL and *host* is an NIS name such as myhost.sun.example.com, the host2netname() function uses the domain sun.example.com rather than the default domain name of the machine.

key_decryptsession()   This function is an interface to the keyserver daemon, which is associated with RPC's secure authentication system (AUTH_DES authentication). User programs rarely need to call key_decryptsession() or the associated functions key_encryptsession(), key_gendes(), and key_setsecret().

The key_decryptsession() function takes a server netname *remotename* and a DES key *deskey*, and decrypts the key by using the the public key of the server and the secret key associated with the effective UID of the calling process. The key_decryptsession() function is the inverse of key_encryptsession() function.

key_encryptsession()   This function is a keyserver interface that takes a server netname *remotename* and a DES key *deskey*, and encrypts the key using the public key of the the server and the secret key

associated with the effective UID of the calling process. If the keyserver does not have a key registered for the UID, it falls back to using the secret key for the netname nobody unless this feature has been disabled. See keyserv(1M). The key_encryptsession() function is the inverse of key_decryptsession() function. The key_encryptsession() function returns 0 if it succeeds, −1 if it fails.

key_gendes()

This is a keyserver interface function used to ask the keyserver for a secure conversation key. Selecting a conversion key at random is generally not secure because the common ways of choosing random numbers are too easy to guess. The key_gendes() function returns 0 if it succeeds, −1 if it fails.

key_setsecret()

This is a keyserver interface function used to set the key for the effective UID of the calling process. This function returns 0 if it succeeds, −1 if it fails.

key_secretkey_is_set()

This is a keyserver interface function used to determine if a key has been set for the effective UID of the calling process. If the keyserver has a key stored for the effective UID of the calling process, the key_secretkey_is_set() function returns 1. Otherwise it returns 0.

netname2host()

This function converts an operating system independent netname *name* to a domain-specific hostname *host*. The *hostlen* parameter is the maximum size of *host*. The netname2host() function returns 1 if it succeeds and 0 if it fails. The function is the inverse of the host2netname() function.

netname2user()

This function converts an operating system independent netname to a domain-specific user ID. The netname2user() function returns 1 if it succeeds and 0 if it fails. The function is the inverse of the user2netname() function.

The *uidp* parameter is set to the user's numerical ID associated with *name*. The *gidp* parameter is set to the numerical ID of the user's group. The *gidlist* parameter contains the numerical IDs of the other groups to which the user belongs. The *gidlenp* parameter is set to the number of valid group ID entries specified by the *gidlist* parameter.

user2netname()

This function converts a domain-specific username to an operating system independent netname. The user2netname() function returns 1 if it succeeds and 0 if it fails. The function is

the inverse of `netname2user()` function.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also   chkey(1), keylogin(1), keyserv(1M), newkey(1M), rpc(3NSL), rpc_clnt_auth(3NSL), attributes(5)

Name send, sendto, sendmsg – send a message from a socket

Synopsis cc [*flag*...] *file*... -lsocket -lnsl [*library*...]
#include <sys/socket.h>

ssize_t send(int *s*, const void **msg*, size_t *len*, int *flags*);

ssize_t sendto(int *s*, const void **msg*, size_t *len*, int *flags*,
    const struct sockaddr **to*, socklen_t *tolen*);

ssize_t sendmsg(int *s*, const struct msghdr **msg*, int *flags*);

Description The send(), sendto(), and sendmsg() functions are used to transmit a message to another
transport end-point. The send() function can be used only when the socket is in a connected
state. See connect(3SOCKET). The sendto() and sendmsg() functions can be used at any
time. The *s* socket is created with socket(3SOCKET).

The address of the target is supplied by *to* with a *tolen* parameter used to specify the size. The
length of the message is supplied by the *len* parameter. For socket types such as SOCK_DGRAM
and SOCK_RAW that require atomic messages, the error EMSGSIZE is returned and the message is
not transmitted when it is too long to pass atomically through the underlying protocol. The
same restrictions do not apply to SOCK_STREAM sockets.

A return value −1 indicates locally detected errors. It does not imply a delivery failure.

If the socket does not have enough buffer space available to hold a message, the send()
function blocks the message, unless the socket has been placed in non-blocking I/O mode (see
fcntl(2)). The select(3C) or poll(2) call can be used to determine when it is possible to send
more data.

The *flags* parameter is formed from the bitwise OR of zero or more of the following:

MSG_OOB   Send *out-of-band* data on sockets that support this notion. The
                  underlying protocol must also support *out-of-band* data. Only
                  SOCK_STREAM sockets created in the AF_INET or the AF_INET6 address
                  family support out-of-band data.

MSG_DONTROUTE   The SO_DONTROUTE option is turned on for the duration of the operation.
                  It is used only by diagnostic or routing programs.

See recv(3SOCKET) for a description of the msghdr structure.

Return Values Upon successful completion, these functions return the number of bytes sent. Otherwise, they
return -1 and set errno to indicate the error.

Errors The send(), sendto(), and sendmsg() functions return errors under the following
conditions:

EBADF   *s* is not a valid file descriptor.

EINTR	The operation was interrupted by delivery of a signal before any data could be buffered to be sent.
EMSGSIZE	The message is too large to be sent all at once (as the socket requires), or the *msg_iovlen* member of the msghdr structure pointed to by message is less than or equal to 0 or is greater than {IOV_MAX}.
ENOMEM	Insufficient memory is available to complete the operation.
ENOSR	Insufficient STREAMS resources are available for the operation to complete.
ENOTSOCK	*s* is not a socket.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block. EWOULDBLOCK is also returned when sufficient memory is not immediately available to allocate a suitable buffer. In such a case, the operation can be retried later.
ECONNREFUSED	The requested connection was refused by the peer. For conected IPv4 and IPv6 datagram sockets, this indicates that the system received an ICMP Destination Port Unreachable message from the peer in response to some prior transmission.

The send() and sendto() functions return errors under the following conditions:

EINVAL   The *len* argument overflows a ssize_t.

Inconsistent port attributes for system call.

The sendto() function returns errors under the following conditions:

EINVAL   The value specified for the *tolen* parameter is not the size of a valid address for the specified address family.

EISCON   A destination address was specified and the socket is already connected.

The sendmsg() function returns errors under the following conditions:

EINVAL   The msg_iovlen member of the msghdr structure pointed to by *msg* is less than or equal to 0, or the sum of the *iov_len* values in the msg_iov array overflows a ssize_t.

One of the *iov_len* values in the msg_iov array member of the msghdr structure pointed to by *msg* is negative, or the sum of the *iov_len* values in the msg_iov array overflows a ssize_t.

msg_iov contents are inconsistent with port attributes.

The send() function returns errors under the following conditions:

EPIPE   The socket is shut down for writing, or the socket is connection–mode and is no longer connected. In the latter case, if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling thread.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   fcntl(2), poll(2), write(2), connect(3SOCKET), getsockopt(3SOCKET), recv(3SOCKET), select(3C), socket(3SOCKET), socket.h(3HEAD), attributes(5)

Name send – send a message on a socket

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <sys/socket.h>

ssize_t send(int *socket*, const void **buffer*, size_t *length*, int flags);

Parameters *socket*   Specifies the socket file descriptor.

buffer   Points to the buffer containing the message to send.

length   Specifies the length of the message in bytes.

flags   Specifies the type of message transmission. Values of this argument are formed by
logically OR'ing zero or more of the following flags:

MSG_EOR   Terminates a record (if supported by the protocol)

MSG_OOB   Sends out-of-band data on sockets that support out-of-band
communications. The significance and semantics of out-of-band data
are protocol-specific.

Description The send() function initiates transmission of a message from the specified socket to its peer.
The send() function sends a message only when the socket is connected (including when the
peer of a connectionless socket has been set via connect(3XNET)).

The length of the message to be sent is specified by the *length* argument. If the message is too
long to pass through the underlying protocol, send() fails and no data is transmitted.

Successful completion of a call to send() does not guarantee delivery of the message. A return
value of −1 indicates only locally-detected errors.

If space is not available at the sending socket to hold the message to be transmitted and the
socket file descriptor does not have O_NONBLOCK set, send() blocks until space is available. If
space is not available at the sending socket to hold the message to be transmitted and the
socket file descriptor does have O_NONBLOCK set, send() will fail. The select(3C) and poll(2)
functions can be used to determine when it is possible to send more data.

The socket in use may require the process to have appropriate privileges to use the send()
function.

Usage The send() function is identical to sendto(3XNET) with a null pointer *dest_len* argument,
and to write() if no flags are used.

Return Values Upon successful completion, send() returns the number of bytes sent. Otherwise, −1 is
returned and errno is set to indicate the error.

Errors The send() function will fail if:

EAGAIN	
EWOULDBLOCK	The socket's file descriptor is marked O_NONBLOCK and the requested operation would block.
EBADF	The *socket* argument is not a valid file descriptor.
ECONNRESET	A connection was forcibly closed by a peer.
EDESTADDRREQ	The socket is not connection-mode and no peer address is set.
EFAULT	The *buffer* parameter can not be accessed.
EINTR	A signal interrupted send() before any data was transmitted.
EMSGSIZE	The message is too large be sent all at once, as the socket requires.
ENOTCONN	The socket is not connected or otherwise has not had the peer prespecified.
ENOTSOCK	The *socket* argument does not refer to a socket.
EOPNOTSUPP	The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.
EPIPE	The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling thread.

The send() function may fail if:

EACCES	The calling process does not have the appropriate privileges.
EIO	An I/O error occurred while reading from or writing to the file system.
ENETDOWN	The local interface used to reach the destination is down.
ENETUNREACH	No route to the network is present.
ENOBUFS	Insufficient resources were available in the system to perform the operation.
ENOSR	There were insufficient STREAMS resources available for the operation to complete.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also connect(3XNET), getsockopt(3XNET), poll(2), recv(3XNET), recvfrom(3XNET), recvmsg(3XNET), select(3C), sendmsg(3XNET), sendto(3XNET), setsockopt(3XNET), shutdown(3XNET), socket(3XNET), attributes(5), standards(5)

Name   sendmsg – send a message on a socket using a message structure

Synopsis   cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <sys/socket.h>

ssize_t sendmsg(int *socket*, const struct msghdr **message*, int *flags*);

Parameters   The function takes the following arguments:

socket   Specifies the socket file descriptor.

message   Points to a msghdr structure, containing both the destination address and the
                buffers for the outgoing message. The length and format of the address depend on
                the address family of the socket. The msg_flags member is ignored.

flags   Specifies the type of message transmission. The application may specify 0 or the
                following flag:

                MSG_EOR   Terminates a record (if supported by the protocol)

                MSG_OOB   Sends out-of-band data on sockets that support out-of-bound
                            data. The significance and semantics of out-of-band data are
                            protocol-specific.

Description   The sendmsg() function sends a message through a connection-mode or
connectionless-mode socket. If the socket is connectionless-mode, the message will be sent to
the address specified by *msghdr*. If the socket is connection-mode, the destination address in
msghdr is ignored.

The *msg_iov* and *msg_iovlen* fields of message specify zero or more buffers containing the data
to be sent. *msg_iov* points to an array of iovec structures; *msg_iovlen* must be set to the
dimension of this array. In each iovec structure, the *iov_base* field specifies a storage area and
the *iov_len* field gives its size in bytes. Some of these sizes can be zero. The data from each
storage area indicated by *msg_iov* is sent in turn.

Successful completion of a call to sendmsg() does not guarantee delivery of the message. A
return value of −1 indicates only locally-detected errors.

If space is not available at the sending socket to hold the message to be transmitted and the
socket file descriptor does not have O_NONBLOCK set, sendmsg() function blocks until space is
available. If space is not available at the sending socket to hold the message to be transmitted
and the socket file descriptor does have O_NONBLOCK set, sendmsg() function will fail.

If the socket protocol supports broadcast and the specified address is a broadcast address for
the socket protocol, sendmsg() will fail if the SO_BROADCAST option is not set for the socket.

The socket in use may require the process to have appropriate privileges to use the sendmsg()
function.

Usage The select(3C) and poll(2) functions can be used to determine when it is possible to send more data.

Return Values Upon successful completion, sendmsg() function returns the number of bytes sent. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The sendmsg() function will fail if:

EAGAIN
EWOULDBLOCK   The socket's file descriptor is marked O_NONBLOCK and the requested operation would block.

EAFNOSUPPORT   Addresses in the specified address family cannot be used with this socket.

EBADF   The *socket* argument is not a valid file descriptor.

ECONNRESET   A connection was forcibly closed by a peer.

EFAULT   The *message* parameter, or storage pointed to by the *msg_name*, *msg_control* or *msg_iov* fields of the *message* parameter, or storage pointed to by the iovec structures pointed to by the *msg_iov* field can not be accessed.

EINTR   A signal interrupted sendmsg() before any data was transmitted.

EINVAL   The sum of the iov_len values overflows an ssize_t.

EMSGSIZE   The message is to large to be sent all at once (as the socket requires), or the msg_iovlen member of the msghdr structure pointed to by *message* is less than or equal to 0 or is greater than IOV_MAX.

ENOTCONN   The socket is connection-mode but is not connected.

ENOTSOCK   The *socket* argument does not refer a socket.

EOPNOTSUPP   The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.

EPIPE   The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling thread.

If the address family of the socket is AF_UNIX, then sendmsg() will fail if:

EIO   An I/O error occurred while reading from or writing to the file system.

ELOOP   Too many symbolic links were encountered in translating the pathname in the socket address.

ENAMETOOLONG   A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX characters.

| ENOENT | A component of the pathname does not name an existing file or the pathname is an empty string. |
| ENOTDIR | A component of the path prefix of the pathname in the socket address is not a directory. |

The sendmsg() function may fail if:

EACCES	Search permission is denied for a component of the path prefix; or write access to the named socket is denied.
EDESTADDRREQ	The socket is not connection-mode and does not have its peer address set, and no destination address was specified.
EHOSTUNREACH	The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).
EIO	An I/O error occurred while reading from or writing to the file system.
EISCONN	A destination address was specified and the socket is already connected.
ENETDOWN	The local interface used to reach the destination is down.
ENETUNREACH	No route to the network is present.
ENOBUFS	Insufficient resources were available in the system to perform the operation.
ENOMEM	Insufficient memory was available to fulfill the request.
ENOSR	There were insufficient STREAMS resources available for the operation to complete.

If the address family of the socket is AF_UNIX, then sendmsg() may fail if:

| ENAMETOOLONG | Pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX. |

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also poll(2) getsockopt(3XNET), recv(3XNET), recvfrom(3XNET), recvmsg(3XNET), select(3C), send(3XNET), sendto(3XNET), setsockopt(3XNET), shutdown(3XNET), socket(3XNET), attributes(5), standards(5)

Name sendto – send a message on a socket

Synopsis cc [*flag*...] *file*... -lxnet [*library* ...]
#include <sys/socket.h>

ssize_t sendto(int *socket*, const void **message*, size_t *length*, int *flags*,
    const struct sockaddr **dest_addr*, socklen_t *dest_len*);

Description The sendto() function sends a message through a connection-mode or connectionless-mode
socket. If the socket is connectionless-mode, the message will be sent to the address specified
by *dest_addr*. If the socket is connection-mode, *dest_addr* is ignored.

If the socket protocol supports broadcast and the specified address is a broadcast address for
the socket protocol, sendto() will fail if the SO_BROADCAST option is not set for the socket.

The *dest_addr* argument specifies the address of the target. The *length* argument specifies the
length of the message.

Successful completion of a call to sendto() does not guarantee delivery of the message. A
return value of −1 indicates only locally-detected errors.

If space is not available at the sending socket to hold the message to be transmitted and the
socket file descriptor does not have O_NONBLOCK set, sendto() blocks until space is available. If
space is not available at the sending socket to hold the message to be transmitted and the
socket file descriptor does have O_NONBLOCK set, sendto() will fail.

The socket in use may require the process to have appropriate privileges to use the sendto()
function.

Parameters The function takes the following arguments:

socket   Specifies the socket file descriptor.

message   Points to a buffer containing the message to be sent.

length   Specifies the size of the message in bytes.

flags   Specifies the type of message transmission. Values of this argument are formed
              by logically OR'ing zero or more of the following flags:

              MSG_EOR   Terminates a record (if supported by the protocol)

              MSG_OOB   Sends out-of-band data on sockets that support out-of-band
                         data. The significance and semantics of out-of-band data are
                         protocol-specific.

dest_addr   Points to a sockaddr structure containing the destination address. The length
              and format of the address depend on the address family of the socket.

dest_len   Specifies the length of the sockaddr structure pointed to by the *dest_addr*
              argument.

Usage The select(3C) and poll(2) functions can be used to determine when it is possible to send more data.

Return Values Upon successful completion, sendto() returns the number of bytes sent. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The sendto() function will fail if:

EAFNOSUPPORT	Addresses in the specified address family cannot be used with this socket.
EAGAIN	
EWOULDBLOCK	The socket's file descriptor is marked O_NONBLOCK and the requested operation would block.
EBADF	The *socket* argument is not a valid file descriptor.
ECONNRESET	A connection was forcibly closed by a peer.
EFAULT	The *message* or *destaddr* parameter cannot be accessed.
EINTR	A signal interrupted sendto() before any data was transmitted.
EMSGSIZE	The message is too large to be sent all at once, as the socket requires.
ENOTCONN	The socket is connection-mode but is not connected.
ENOTSOCK	The *socket* argument does not refer to a socket.
EOPNOTSUPP	The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.
EPIPE	The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling thread.

If the address family of the socket is AF_UNIX, then sendto() will fail if:

EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating the pathname in the socket address.
ENAMETOOLONG	A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX characters.
ENOENT	A component of the pathname does not name an existing file or the pathname is an empty string.
ENOTDIR	A component of the path prefix of the pathname in the socket address is not a directory.

The sendto() function may fail if:

EACCES	Search permission is denied for a component of the path prefix; or write access to the named socket is denied.
EDESTADDRREQ	The socket is not connection-mode and does not have its peer address set, and no destination address was specified.
EHOSTUNREACH	The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).
EINVAL	The *dest_len* argument is not a valid length for the address family.
EIO	An I/O error occurred while reading from or writing to the file system.
EISCONN	A destination address was specified and the socket is already connected.
ENETDOWN	The local interface used to reach the destination is down.
ENETUNREACH	No route to the network is present.
ENOBUFS	Insufficient resources were available in the system to perform the operation.
ENOMEM	Insufficient memory was available to fulfill the request.
ENOSR	There were insufficient STREAMS resources available for the operation to complete.

If the address family of the socket is AF_UNIX, then sendto() may fail if:

ENAMETOOLONG	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also   poll(2), getsockopt(3XNET), recv(3XNET), recvfrom(3XNET), recvmsg(3XNET), select(3C), send(3XNET), sendmsg(3XNET), setsockopt(3XNET), shutdown(3XNET), socket(3XNET), attributes(5), standards(5)

Name setsockopt – set the socket options

Synopsis cc [*flag*...] *file*... -lxnet [*library*...]
#include <sys/socket.h>

int setsockopt(int *socket*, int *level*, int *option_name*,
    const void*option_value, socklen_t *option_len*);

Description The setsockopt() function sets the option specified by the *option_name* argument, at the
protocol level specified by the *level* argument, to the value pointed to by the *option_value*
argument for the socket associated with the file descriptor specified by the *socket* argument.

The *level* argument specifies the protocol level at which the option resides. To set options at
the socket level, specify the *level* argument as SOL_SOCKET. To set options at other levels,
supply the appropriate protocol number for the protocol controlling the option. For example,
to indicate that an option will be interpreted by the TCP (Transport Control Protocol), set
level to the protocol number of TCP, as defined in the<netinet/in.h> header, or as
determined by using getprotobyname(3XNET).

The *option_name* argument specifies a single option to set. The *option_name* argument and
any specified options are passed uninterpreted to the appropriate protocol module for
interpretations. The <sys/socket.h> header defines the socket level options. The options are
as follow

SO_DEBUG   Turns on recording of debugging information. This option enables or
                  disables debugging in the underlying protocol modules. This option takes
                  an int value. This is a boolean option.

SO_BROADCAST   Permits sending of broadcast messages, if this is supported by the
                  protocol. This option takes an int value. This is a boolean option.

SO_REUSEADDR   Specifies that the rules used in validating addresses supplied to
                  bind(3XNET) should allow reuse of local addresses, if this is supported by
                  the protocol. This option takes an int value. This is a boolean option.

SO_KEEPALIVE   Keeps connections active by enabling the periodic transmission of
                  messages, if this is supported by the protocol. This option takes an int
                  value.

                  If the connected socket fails to respond to these messages, the connection
                  is broken and threads writing to that socket are notified with a SIGPIPE
                  signal.

                  This is a boolean option.

SO_LINGER   Lingers on a close(2) if data is present. This option controls the action
                  taken when unsent messages queue on a socket and close(2) is
                  performed. If SO_LINGER is set, the system blocks the process during
                  close(2) until it can transmit the data or until the time expires. If

SO_LINGER is not specified, and close(2) is issued, the system handles the call in a way that allows the process to continue as quickly as possible. This option takes a linger structure, as defined in the <sys/socket.h> header, to specify the state of the option and linger interval.

SO_OOBINLINE   Leaves received out-of-band data (data marked urgent) in line. This option takes an int value. This is a boolean option.

SO_SNDBUF   Sets send buffer size. This option takes an int value.

SO_RCVBUF   Sets receive buffer size. This option takes an int value.

SO_DONTROUTE   Requests that outgoing messages bypass the standard routing facilities. The destination must be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. The effect, if any, of this option depends on what protocol is in use. This option takes an int value. This is a boolean option.

SO_MAC_EXEMPT   Sets the mandatory access control on the socket. A socket that has this option enabled can communicate with an unlabeled peer if the socket is in the global zone or has a label that dominates the default label of the peer. Otherwise, the socket must have a label that is equal to the default label of the unlabeled peer. SO_MAC_EXEMPT is a boolean option that is available only when the system is configured with Trusted Extensions.

SO_ALLZONES   Bypasses zone boundaries (privileged). This option stores an int value. This is a boolean option.

The SO_ALLZONES option can be used to bypass zone boundaries between shared-IP zones. Normally, the system prevents a socket from being bound to an address that is not assigned to the current zone. It also prevents a socket that is bound to a wildcard address from receiving traffic for other zones. However, some daemons which run in the global zone might need to send and receive traffic using addresses that belong to other shared-IP zones. If set before a socket is bound, SO_ALLZONES causes the socket to ignore zone boundaries between shared-IP zones and permits the socket to be bound to any address assigned to the shared-IP zones. If the socket is bound to a wildcard address, it receives traffic intended for all shared-IP zones and behaves as if an equivalent socket were bound in each active shared-IP zone. Applications that use the SO_ALLZONES option to initiate connections or send datagram traffic should specify the source address for outbound traffic by binding to a specific address. There is no effect from setting this option in an exclusive-IP zone. Setting this option requires the sys_net_config privilege. See zones(5).

For boolean options, 0 indicates that the option is disabled and 1 indicates that the option is enabled.

Options at other protocol levels vary in format and name.

Usage The setsockopt() function provides an application program with the means to control socket behavior. An application program can use setsockopt() to allocate buffer space, control timeouts, or permit socket data broadcasts. The <sys/socket.h> header defines the socket-level options available to setsockopt().

Options may exist at multiple protocol levels. The SO_ options are always present at the uppermost socket level.

Return Values Upon successful completion, setsockopt() returns 0. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The setsockopt() function will fail if:

EBADF	The *socket* argument is not a valid file descriptor.
EDOM	The send and receive timeout values are too big to fit into the timeout fields in the socket structure.
EFAULT	The *option_value* parameter can not be accessed or written.
EINVAL	The specified option is invalid at the specified socket level or the socket has been shut down.
EISCONN	The socket is already connected, and a specified option can not be set while the socket is connected.
ENOPROTOOPT	The option is not supported by the protocol.
ENOTSOCK	The *socket* argument does not refer to a socket.

The setsockopt() function may fail if:

ENOMEM	There was insufficient memory available for the operation to complete.
ENOBUFS	Insufficient resources are available in the system to complete the call.
ENOSR	There were insufficient STREAMS resources available for the operation to complete.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe
Standard	See standards(5).

See Also bind(3XNET), endprotoent(3XNET), getsockopt(3XNET), socket(3XNET), attributes(5), standards(5)

Name   shutdown – shut down part of a full-duplex connection

Synopsis   cc [*flag*...] *file*... -lsocket -lnsl [*library*...]
#include <sys/socket.h>

int shutdown(int *s*, int *how*);

Description   The shutdown() call shuts down all or part of a full-duplex connection on the socket
associated with *s*. If *how* is SHUT_RD, further receives are disallowed. If *how* is SHUT_WR, further
sends are disallowed. If *how* is SHUT_RDWR, further sends and receives are disallowed.

The *how* values should be defined constants.

Return Values   0 is returned if the call succeeds.

−1 is returned if the call fails.

Errors   The call succeeds unless one of the following conditions exists:

EBADF   The *s* value is not a valid file descriptor.

ENOMEM   Insufficient user memory is available for the operation to complete.

ENOSR   Insufficient STREAMS resources are available for the operation to complete.

ENOTCONN   The specified socket is not connected.

ENOTSOCK   The *s* value is not a socket.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   connect(3SOCKET), socket(3SOCKET), socket.h(3HEAD), attributes(5)

Name shutdown – shut down socket send and receive operations

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <sys/socket.h>

int shutdown(int *socket*, int *how*);

Description The shutdown() function disables subsequent send() and receive() operations on a socket, depending on the value of the *how* argument.

Parameters *how* Specifies the type of shutdown. The values are as follows:

SHUT_RD Disables further receive operations.

SHUT_WR Disables further send operations.

SHUT_RDWR Disables further send and receive operations.

socket Specifies the file descriptor of the socket.

Return Values Upon successful completion, shutdown() returns 0. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The shutdown() function will fail if:

EBADF The *socket* argument is not a valid file descriptor.

EINVAL The *how* argument is invalid.

ENOTCONN The socket is not connected.

ENOTSOCK The *socket* argument does not refer to a socket.

The shutdown() function may fail if:

ENOBUFS Insufficient resources were available in the system to perform the operation.

ENOSR There were insufficient STREAMS resources available for the operation to complete.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also   getsockopt(3XNET), recv(3XNET), recvfrom(3XNET), recvmsg(3XNET), select(3C), send(3XNET), sendto(3XNET), setsockopt(3XNET), socket(3XNET), attributes(5), standards(5)

Name sip_add_branchid_to_via – add a branch parameter to the topmost VIA header in the SIP message

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

int *sip_add_branchid_to_via*(sip_msg_t *sip_msg*, char **branchid*);

Description The sip_add_branchid_to_via() function adds a branch *param* to the topmost VIA header in the SIP message *sip_msg*. Note that a new header is created as a result of adding the branch parameter and the old header is marked deleted. Applications with multiple threads working on the same VIA header need to take note of this.

Return Values These functions return 0 on success and the appropriate error value on failure.

Errors On failure, functions that return an error value may return one of the following:

EINVAL   Mandatory parameters are not provided or are NULL.

For sip_add_branchid_to_via(), the topmost VIA header already has a branch *param* or the SIP message does not have a VIA header.

EPERM   The message cannot be modified.

ENOMEM   There is an error allocating memory for creating headers/parameters.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_add_from, sip_add_to, sip_add_contact, sip_add_via, sip_add_maxforward,
sip_add_callid, sip_add_cseq, sip_add_content_type, sip_add_content, sip_add_accept,
sip_add_accept_enc, sip_add_accept_lang, sip_add_alert_info, sip_add_allow,
sip_add_call_info, sip_add_content_disp, sip_add_content_enc, sip_add_content_lang,
sip_add_date, sip_add_error_info, sip_add_expires, sip_add_in_reply_to,
sip_add_mime_version, sip_add_min_expires, sip_add_org, sip_add_priority,
sip_add_reply_to, sip_add_passertedid, sip_add_ppreferredid, sip_add_require,
sip_add_retry_after, sip_add_route, sip_add_record_route, sip_add_server, sip_add_subject,
sip_add_supported, sip_add_tstamp, sip_add_unsupported, sip_add_user_agent,
sip_add_warning, sip_add_rseq, sip_add_privacy, sip_add_rack, sip_add_author,
sip_add_authen_info, sip_add_proxy_authen, sip_add_proxy_author,
sip_add_proxy_require, sip_add_www_authen, sip_add_allow_events, sip_add_event,
sip_add_substate – add specific SIP headers to the SIP message

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

int *sip_add_from*(sip_msg_t *sip_msg*, char **display_name*, char **from_uri*,
    char **from_tag*, boolean_t *add_aquot*, char **from_params*);

int *sip_add_to*(sip_msg_t *sip_msg*, char **display_name*, char **to_uri*,
    char **to_tag*, boolean_t *add_aquot*, char **to_params*);

int *sip_add_contact*(sip_msg_t *sip_msg*, char **display_name*,
    char **contact_uri*, boolean_t *add_aquot*, char **contact_params*);

int *sip_add_via*(sip_msg_t *sip_msg*, char **sent_protocol_transport*,
    char **sent_by_host*, int *sent_by_port*, char **via_params*);

int *sip_add_maxforward*(sip_msg_t *sip_msg*, uint_t *maxforward*);

int *sip_add_callid*(sip_msg_t *sip_msg*, char **callid*);

int *sip_add_cseq*(sip_msg_t *sip_msg*, sip_method_t *method*, uint32_t *cseq*);

int *sip_add_content_type*(sip_msg_t *sip_msg*, char * *type*, char **subtype*);

int *sip_add_content*(sip_msg_t *sip_msg*, char * *content*);

int *sip_add_accept*(sip_msg_t *sip_msg*, char **type*, char **subtype*,
    char **media_param*, char **accept_param*);

int *sip_add_accept_enc*(sip_msg_t *sip_msg*, char **code*,
    char **param*);

int *sip_add_accept_lang*(sip_msg_t *sip_msg*, char **lang*,
    char **param*);

int *sip_add_alert_info*(sip_msg_t *sip_msg*, char **alert*,
    char **param*);

int *sip_add_allow*(sip_msg_t *sip_msg*, sip_method_t *method_name*);

```
int sip_add_call_info(sip_msg_t sip_msg, char *uri,
     char *param);

int sip_add_content_disp(sip_msg_t sip_msg, char *dis_type, char *param);

int sip_add_content_enc(sip_msg_t sip_msg, char *code);

int sip_add_content_lang(sip_msg_t sip_msg, char *lang);

int sip_add_date(sip_msg_t sip_msg, char *date);

int sip_add_error_info(sip_msg_t sip_msg, char *uri, char *param);

int sip_add_expires(sip_msg_t sip_msg, int secs);

int sip_add_in_reply_to(sip_msg_t sip_msg, char *reply_id);

int sip_add_mime_version(sip_msg_t sip_msg, char *version);

int sip_add_min_expires(sip_msg_t sip_msg, int secs);

int sip_add_org(sip_msg_t sip_msg, char *org);

int sip_add_priority(sip_msg_t sip_msg, char *prio);

int sip_add_reply_to(sip_msg_t sip_msg, char *display_name,
     char *addr, char *param, boolean_t add_aquot);

int sip_add_passertedid(sip_msg_t sip_msg, char *display_name,
     char *addr, boolean_t add_aqout);

int sip_add_ppreferredid(sip_msg_t sip_msg, char *display_name,
     char *addr, boolean_t add_aquot);

int sip_add_require(sip_msg_t sip_msg, char *req);

int sip_add_retry_after(sip_msg_t sip_msg, int secs, char *cmt,
     char *param);

int sip_add_route(sip_msg_t sip_msg, char *display_name, char *uri,
     char *route_params);

int sip_add_record_route(sip_msg_t sip_msg, char *display_name,
     char *uri, char *route_params);

int sip_add_server(sip_msg_t sip_msg, char *svr);

int sip_add_subject(sip_msg_t sip_msg, char *subject);

int sip_add_supported(sip_msg_t sip_msg, char *support);

int sip_add_tstamp(sip_msg_t sip_msg, char *time, char *delay);

int sip_add_unsupported(sip_msg_t sip_msg, char *unsupport);

int sip_add_user_agent(sip_msg_t sip_msg, char *usr);

int sip_add_warning(sip_msg_t sip_msg, int code, char *addr, char *msg);

int sip_add_privacy(sip_msg_t sip_msg, char *priv_val);
```

```
int sip_add_rseq(sip_msg_t sip_msg, int resp_num);

int sip_add_rack(sip_msg_t sip_msg, int resp_num, int cseq,
    sip_method_t method);

int sip_add_author(sip_msg_t sip_msg, char *scheme, char *param);

int sip_add_authen_info(sip_msg_t sip_msg, char *ainfo);

int sip_add_proxy_authen(sip_msg_t sip_msg, char *pascheme,
    char *param);

int sip_add_proxy_author(sip_msg_t sip_msg, char *pascheme,
    char *param);

int sip_add_proxy_require(sip_msg_t sip_msg, char *opt);

int sip_add_www_authen(sip_msg_t sip_msg, char *wascheme,
    char *param);

int sip_add_allow_events(sip_msg_t sip_msg, char *events);

int sip_add_event(sip_msg_t sip_msg, char *event, char *param);

int sip_add_substate(sip_msg_t sip_msg, char *sub, char *param);
```

Description   For each of the following functions that add a header to a SIP message, the function adds a CRLF before appending the header to the SIP message.

The sip_add_from() and sip_add_to() functions appends a FROM and TO header respectively to the SIP message *sip_msg*. The header is created using the *display_name*, if non-null, and the *uri* values. The *add_aquot* parameter is used to specify whether the *uri* should be enclosed within '<>'. If a *display_name* is provided then *add_aquot* cannot be B_FALSE. The *display_name* parameter, if provided, is enclosed within quotes before creating to the SIP header. Tag value for the FROM/TO header can be specified which will be added to the SIP header by prefixing it with "TAG=". Any generic parameters can be specified as the last argument, which will be added, as is, to the SIP header.

Either the tag or the generic parameter can be specified not both, if both are specified, the resulting header contains only the tag parameter.

The sip_add_contact() function appends a CONTACT header to the SIP message *sip_msg* using the *display_name* and *contact_uri*. The *add_aquot* parameter has the same semantics as in sip_add_from()/sip_add_to(). Any contact parameters specified in *contact_param* is added to the CONTACT header before appending the header to the message.

The sip_add_via() function appends a VIA header to the SIP message *sip_msg*. The VIA header is constructed using sent_protocol_transport, sent_by_host and *sent_by_port*. A value of 0 for *sent_by_port* means that the port information is not present in the resulting VIA header. The VIA header that is created has the protocol set to "SIP" and version set to "2.0". Any parameters specific in *via_params* is added to the VIA header before appending the header to the SIP message.

The `sip_add_maxforward()` function appends a MAX-FORWARDS header to the SIP message *sip_msg* using the value in *maxforward*. The *maxforward* value is a positive integer.

The `sip_add_callid()` function appends a CALL-ID header to the SIP message *sip_msg* using the value in *callid*, if non-null. If *callid* is null, this function creates a CALL-ID header using a randomly generated value.

The `sip_add_cseq()` function appends a CSEQ header to the SIP message using the values in *method* and *cseq*. Permissible values for method include:

INVITE
ACK
OPTIONS
BYE
CANCEL
REGISTER
REFER
SUBSCRIBE
NOTIFY
PRACK
INFO

The *cseq* value is a positive integer.

The `sip_add_content_type()` function appends a CONTENT-TYPE to the SIP message *sip_msg*. The CONTENT-TYPE is created using the type and subtype, both should be non-null.

The `sip_add_content()` function adds a message body to the SIP message *sip_msg*. The message body is given by the null terminated string contents. Once the function returns, the caller may reuse or delete contents as `sip_add_content()` creates a new buffer and copies over contents for its use.

The `sip_add_accept()` function appends an ACCEPT header to the SIP message *sip_msg*. The ACCEPT header is created using type and subtype. If both type and subtype are null, then an empty ACCEPT header is added to the SIP message. If type is non-null, but subtype is null, then the ACCEPT header has the specified type and sets the subtype in the header to '*'. Any *accept_param* or *media_param*, if provided, are added to the ACCEPT header before appending the header to the SIP message.

The `sip_add_accept_enc()` function appends an ACCEPT-ENCODING header to the SIP message *sip_msg*. The ACCEPT-ENCODING is created using code. Any parameter specified in *param* is added to the ACCEPT-ENCODING header before appending the header to the SIP message.

The `sip_add_accept_lang()` function appends an `ACCEPT-LANGUAGE` header to the SIP message *sip_msg*. The `ACCEPT-LANGUAGE` header is created using lang. Any parameter specified in *param* is added to the `ACCEPT-LANGUAGE` header before appending the header to the SIP message.

The `sip_add_alert_info()` function appends an `ALERT-INFO` header to the SIP message *sip_msg*. The `ALERT-INFO` header is created using alert. Any parameter specified in *param* is added to the `ALERT-INFO` header before appending the header to the SIP message.

The `sip_add_allow()` function appends an `ALLOW` header to the SIP message *sip_msg*. The `ALLOW` header is created using alert and method. Permissible values for method include:

INVITE
ACK
OPTIONS
BYE
CANCEL
REGISTER
REFER
INFO
SUBSCRIBE
NOTIFY
PRACK

The `sip_add_call_info()` function appends a `CALL-INFO` header to the SIP message *sip_msg*. The `CALL-INFO` header is created using *uri*. Any parameter specified in *param* is added to the `CALL-INFO` before appending the header to the SIP message.

The `sip_add_content_disp()` function appends a `CONTENT-DISPOSITION` header to the SIP message *sip_msg*. The `CONTENT-DISPOSITION` header is created using *disp_type*. Any parameter specified in *param* is added to the `CONTENT-DISPOSITION` header before appending the header to the SIP message.

The `sip_add_content_enc()` function appends a `CONTENT-ENCODING` header to the SIP message *sip_msg*. The `CONTENT-ENCODING` header is created using code.

The `sip_add_content_lang()` function appends a `CONTENT-LANGUAGE` header to the SIP message *sip_msg*. The `CONTENT-LANGUAGE` header is created using *lang*.

The `sip_add_date()` appends a `DATE` header to the SIP message *sip_msg*. The `DATE` header is created using the date information specified in date. The semantics for the date string is given is RFC 3261, section 25.1.

The `sip_add_error_info()` function appends an `ERROR-INFO` header to the SIP message *sip_msg*. The `ERROR-INFO` header is created using *uri*. An parameters specified in *param* is added to the `ERROR-INFO` header before adding the header to the SIP message.

The sip_add_expires() function appends an EXPIRES header to the SIP message *sip_msg*. The EXPIRES header is created using the seconds specified in *secs*.

The sip_add_in_reply_to() function appends a IN-REPLY-TO header to the SIP message *sip_msg*. The IN-REPLY-TO header is created using the *call-id* value specified in *reply_id*.

The sip_add_mime_version() function appends a MIME-VERSION header to the SIP message *sip_msg*. The MIME-VERSION header is created using version.

The sip_add_min_expires() function appends a MIN-EXPIRES header to the SIP message *sip_msg*. The MIN-EXPIRES is created using the time in seconds specified in *secs*.

The sip_add_org() function appends a ORGANIZATION header to the SIP message *sip_msg*. The ORGANIZATION header is created using the information specified in *org*.

The sip_add_priority() function appends a PRIORITY header to the SIP message *sip_msg*. The PRIORITY header is created using the value specified in *prio*.

The sip_add_reply_to() function appends a REPLY-TO header to the SIP message *sip_msg*. The REPLY-TO header is created using the *display_name*, if provided, and *addr*. The *add_aquot* parameter has the same semantics as in sip_add_from()/sip_add_to(). Any parameters specified in *param* is added to the REPLY-TO header before appending the header to the SIP message.

The sip_add_passertedid() function appends a P-ASSERTED-IDENTITY header to the SIP message *sip_msg*. The P-ASSERTED-IDENTITY header is created using the *display_name*, if provided, and the *addr*. The *add_aquot* parameter has the same semantics as in sip_add_from()/sip_add_to().

The sip_add_ppreferredid() function appends a P-PREFERRED-IDENTITY header to the SIP message *sip_msg*. The P-PREFERRED-IDENTITY header is created using the *display_name*, if provided, and the *addr*. The *add_aquot* parameter has the same semantics as in sip_add_from()/sip_add_to().

The sip_add_require() function appends a REQUIRE header to the SIP message *sip_msg*. The REQUIRE header is created using the information in *req*.

The sip_add_retry_after() function appends a RETRY-AFTER header to the SIP message *sip_msg*. The RETRY-AFTER is created using the time in seconds specified in *secs* comments, if any, in *cmt*. Any parameters specified in *param*, if provided, is added to the RETRY-AFTER header before appending the header to the SIP message.

The sip_add_route() function appends a ROUTE header to the SIP message *sip_msg*. The ROUTE header is created using the *display_name*, if any, and the *uri*. The *uri* is enclosed in '<>' before adding to the header. Parameters specified in *route_params* are added to the ROUTE header before appending the header to the SIP message.

The sip_add_record_route() function appends a RECORD-ROUTE header to the SIP message *sip_msg*. The RECORD-ROUTE header is created using the *display_name*, if any, and the *uri*. The *uri* parameter is enclosed in '<>' before adding to the header. Any parameters specified in *route_params* is added to the ROUTE header before appending the header to the SIP message.

The sip_add_server() function appends a SERVER header to the SIP message *sip_msg*. The SERVER header is created using the information in *srv*.

The sip_add_subject() function appends a SUBJECT header to the SIP message *sip_msg*. The SUBJECT header is created using the information in *subject*.

The sip_add_supported() function appends a SUPPORTED header to the SIP message *sip_msg*. The SUPPORTED header is created using the information in *support*.

The sip_add_tstamp() function appends a TIMESTAMP header to the SIP message *sip_msg*. The TIMESTAMP header is created using the time value in *time* and the delay value, if provided, in *delay*.

The sip_add_unsupported() function appends an UNSUPPORTED header to the SIP message *sip_msg*. The UNSUPPORTED header is created using the option-tag value in *unsupport*.

The sip_add_user_agent() function appends an USER-AGENT header to the SIP message *sip_msg*. The USER-AGENT header is created using the server-val specified in *usr*.

The sip_add_warning() function appends a WARNING header to the SIP message *sip_msg*. The WARNING header is created using the warn-code in *code*, warn-agent in *addr* and warn-test in *msg*.

The sip_add_privacy() function appends a PRIVACY header to the SIP message *sip_msg*. The PRIVACY header is created using the privacy value specified in *priv_val*.

The sip_add_rseq() function appends a RSEQ header to the SIP message *sip_msg*. The RSEQ header is created using the sequence number specified in *resp_num*.

The sip_add_rack() function appends a RACK header to the SIP message *sip_msg*. The RACK header is created using the sequence number in *resp_num*, the SIP method in *method* and the CSEQ number in *cseq*. Permissible values for method include: INVITE, ACK, OPTIONS, BYE, CANCEL, REGISTER, REFER, INFO, SUBSCRIBE, NOTIFY, PRACK.

The sip_add_author() function appends an AUTHORIZATION header to the SIP message *sip_msg*. The AUTHORIZATION header is created using scheme. Any parameter specified in *param* is added to the AUTHORIZATION header before the header is appended to the SIP message.

The sip_add_authen_info() function appends an AUTHENTICATION-INFO() header to the SIP message *sip_msg*. The AUTHENTICATION-INFO header is created using the authentication information in *ainfo*.

The sip_add_proxy_authen() function appends a PROXY-AUTHENTICATE header to the SIP message *sip_msg*. The PROXY-AUTHENTICATE is created using the value specified in *pscheme*. Any parameter in *param* is added to the PROXY-AUTHENTICATE header before adding the header to the SIP message.

The sip_add_proxy_author() function appends a PROXY-AUTHORIZATION header to the SIP message *sip_msg*. The PROXY-AUTHORIZATION header is created using the value specified in *pascheme*. Any parameter in *param* is added to the PROXY-AUTHORIZATION header before adding the header to the SIP message.

The sip_add_proxy_require() function appends a PROXY-REQUIRE header to the SIP message *sip_msg*. The PROXY-REQUIRE header is created using the option-tag in *opt*.

The sip_add_www_authen() function appends a WWW-AUTHENTICATE header to the SIP message *sip_msg*. The WWW-AUTHENTICATE header is created using the challenge in *wascheme*. Any parameter in *param* is added to the WWW-AUTHENTICATE header before adding the header to the SIP message.

The sip_add_allow_events() function appends an ALLOW-EVENTS header to the SIP message The ALLOW-EVENTS header is created using the event specified in events.

The sip_add_event() function appends an EVENT header to the SIP message. The EVENT header is created using the value specified in *event*. Any parameter in *param* is added to the EVENT header before appending the header to the SIP message.

The sip_add_substate() function appends a SUBSCRIPTION-STATE header to the SIP message. The SUBSCRIPTION-STATE header is created using the state specified in *sub*. Any parameter in *param* is added to the SUBSCRIPTION-STATE header before appending the header to the SIP message.

Return Values These functions return 0 on success and the appropriate error value on failure.

Errors On failure, functions that return an error value can return one of the following:

EINVAL Mandatory parameters are not provided, i.e. null.

For sip_add_from(), sip_add_to(), sip_add_contact(), sip_add_reply_to(), sip_add_passertedid(), sip_add_ppreferredid() if *display_name* is non-null and *add_aquot* is B_FALSE.

For sip_add_branchid_to_via() the topmost VIA header already has a branch *param* or the SIP message does not have a VIA header.

EPERM The message cannot be modified.

ENOMEM There is an error allocating memory for creating headers/parameters.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_add_header – add a SIP header to the SIP message

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

int *sip_add_header*(sip_msg_t *sip_msg*, char **header_string*);

Description The sip_add_header() function takes the SIP header *header_string*, adds a CRLF (carriage
return/line feed) and appends it to the SIP message *sip_msg*. The sip_add_header() function
is typically used when adding a SIP header with multiple values.

Return Values The sip_add_header() function returns 0 on success and the appropriate error value on
failure.

Errors On failure, the sip_add_header() function can return one of the following error values:

EINVAL   Mandatory parameters are not provided, i.e. null.

EPERM   The message cannot be modified.

ENOMEM   Error allocating memory for creating headers/parameters.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name   sip_add_param – add a parameter to the SIP header

Synopsis   cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

sip_header_t *sip_add_param*(sip_header_t *sip_header*, char **param*,
    int **error*);

Description   The sip_add_param() function adds the parameter provided in *param* to the SIP header
sip_header. The function returns the header with the parameter added. A new header is
created as a result of adding the parameter and the old header is marked deleted. Applications
with multiple threads working on the same SIP header need to take note of this. If error is
non-null, it (the location pointer by the variable) is set to 0 on success and the appropriate
error value on error.

Return Values   The sip_add_param() function returns the new header on success and null on failure.
Further, if error is non-null, then on success the value in the location pointed by error is 0 and
the appropriate error value on failure.

Errors   On failure, functions that return an error value may return one of the following:

EINVAL   Mandatory parameters are not provided, i.e. null.

For sip_add_param(), the header to be modified is marked deleted.

EPERM   The message cannot be modified.

ENOMEM   There is an error allocating memory for creating headers/parameters.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also   libsip(3LIB)

Name sip_add_request_line, sip_add_response_line – add a request/response line to a SIP message

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

int sip_add_request_line(sip_msg_t *sip_request*,
    sip_method_t *method*, char *request_uri*);

int *sip_add_response_line*(sip_msg_t *sip_response*,
    int *response_code*, char *response_phase*);

Description The sip_add_request_line() function adds a request line to the SIP message *sip_request*.
The request line is created using the SIP method specified in *method* and the URI in
request_uri. The SIP method can be one of the following:

    INVITE
    ACK
    OPTIONS
    BYE
    CANCEL
    REGISTER
    REFER
    SUBSCRIBE
    NOTIFY
    PRACK
    INFO

The resulting request line has the SIP-Version of "2.0".

The sip_add_response_line() function adds a response line to the SIP message
sip_response. The response line is created using the response code *response_code* and the
phrase in *response_phrase*. If the *response_code* is one that is listed in RFC 3261,
sip_get_resp_desc() can be used to get the response phase for the *response_code*. The
resulting response line has the SIP-Version of "2.0".

Return Values The sip_add_response_line() and sip_add_request_line() functions return 0 on success
and the appropriate error value in case of failure.

The value of errno is not changed by these calls in the event of an error.

Errors On failure, the sip_add_response_line() and sip_add_request_line() functions could
return one of the following errors:

EINVAL   If mandatory input is not provided or if the input is invalid.

ENOTSUP   If the input SIP message cannot be modified.

ENOMEM   If memory allocation fails when creating the request/response line or when
            creating headers in the ACK request.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name   sip_branchid – generate a RFC 3261 complaint branch ID

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

char *sip_branchid(sip_msg_t *sip_msg*);

Description The sip_branchid() function can be used to generate a value for the branch parameter for a
VIA header. The returned string is prefixed with z9hG4bK to conform to RFC 3261. If *sip_msg*
is null or *sip_msg* does not have a VIA header, a random value is generated. Otherwise, the
value is generated using the MD5 hash of the VIA, FROM, CALL-ID, CSEQ headers and the URI
from the request line. The caller is responsible for freeing the returned string.

Return Values The sip_branchid() function returns a string on success and NULL on failure.

The value of errno is not changed by these calls in the event of an error.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_clone_msg – clone a SIP message

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

sip_msg_t *sip_clone_msg*(sip_msg_t *sip_msg*);

Description The sip_clone_msg() function clones the input SIP message and returns the cloned message.
The resulting cloned message has all the SIP headers and message body, if present, from the
input message.

Return Values The sip_clone_msg() function returns the cloned message on success and NULL on failure.

The value of errno is not changed by these calls in the event of an error.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_copy_start_line, sip_copy_header, sip_copy_header_by_name, sip_copy_all_headers – copy headers from a SIP message

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

int *sip_copy_start_line*(sip_msg_t *from_msg*, sip_msg_t *to_msg*);

int *sip_copy_header*(sip_msg_t *sip_msg*, sip_header_t *sip_header*,
    char *param);

int *sip_copy_header_by_name*(sip_msg_t *from_msg*, sip_msg_t *to_msg*,
    char *header_name, char *param);

int *sip_copy_all_headers*(sip_msg_t *from_msg*, sip_msg_t *to_msg*);

Description The sip_copy_start_line() function copies the start line, a request or a response line, from *from_msg* to *to_msg*.

The sip_copy_header() function copies the SIP header specified by *sip_header* to the SIP message *sip_msg*. A new SIP header is created from *sip_header* and *param*, and is appended to *sip_msg*. The *param* can be non-null.

The sip_copy_header_by_name() function copies the header specified by *header_name* (long or short form) from *from_msg* to *to_msg*. The new header is created using the header value from *from_msg* and *param*, if non-null, and appended to *to_msg*.

The sip_copy_all_headers() copies all the headers from *from_msg* to *to_msg*.

Return Values These functions return 0 on success and the appropriate error on failure.

The value of errno is not changed by these calls in the event of an error.

Errors These functions can return one of the following errors in case of failure:

EINVAL   If the required input parameters are NULL or if the header being copied does not exist or is deleted in source SIP message.

ENOMEM   Error while allocating memory for creating the new header.

EPERM   If the input SIP message cannot be modified.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_create_dialog_req, sip_create_dialog_req_nocontact – create an in-dialog request

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

sip_msg_t sip_create_dialog_req(sip_method_t *method*,
    sip_dialog_t *dialog*, char **transport*, char **sent_by*,
    int *sent_by_port*, char **via_param*, uint32_t *smaxforward*,
    int *cseq*);

sip_msg_t sip_create_dialog_req_nocontact(sip_method_t *method*,
    sip_dialog_t *dialog*, char **transport*, char **sent_by*,
    int *sent_by_port*, char **via_param*, uint32_t *smaxforward*,
    int *cseq*);

Description The sip_create_dialog_req() function creates and returns a SIP request with the state
information contained in *dialog*. The method in the resulting request is from *method*. The
method can be one of the following:

```
INVITE
ACK
OPTIONS
BYE
CANCEL
REGISTER
REFER
INFO
SUBSCRIBE
NOTIFY
PRACK
```

The resulting request line in the SIP message has the SIP-Version of "2.0". The URI in the
request line is from the remote target in the *dialog* or from the route set in the *dialog*, if
present. See RFC 3261 (section 12.2) for details. The FROM, TO, and CALL-ID headers are added
from the *dialog*. The MAX-FORWARDS header is added using the value in *maxforward*. The CSEQ
header is added using the SIP method in *method* and the sequence number value in *cseq*. If
cseq is -1, the sequence number is obtained from the local sequence number in the *dialog*. The
local sequence number in the *dialog* is incremented and is used in the CSEQ header. The VIA
header added is created using the *transport*, *sent_by*, *sent_by_port* (if *non-zero*), and
via_param (if any). If *dialog* has a non-empty route set, the resulting SIP request has the route
set from the *dialog*.

The sip_create_dialog_req_nocontact() function is similar to
sip_create_dialog_req(), except that it does not add the contact header.

Return Values The sip_create_dialog_req() and sip_create_dialog_req_nocontact() functions return the resulting SIP message on success and NULL on failure.

The value of errno is not changed by these calls in the event of an error.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB), attributes(5)

Name sip_create_OKack – create an ACK request for a final response

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

int *sip_create_OKack*(sip_msg_t *response*,
     sip_msg_t *ack_msg*, char **transport*,
     char **sent_by*, int *sent_by_port*,
     char **via_params*);

Description The sip_create_OKack() function constructs an ACK request in *ack_msg* for the final 2*XX* SIP response. The request line is created using the URI in the CONTACT header from the *response*. The SIP-Version in the request line is "2.0". The VIA header for the ACK request is created using *transport*, *sent_by*, *sent_by_port* (if non-zero), and *via_params* (if non-null). The following headers are copied to *ack_msg* from *response*:

FROM
TO
CALL-ID
MAX_FORWARDS

The CSEQ header is created using the method as ACK and the sequence number from the CSEQ header in *response*.

Return Values The sip_create_OKack() function returns 0 on success and the appropriate error value in case of failure.

The value of errno is not changed by these calls in the event of an error.

Errors On failure, the sip_create_OKack() function could return one of the following errors:

EINVAL   If mandatory input is not provided or if the input is invalid.

The sip_create_OKack() function can return this error if it does not find a CONTACT header or if it is unable to obtain the URI from the CONTACT header for the request line.

ENOTSUP   If the input SIP message cannot be modified.

ENOMEM   If memory allocation fails when creating the request/response line or when creating headers in the ACK request.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also libsip(3LIB)

Name   sip_create_response – create a response for a SIP request

Synopsis   cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

sip_msg_t *sip_create_response*(sip_msg_t *sip_request*,
    int *response_code*, char **response_phase*,
    char **totag*, char **contact_uri*);

Description   The sip_create_response() function creates and returns a SIP message in response to the
SIP request *sip_request*. The response line in the resulting SIP message is created using the
response code in *response_code* and the phrase in *response_phrase*. The response line has the
SIP-Version of "2.0". If a non-null *totag* is specified, the resulting SIP response has a TO header
with a tag value from *totag*. If *totag* is null and the *response_code* is anything other than 100
(TRYING), sip_create_response() adds a TO header with a randomly generated tag value. If
the *response_code* is 100 and *totag* is null, the SIP response has a TO header without a tag
parameter. If *contact_uri* is non-null, a CONTACT header is added to the SIP response with the
URI specified in *contact_uri*. The SIP response has the following headers copied from
sip_request:

All VIA headers
FROM header
TO header (with tag added, if required, as stated above)
CALL-ID header
CSEQ header
All RECORD-ROUTE headers

Return Values   The sip_create_response() function returns the resulting SIP message on success and NULL
on failure.

The value of errno is not changed by these calls in the event of an error.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also   libsip(3LIB)

Name sip_delete_dialog – delete a dialog

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

void *sip_delete_dialog*(sip_dialog_t *dialog*);

Description For functions that return a pointer of type sip_str_t, sip_str_t is supplied by:

```
typedef struct sip_str {
    char    *sip_str_ptr;
    int     sip_str_len;
}sip_str_t;
```

The *sip_str_ptr* parameter points to a specified value at the start of an input string. The *sip_str_len* supplies the length of the returned value starting from *sip_str_ptr*.

The sip_delete_dialog() function is used to delete the dialog specified in *dialog*. The dialog is not freed if it has outstanding references on it. When the last reference is released the dialog is freed.

Return Values The value of errno is not changed by these calls in the event of an error.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_delete_start_line, sip_delete_header, sip_delete_header_by_name, sip_delete_value – delete a SIP header or a header value

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

int *sip_delete_start_line*(sip_msg_t *sip_msg*);

int *sip_delete_header*(sip_msg_t *sip_header*);

int *sip_delete_header_by_name*(sip_msg_t *msg*,
    char **header_name*);

int *sip_delete_value*(sip_header_t *sip_header*,
    sip_header_value_t *sip_header_value*);

Description The sip_delete_start_line() function deletes the start line, a request or a response line, from the SIP message *sip_msg*.

The sip_delete_header() function deletes the SIP header specified by *sip_header* from the associated SIP message *sip_msg*.

The sip_delete_header_by_name() function deletes the SIP header name specified by *header_name* (long or compact form) from the SIP message *sip_msg*.

The sip_delete_value() deletes the SIP header value specified by *sip_header_value* from the SIP header *sip_header*.

When a SIP header or value is deleted, the corresponding header or value is marked as deleted. Lookups ignore headers or values that are marked as deleted.

Return Values These functions return 0 on success and the appropriate error on failure.

The value of errno is not changed by these calls in the event of an error.

Errors On failure, the returned error could be one of the following:

EINVAL   If any of the required input is NULL.

           If the header or value to be deleted does not exist.

           If the header or value to be deleted has already been deleted.

EPERM   If the SIP message cannot be modified.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also   libsip(3LIB)

Name sip_enable_counters, sip_disable_counters, sip_get_counter_value – counter operations

Synopsis cc [*flag*...] *file*... -lsip [*library*...]
#include <sip.h>

int sip_enable_counters(int *counter_group*);

int sip_disable_counters(int *counter_group*);

int sip_get_counter_value(int *group*, int *counter*, void **counterval*,
    size_t *counterlen*);

Description The sip_enable_counters() function enables the measurement and counting of the selected
counter group. The only allowed value for the *counter_group* is SIP_TRAFFIC_COUNTERS,
which is defined in <sip.h>. Once enabled, the SIP stack starts measuring end-to-end SIP
traffic. The SIP stack keeps track of:

- the number of SIP requests sent and received (broken down by methods),
- the number of SIP responses sent and received (broken down by response codes), and
- the number of bytes sent and received.

The following counters are defined in <sip.h> for the SIP_TRAFFIC_COUNTERS group. These
counter values are retrieved using the sip_get_counter_value() function.

```
SIP_TOTAL_BYTES_RCVD
SIP_TOTAL_BYTES_SENT
SIP_TOTAL_REQ_RCVD
SIP_TOTAL_REQ_SENT
SIP_TOTAL_RESP_RCVD
SIP_TOTAL_RESP_SENT
SIP_ACK_REQ_RCVD
SIP_ACK_REQ_SENT
SIP_BYE_REQ_RCVD
SIP_BYE_REQ_SENT
SIP_CANCEL_REQ_RCVD
SIP_CANCEL_REQ_SENT
SIP_INFO_REQ_RCVD
SIP_INFO_REQ_SENT
SIP_INVITE_REQ_RCVD
SIP_INVITE_REQ_SENT
SIP_NOTIFY_REQ_RCVD
SIP_NOTIFY_REQ_SENT
SIP_OPTIONS_REQ_RCVD
SIP_OPTIONS_REQ_SENT
SIP_PRACK_REQ_RCVD
SIP_PRACK_REQ_SENT
SIP_REFER_REQ_RCVD
SIP_REFER_REQ_SENT
SIP_REGISTER_REQ_RCVD
```

```
                    SIP_REGISTER_REQ_SENT
                    SIP_SUBSCRIBE_REQ_RCVD
                    SIP_SUBSCRIBE_REQ_SENT
                    SIP_UPDATE_REQ_RCVD
                    SIP_UPDATE_REQ_SENT
                    SIP_1XX_RESP_RCVD
                    SIP_1XX_RESP_SENT
                    SIP_2XX_RESP_RCVD
                    SIP_2XX_RESP_SENT
                    SIP_3XX_RESP_RCVD
                    SIP_3XX_RESP_SENT
                    SIP_4XX_RESP_RCVD
                    SIP_4XX_RESP_SENT
                    SIP_5XX_RESP_RCVD
                    SIP_5XX_RESP_SENT
                    SIP_6XX_RESP_RCVD
                    SIP_6xx_RESP_SENT
                    SIP_COUNTER_START_TIME  /* records time when counting was enabled */
                    SIP_COUNTER_STOP_TIME   /* records time when counting was disabled */
```

All of the above counters are defined to be uint64_t, except for SIP_COUNTER_START_TIME
and SIP_COUNTER_STOP_TIME, which are defined to be time_t.

The sip_disable_counters() function disables measurement and counting for the specified
counter_group. When disabled, the counter values are not reset and are retained until the
measurement is enabled again. Calling sip_enable_counters() again would reset all counter
values to zero and counting would start afresh.

The sip_get_counter_value() function retrieves the value of the specified counter within
the specified counter group. The value is copied to the user provided buffer, *counterval*, of
length *counterlen*. For example, after the following call, *invite_rcvd* would have the correct
value.

```
uint64_t invite_rcvd;

sip_get_counter_value(SIP_TRAFFIC_COUNTERS, SIP_INVITE_REQ_RCVD,
        &invite_rcvd, sizeof (uint64_t));
```

Return Values Upon successful completion, sip_enable_counters() and sip_disable_counters() return
0. They will return EINVAL if an incorrect group is specified.

Upon successful completion, sip_get_counter_value() returns 0. It returns EINVAL if an
incorrect counter name or counter size is specified, or if *counterval* is NULL.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also attributes(5)

Name sip_enable_trans_logging, sip_enable_dialog_logging, sip_disable_dialog_logging, sip_disable_trans_logging – transaction and dialog logging operations

Synopsis cc [*flag*...] *file*... -lsip [*library*...]
#include <sip.h>

int sip_enable_trans_logging(FILE **logfile*, int *flags*);

int sip_enable_dialog_logging(FILE **logfile*, int *flags*);

void sip_disable_dialog_logging();

void sip_disable_trans_logging();

Description The sip_enable_trans_logging() and sip_enable_dialog_logging() functions enable transaction and dialog logging respectively. The *logfile* argument points to a file to which the SIP messages are logged. The flags argument controls the amount of logging. The only flag defined in <sip.h> is SIP_DETAIL_LOGGING. Either transaction or dialog logging, or both, can be enabled at any time. For dialog logging to work, the SIP stack must be enabled to manage dialogs (using SIP_STACK_DIALOGS, see sip_stack_init(3SIP)) when the stack is initialized.

All the messages exchanged within a transaction/dialog is captured and later dumped to a log file when the transaction or dialog is deleted or terminated. Upon termination, each dialog writes to the file the messages that were processed in its context. Similarly, upon termination each transaction writes to the file the messages that were processed in its context.

The sip_disable_trans_logging() and sip_disable_dialog_logging() functions disable the transaction or dialog logging. These functions do not close the files. It is the responsibility of the application to close them.

The log contains the state of the transaction or dialog at the time the message was processed.

Return Values Upon successful completion, sip_enable_trans_logging() and sip_enable_dialog_logging() return 0. They return EINVAL if *logfile* is NULL or *flags* is unrecognized.

Examples EXAMPLE 1 Dialog logging

The following is an example of dialog logging.

```
FILE    *logfile;

logfile = fopen("/tmp/ApplicationA", "a+");
sip_enable_dialog_logging(logfile, SIP_DETAIL_LOGGING);

/* Application sends INVITE, recieves 180 and 200 response and dialog is
        created. */
/* Application sends ACK request */
/* Application sends BYE and recieves 200 response */
```

EXAMPLE 1 Dialog logging   *(Continued)*

```
/* Application disables logging */
sip_disable_dialog_logging();
```

The log file will be of the following format.

```
************* Begin Dialog *************
Digest      : 43854 43825 26120 9475 5415 21595 25658 18538

-----------------------------
Dialog State            : SIP_DLG_NEW

Tue Nov 27 15:53:34 2007| Message - 1
INVITE sip:user@example.com SIP/2.0
From: "Me" < sip:me@mydomain.com > ; TAG=tag-from-01
To: "You" < sip:you@yourdomain.com >
Contact: < sip:myhome.host.com >
MAX-FORWARDS: 70
Call-ID: 1261K6A6492KF33549XM
CSeq: 111 INVITE
CONTENT-TYPE: application/sdp
Via: SIP/2.0/UDP 192.0.0.1 : 5060 ;branch=z9hG4bK-via-EVERYTHINGIDO-05
Record-Route: <sip:server1.com;lr>
Record-Route: <sip:server2.com;lr>
CONTENT-LENGTH : 0

Tue Nov 27 15:53:34 2007| Message - 2
SIP/2.0 180 Ringing
Via: SIP/2.0/UDP 192.0.0.1 : 5060 ;branch=z9hG4bK-via-EVERYTHINGIDO-05
From: "Me" < sip:me@mydomain.com > ; TAG=tag-from-01
To: "You" < sip:you@yourdomain.com >;tag=1
Call-ID: 1261K6A6492KF33549XM
CSeq: 111 INVITE
Contact: <sip:whitestar2-0.East.Sun.COM:5060;transport=UDP>
Record-Route: <sip:server1.com;lr>
Record-Route: <sip:server2.com;lr>
Content-Length: 0

-----------------------------
Dialog State            : SIP_DLG_EARLY

/* Entire 200 OK SIP Response */

-----------------------------
Dialog State            : SIP_DLG_CONFIRMED
```

EXAMPLE 1 Dialog logging   *(Continued)*

```
/* Entire ACK Request */

/* Entire BYE Request */
/* Entire 200 OK Response */
-----------------------------
************ End Dialog ************
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also sip_stack_init(3SIP), attributes(5)

Name sip_get_contact_display_name, sip_get_from_display_name, sip_get_to_display_name, sip_get_from_tag, sip_get_to_tag, sip_get_callid, sip_get_callseq_num, sip_get_callseq_method, sip_get_via_sent_by_host, sip_get_via_sent_by_port, sip_get_via_sent_protocol_version, sip_get_via_sent_protocol_name, sip_get_via_sent_transport, sip_get_maxforward, sip_get_content_length, sip_get_content_type, sip_get_content_sub_type, sip_get_content, sip_get_accept_type, sip_get_accept_sub_type, sip_get_accept_enc, sip_get_accept_lang, sip_get_alert_info_uri, sip_get_allow_method, sip_get_min_expires, sip_get_mime_version, sip_get_org, sip_get_priority, sip_get_replyto_display_name, sip_get_replyto_uri_str, sip_get_date_time, sip_get_date_day, sip_get_date_month, sip_get_date_wkday, sip_get_date_year, sip_get_date_timezone, sip_get_content_disp, sip_get_content_enc, sip_get_error_info_uri, sip_get_expires, sip_get_require, sip_get_subject, sip_get_supported, sip_get_tstamp_delay, sip_get_unsupported, sip_get_server, sip_get_user_agent, sip_get_warning_code, sip_get_warning_agent, sip_get_warning_text, sip_get_call_info_uri, sip_get_in_reply_to, sip_get_retry_after_time, sip_get_retry_after_cmts, sip_get_rack_resp_num, sip_get_rack_cseq_num, sip_get_rack_method, sip_get_rseq_resp_num, sip_get_priv_value, sip_get_passertedid_display_name, sip_get_passertedid_uri_str, sip_get_ppreferredid_display_name, sip_get_ppreferredid_uri_str, sip_get_author_scheme, sip_get_author_param, sip_get_authen_info, sip_get_proxy_authen_scheme, sip_get_proxy_authen_param, sip_get_proxy_author_scheme, sip_get_proxy_author_param, sip_get_proxy_require, sip_get_www_authen_scheme, sip_get_www_authen_param, sip_get_allow_events, sip_get_event, sip_get_substate, sip_get_content_lang, sip_get_tstamp_value, sip_get_route_uri_str, sip_get_route_display_name, sip_get_contact_uri_str, sip_get_from_uri_str, sip_get_to_uri_str – obtain header specific attributes

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

const sip_str_t *sip_get_contact_display_name(sip_header_value_t *value*,
    int *error*);

const sip_str_t *sip_get_from_display_name(sip_msg_t *sip_msg*,
    int *error*);

const sip_str_t *sip_get_to_display_name(sip_msg_t *sip_msg*,
    int *error*);

const sip_str_t *sip_get_contact_uri_str(sip_header_value_t *value*,
    int *error*);

const sip_str_t *sip_get_from_uri_str(sip_msg_t *sip_msg*,
    int *error*);

const sip_str_t *sip_get_to_uri_str(sip_msg_t *sip_msg*,
    int *error*);

const sip_str_t *sip_get_from_tag(sip_msg_t *sip_msg*,
    int *error*);

```
const sip_str_t *sip_get_to_tag(sip_msg_t sip_msg,
    int *error);

const sip_str_t *sip_get_callid(sip_msg_t sip_msg,
    int *error);

int sip_get_callseq_num(sip_msg_t sip_msg,
    int *error);

sip_method_t sip_get_callseq_method(sip_msg_t sip_msg,
    int *error);

const sip_str_t *sip_get_via_sent_by_host(sip_header_value_t value,
    int *error);

int sip_get_via_sent_by_port (sip_header_value_t value,
    int *error);

const sip_str_t *sip_get_via_sent_protocol_version
    (sip_header_value_t value, int *error);

const sip_str_t *sip_get_via_sent_transport(sip_header_value_t value,
    int *error);

int sip_get_maxforward(sip_msg_t sip_msg,
    int *error);

int sip_get_content_length(sip_msg_t sip_msg,
    int *error);

const sip_str_t *sip_get_content_type(sip_msg_t sip_msg,
    int *error);

const sip_str_t *sip_get_content_sub_type(sip_msg_t sip_msg,
    int *error);

char *sip_get_content(sip_msg_t sip_msg,
    int *error);

const sip_str_t *sip_get_accept_type(sip_header_value_t value,
    int *error);

const sip_str_t *sip_get_accept_sub_type(sip_header_value_t value,
    int *error);

const sip_str_t *sip_get_accept_enc(sip_header_value_t value,
    int *error);

const sip_str_t *sip_get_accept_lang(sip_header_value_t value,
    int *error);

const sip_str_t *sip_get_alert_info_uri(sip_header_value_t value,
    int *error);

sip_method_t sip_get_allow_method(sip_header_value_t value,
    int *error);
```

```
int sip_get_min_expire(sip_msg_t sip_msg,
     int *error);

const sip_str_t *sip_get_mime_version(sip_msg_t sip_msg,
     int *error);

const sip_str_t *sip_get_org(sip_msg_t sip_msg,
     int *error);

const sip_str_t *sip_get_priority(sip_msg_t sip_msg,
     int *error);

const sip_str_t *sip_get_replyto_display_name(sip_msg_t sip_msg,
     int *error);

const sip_str_t *sip_get_replyto_uri_str(sip_msg_t sip_msg,
     int *error);

const sip_str_t *sip_get_date_time(sip_msg_t sip_msg,
     int *error);

int sip_get_date_day(sip_msg_t sip_msg,
     int *error);

const sip_str_t *sip_get_date_month(sip_msg_t sip_msg,
     int *error);

int sip_get_date_year(sip_msg_t sip_msg,
     int *error);

const sip_str_t *sip_get_date_wkday(sip_msg_t sip_msg,
     int *error);

const sip_str_t *sip_get_date_timezone(sip_msg_t sip_msg,
     int *error);

const sip_str_t *sip_get_content_disp(sip_msg_t sip_msg,
     int *error);

const sip_str_t *sip_get_content_enc(sip_header_value_t value,
     int *error);

const sip_str_t *sip_get_error_info_uri(sip_header_value_t value,
     int *error);

int sip_get_expires(sip_msg_t sip_msg,
     int *error);

const sip_str_t *sip_get_require(sip_header_value_t value,
     int *error);

const sip_str_t *sip_get_subject(sip_msg_t sip_msg,
     int *error);

const sip_str_t *sip_get_supported(sip_header_value_t value,
     int *error);
```

```
const sip_str_t *sip_get_tstamp_delay(sip_msg_t sip_msg,
    int *error);

const sip_str_t *sip_get_unsupported(sip_header_value_t value,
    int *error);

const sip_str_t *sip_get_server(sip_msg_t sip_msg,
    int *error);

const sip_str_t *sip_get_user_agent(sip_msg_t sip_msg,
    int *error);

int sip_get_warning_code(sip_header_value_t value,
    int *error);

const sip_str_t *sip_get_warning_agent(sip_header_value_t value,
    int *error);

const sip_str_t *sip_get_warning_text(sip_header_value_t value,
    int *error);

const sip_str_t *sip_get_call_info_uri(sip_header_value_t value,
    int *error);

const sip_str_t *sip_get_in_reply_to(sip_header_value_t value,
    int *error);

int sip_get_retry_after_time(sip_msg_t sip_msg,
    int *error);

const sip_str_t *sip_get_retry_after_cmts(sip_msg_t sip_msg,
    int *error);

const sip_str_t *sip_get_passertedid_display_name
    (sip_header_value_t value, int *error);

const sip_str_t *sip_get_passertedid_uri_str
    (sip_header_value_t value, int *error);

int sip_get_rack_resp_num(sip_msg_t sip_msg,
    int *error);

int sip_get_rack_cseq_num(sip_msg_t sip_msg, int *error);

sip_method_t sip_get_rack_method(sip_msg_t sip_msg, int *error);

int sip_get_rseq_resp_num(sip_msg_t sip_msg,
    int *error);

const sip_str_t *sip_get_priv_value(sip_header_value_t value,
    int *error);

const sip_str_t *sip_get_author_scheme(sip_msg_t sip_msg,
    int *error);

const sip_str_t *sip_get_author_param(sip_msg_t sip_msg,
    char *name, int *error);
```

```
const sip_str_t *sip_get_authen_info(sip_header_value_t value,
     int *error);
```

```
const sip_str_t *sip_get_proxy_authen_scheme(sip_msg_t msg,
     int *error);
```

```
const sip_str_t *sip_get_proxy_authen_param(sip_msg_t sip_msg,
     char *name, int *error);
```

```
const sip_str_t *sip_get_proxy_author_scheme(sip_msg_t msg,
     int *error);
```

```
const sip_str_t *sip_get_proxy_author_param(sip_msg_t sip_msg,
     char *name, int *error);
```

```
const sip_str_t *sip_get_proxy_require(sip_header_value_t value,
     int *error);
```

```
const sip_str_t *sip_get_www_authen_scheme(sip_msg_t msg,
     int *error);
```

```
const sip_str_t *sip_get_www_authen_param(sip_msg_t sip_msg,
     char *name, int *error);
```

```
const sip_str_t *sip_get_allow_events(sip_header_value_t value, int *error);
```

```
const sip_str_t *sip_get_event(sip_msg_t sip_msg,
     int *error);
```

```
const sip_str_t *sip_get_substate(sip_msg_t sip_msg,
     int *error);
```

```
const sip_str_t *sip_get_content_lang(sip_header_value_t value,
     int *error);
```

```
const sip_str_t *sip_get_tstamp_value(sip_msg_t sip_msg,
     int *error);
```

```
const sip_str_t *sip_get_route_uri_str(sip_header_value_t value,
     int *error);
```

```
const sip_str_t *sip_get_route_display_name(sip_header_value_t value,
     int *error);
```

Description For functions that return a pointer of type sip_str_t, sip_str_t is supplied by:

```
typedef struct sip_str {
     char    *sip_str_ptr;
     int     sip_str_len;
}sip_str_t;
```

The *sip_str_ptr* parameter points to the start of the returned value and *sip_str_len* supplies the length of the returned value.

For example, given the following request line in a SIP message *sip_msg* that is input to sip_get_request_uri_str():

```
FROM : <Alice sip:alice@atlanta.com>;tag=1928301774
```

the return is a pointer to *sip_str_t* with the *sip_str_ptr* member pointing to "A" of Alice and *sip_str_len* being set to 5, the length of Alice.

Access functions for headers that can have multiple values take the value as the input, while those that can have only one value take the SIP message *sip_msg* as the input.

The sip_get_contact_display_name(), sip_get_from_display_name(), and sip_get_to_display_name() functions will return the display name, if present, from the CONTACT header value, FROM and TO header respectively.

The sip_get_contact_uri_str(), sip_get_from_uri_str(), and sip_get_to_uri_str() functions will return the URI string from the CONTACT value, FROM and TO header respectively.

The sip_get_from_tag() and sip_get_to_tag() functions will return the TAG parameter value, if present, from the FROM and TO header, respectively, in the provided SIP message *sip_msg*.

The sip_get_callid() function will return the value from the CALL-ID header in the provided SIP message *sip_msg*.

The sip_get_callseq_num() function will return the call sequence number from the CSEQ header in the provided SIP message *sip_msg*.

The sip_get_callseq_method() function will return the method from the CSEQ header in the provided SIP message *sip_msg*. The method can be one of the following:

```
INVITE
ACK
OPTIONS
BYE
CANCEL
REGISTER
REFER
INFO
SUBSCRIBE
NOTIFY
PRACK
UNKNOWN
```

The sip_get_via_sent_by_host(), sip_get_via_sent_by_port(), sip_get_via_sent_protocol_version(), sip_get_via_sent_protocol_name(), and sip_get_via_sent_transport() functions will return the sent-by host, port (if present), protocol version, protocol name and transport information from the provided VIA header

value. Example, if the VIA value is given by SIP/2.0/UDP bobspc.biloxi.com:5060, then the
sent-by host is "bobspc.biloxi.com", protocol name is "SIP", protocol version is "2.0", port
is 5060 and transport is UDP.

The sip_get_maxforward() function will return the value of the MAX-FORWARDS header in the
provided SIP message *sip_msg*.

```
INVITE
ACK
OPTIONS
BYE
CANCEL
REGISTER
REFER
INFO
SUBSCRIBE
NOTIFY
PRACK
UNKNOWN
```

The sip_get_content_length() function will return the value of the CONTENT-LENGTH
header in the provided SIP message *sip_msg*. The method can return one of the following:

The sip_get_content_type() and sip_get_content_sub_type() functions will return the
value of the Type and Sub-Type field, respectively, from the CONTENT-TYPE header in the
provided SIP message *sip_msg*.

The sip_get_content() function will return the message body from the provided SIP
message *sip_msg*. The returned string is a copy of the message body and the caller is
responsible for freeing the string after use.

The sip_get_accept_type() and sip_get_accept_sub_type() functions will return the
value of the Type and Sub-Type field, respectively, from the provided ACCEPT header value.

The sip_get_accept_enc() function will return the content-coding from the provided
ACCEPT-ENCODING header value.

The sip_get_accept_lang() function will return the language from the provided
ACCEPT-LANGUAGE header value.

The sip_get_alert_info_uri() function will return the URI string from the provided
ALERT-INFO header value.

The sip_get_allow_method() function will return the SIP method from the provided ALLOW
header value. The method can return one of the following:

```
INVITE
```

```
ACK
OPTIONS
BYE
CANCEL
REGISTER
REFER
INFO
SUBSCRIBE
NOTIFY
PRACK
UNKNOWN
```

The sip_get_min_expire() function will return the time in seconds from the MIN-EXPIRES header in the provided SIP message *sip_msg*.

The sip_get_mime_version() function will return the MIME version string from the MIME-VERSION header in the provided SIP message *sip_msg*.

The sip_get_org() function will return the organization string value from the ORGANIZATION header in the provided SIP message *sip_msg*.

The sip_get_priority() function will return the priority string value from the PRIORITY header in the provided SIP message *sip_msg*.

The sip_get_replyto_display_name() and sip_get_replyto_uri_str() functions will return the display name (if present) and the URI string, respectively, from the REPLY-TO header in the provided SIP message *sip_msg*.

The sip_get_date_time(), sip_get_date_day(), sip_get_date_month(), sip_get_date_wkday(), sip_get_date_year() and sip_get_date_timezone() functions will return the time, day, month, week day, year and timezone value from the DATE header in the provided SIP message *sip_msg*. Example, if the DATE header has the following value:

```
Sat, 13 Nov 2010 23:29:00 GMT
```

the time is "23:29:00", week day is "Sat", day is "13", month is "Nov", year is "2010", timezone is "GMT".

The sip_get_content_disp() function will return the content-disposition type from the CONTENT-DISPOSITION header in the provided SIP message *sip_msg*.

The sip_get_content_enc() function will return the content-coding value from the CONTENT-ENCODING header value.

The sip_get_error_info_uri() function will return the URI string from the provided ERROR-INFO header value.

The sip_get_expires() function will return the time in seconds from the EXPIRES header in the provided SIP message *sip_msg*.

The `sip_get_require()` function will return the option-tag value from the provided REQUIRE header value.

The `sip_get_subject()` function will return the value of the SUBJECT header in the provided SIP message *sip_msg*.

The `sip_get_supported()` function will return the extension value from the provided SUPPORTED header value.

The `sip_get_tstamp_delay()` function will return the value from the TIMESTAMP header in the provided SIP message *sip_msg*.

The `sip_get_unsupported()` function will return the extension value from the provided UNSUPPORTED header value.

The `sip_get_server()` function will return the value from the SERVER header in the provided SIP message *sip_msg*.

The `sip_get_user_agent()` function will return the value from the USER-AGENT header in the provided SIP message *sip_msg*.

The `sip_get_warning_code()`, `sip_get_warning_agent()`, and `sip_get_warning_text()` functions will return the value of the warn-code, warn-agent and warn-text, respectively, in the provided WARNING header value.

The `sip_get_call_info_uri()` function will return the URI string in the provided CALL-INFO header value.

The `sip_get_in_reply_to()` function will return the Call-Id value in the provided IN-REPLY-TO header value.

The `sip_get_retry_after_time()`, and `sip_get_retry_after_cmts()` functions return the time and comments (if any), respectively, from the RETRY-AFTER header in the provided SIP message *sip_msg*.

The `sip_get_passertedid_display_name()` and `sip_get_passertedid_uri_str()` functions will return the display name (if any) and the URI string, respectively, in the provided P-ASSERTED-IDENTITY header value.

The `sip_get_ppreferredid_display_name()` and `sip_get_ppreferredid_uri_str()` functions will return the display name (if any) and the URI string, respectively, in the provided P-PREFERRED-IDENTITY header value.

The `sip_get_rack_resp_num()`, `sip_get_rack_cseq_num()`, and `sip_get_rack_method()` functions will return the response-number, the CSEQ number and the SIP method from the RACK header in the provided SIP message *sip_msg*. The method can return one of the following:

INVITE

```
ACK
OPTIONS
BYE
CANCEL
REGISTER
REFER
INFO
SUBSCRIBE
NOTIFY
PRACK
UNKNOWN
```

The sip_get_rseq_resp_num() function will return the response-number, the RSEQ header in the provided SIP message *sip_msg*.

The sip_get_priv_value() function will return the priv-value in the provided PRIVACY header value.

The sip_get_route_uri_str() and sip_get_route_display_name() functions will return the URI string, and display name (if present) from the provided ROUTE or RECORD-ROUTE header value.

The sip_get_author_scheme() function will return the scheme from the AUTHORIZATION header in the provided SIP message *sip_msg*.

The sip_get_author_param() function will return the value of the parameter specified in name from the AUTHORIZATION header in the SIP message *sip_msg*.

The sip_get_authen_info() function will return the authentication information from the provided AUTHORIZATION-INFO header value.

The sip_get_proxy_authen_scheme() function will return the scheme from the PROXY-AUTHENTICATE header in the SIP message *sip_msg*.

The sip_get_proxy_authen_param() function will return the value of the parameter in name from the PROXY-AUTHENTICATE header in the SIP message *sip_msg*.

The sip_get_proxy_author_scheme() function will return the value of the scheme from the PROXY-AUTHORIZATION header in the SIP message *sip_msg*.

The sip_get_proxy_author_param() function will return the value of the parameter specified in name from the PROXY-AUTHORIZATION header in the SIP message *sip_msg*.

The sip_get_proxy_require() function will return the option-tag from the provided PROXY-REQUIRE header value.

The sip_get_www_authen_scheme() function will return the challenge from the WWW-AUTHENTICATE header in the SIP message *sip_msg*.

The sip_get_www_authen_param() function will return the value of the parameter specified in name from the WWW-AUTHENTICATE header in the SIP message *sip_msg*.

The sip_get_allow_events() function returns the value of the allowed event from the provided ALLOW-EVENTS header value.

The sip_get_event() function returns the event in the EVENT header in the SIP message *sip_msg*.

The sip_get_substate() function the subscription state from the SUBSCRIPTION-STATE header in the provided SIP message *sip_msg*.

The sip_get_content_lang() function will return the language from the provided CONTENT-LANGUAGE value.

The sip_get_tstamp_value() function will return the timestamp value from the TIMESTAMP header in the SIP message *sip_msg*.

Return Values For functions that return a pointer to *sip_str_t*, the return value is the specified value on success or NULL in case of error. For functions that return an integer, the return value is the specified value on success and -1 on error.

The value of errno is not changed by these calls in the event of an error.

Errors These functions take a pointer to an integer *error* as an argument. If the error is non-null, one of the following values is set:

EINVAL   The input SIP message *sip_msg* or the header value is null; or the specified header/header value is deleted.

EPROTO   The header value is not present or invalid. The parser could not parse it correctly.

ENOMEM   There is an error allocating memory for the return value.

On success, the value of the location pointed to by *error* is set to 0.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_get_cseq, sip_get_rseq – get initial sequence number

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

uint32_t *sip_get_cseq*();

uint32_t *sip_get_rseq*();

Description The sip_get_cseq() and sip_get_rseq() functions can be used to generate an initial
sequence number for the CSEQ and RSEQ headers.

Return Values The value of errno is not changed by these calls in the event of an error.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_get_dialog_state, sip_get_dialog_callid, sip_get_dialog_local_tag, sip_get_dialog_remote_tag, sip_get_dialog_local_uri, sip_get_dialog_remote_uri, sip_get_dialog_local_contact_uri, sip_get_dialog_remote_target_uri, sip_get_dialog_route_set, sip_get_dialog_local_cseq, sip_get_dialog_remote_cseq, sip_get_dialog_type, sip_get_dialog_method, sip_is_dialog_secure, sip_get_dialog_msgcnt – get dialog attributes

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
```
#include <sip.h>
```
```
int sip_get_dialog_state(sip_dialog_t dialog, int *error);
```
```
const sip_str_t *sip_get_dialog_callid(sip_dialog_t dialog,
     int *error);
```
```
const sip_str_t *sip_get_dialog_local_tag(sip_dialog_t dialog,
     int *error);
```
```
const sip_str_t *sip_get_dialog_remote_tag(sip_dialog_t dialog,
     int *error);
```
```
const struct sip_uri *sip_get_dialog_local_uri(sip_dialog_t dialog,
     int *error);
```
```
const struct sip_uri *sip_get_dialog_remote_uri(sip_dialog_t dialog,
     int *error);
```
```
const struct sip_uri *sip_get_dialog_local_contact_uri(
     sip_dialog_t dialog, int *error);
```
```
const struct sip_uri *sip_get_dialog_remote_target_uri(
     sip_dialog_t dialog, int *error);
```
```
const sip_str_t *sip_get_dialog_route_set(sip_dialog_t dialog,
     int *error);
```
```
boolean_t sip_is_dialog_secure(sip_dialog_t dialog,
     int *error);
```
```
uint32_t sip_get_dialog_local_cseq(sip_dialog_t dialog,
     int *error);
```
```
uint32_t sip_get_dialog_remote_cseq(sip_dialog_t dialog,
     int *error);
```
```
int sip_get_dialog_type(sip_dialog_t dialog,int *error);
```
```
int sip_get_dialog_method(sip_dialog_t dialog,int *error);
```
```
int sip_get_dialog_msgcnt(sip_dialog_t dialog,int *error);
```

Description For functions that return a pointer of type sip_str_t, sip_str_t is supplied by:
```
typedef struct sip_str {
     char    *sip_str_ptr;
```

```
       int    sip_str_len;
}sip_str_t;
```

The *sip_str_ptr* parameter points to the start of the returned value and *sip_str_len* supplies the length of the returned value.

The sip_get_dialog_state() returns the state of the *dialog.* A *dialog* can be in one of the following states:

```
SIP_DLG_NEW
SIP_DLG_EARLY
SIP_DLG_CONFIRMED
SIP_DLG_DESTROYED
```

The sip_get_dialog_callid() function returns the call ID value maintained in the *dialog.*

The sip_get_dialog_local_tag() and sip_get_dialog_remote_tag() functions return the local and remote tag values, maintained in the *dialog.*

The sip_get_dialog_local_uri(), sip_get_dialog_remote_uri(), sip_get_dialog_local_contact_uri(), and sip_get_dialog_remote_target_uri() functions return the local, remote, local contract, and the remote target URIs, maintained in the *dialog.*

The sip_get_dialog_route_set() function returns the route set, if any, maintained in the *dialog.*

The sip_get_dialog_local_cseq() and sip_get_dialog_remote_cseq() functions return the local and remote CSEQ numbers maintained in the *dialog.*

The sip_get_dialog_type() function returns one of the following dialog types, depending on whether it is created by the client or the server.

SIP_UAC_DIALOG   created by client

SIP_UAS_DIALOG   created by server

The sip_get_dialog_method() function returns the SIP method, INVITE or SUBSCRIBE, of the request that created the dialog.

The sip_is_dialog_secure() function returns B_TRUE if the *dialog* is secure and B_FALSE otherwise.

The sip_get_dialog_msgcnt() function returns the number of SIP messages (requests and responses) that were sent and received within the context of the given dialog.

Return Values The sip_get_dialog_state(), sip_get_dialog_local_cseq(),
sip_get_dialog_remote_cseq(), sip_get_dialog_type(), sip_get_dialog_method(),
and sip_get_dialog_msgcnt() functions return the required value on success and -1 on
failure.

The sip_get_dialog_callid(), sip_get_dialog_local_tag(),
sip_get_dialog_remote_tag(), sip_get_dialog_local_uri(),
sip_get_dialog_remote_uri(), sip_get_dialog_local_contact_uri(),
sip_get_dialog_remote_target_uri(), and sip_get_dialog_route_set() functions
return the required value on success and NULL on failure.

The value of errno is not changed by these calls in the event of an error.

Errors These functions take an *error* argument.

If the error is non-null, one of the following values is set:

EINVAL
    The *dialog* is NULL or the stack is not configured to manage dialogs.

ENOTSUP
    The input SIP message cannot be modified.

ENOMEM
    The memory allocation fails when the request/response line or the headers in the ACK
    request are created.

On success, the value of the location pointed to by *error* is set to 0.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_get_header – get a SIP header from a message

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

const struct sip_header *sip_get_header(sip_msg_t *sip_msg*,
    char *header_name*, sip_header_t *old_header*, int *error*);

Description The sip_get_header() function returns the header specified by *header_name* (long or
compact form) from the SIP message *sip_msg*. If *header_name* is NULL, the first header in the
SIP message is returned. The old_header, if non-null, specifies the starting position in
sip_msg from which the search is started. Otherwise, the search begins at the start of the SIP
message. For example, to get the first VIA header from the SIP message *sip_msg*:

via_hdr = sip_get_header(sip_msg, "VIA", NULL, &error);

To get the next VIA header from *sip_msg*:

via_hdr = sip_get_header(sip_msg, "VIA", via_hdr, &error);

The sip_get_header() function ignores any header that is marked as deleted.

Return Values On success, the sip_get_header() function returns the queried header. On failure, it returns
NULL.

The value of errno is not changed by these calls in the event of an error.

Errors The following value may be returned:

EINVAL   The *header_name* specified in the SIP message is not present or has been deleted;
          or, the *header_name* is not specified and there are no "un-deleted" headers in the
          SIP message.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_get_header_value, sip_get_next_value – get a SIP header value

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

const struct sip_value **sip_get_header_value*
    (const struct sip_header **sip_header*, int *error*);

const struct sip_value **sip_get_next_value*
    (sip_header_value_t *old_value*, int *error*);

Description The sip_get_header_value() function returns the first valid value from SIP header *sip_header*.

The sip_get_next_value() function returns the next valid value following the SIP value *old_value*.

Return Values These functions return the queried value on success and NULL on failure.

The value of errno is not changed by these calls in the event of an error.

Errors If the error is non-null, one of the following values is set:

EINVAL   If any of the required input is NULL or if the specified SIP header value is marked deleted.

EPROTO   If the returned SIP header value is invalid (i.e. the parser encountered errors when parsing the value).

On success, the value of the location pointed to by *error* is set to 0.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_get_msg_len – returns the length of the SIP message

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

int *sip_get_msg_len*(sip_msg_t *sip_msg*,
    int **error*);

Description The sip_get_msg_len() function will return the length of the SIP message *sip_msg*.

Return Values For functions that return an integer, the return value is the specified value on success and -1 on error.

The value of errno is not changed by these calls in the event of an error.

Errors This function takes a pointer to an integer *error* as an argument. If the error is non-null, one of the following values is set:

EINVAL   The input SIP message *sip_msg* or the header value is null; or the specified header/header value is deleted.

EPROTO   The header value is not present or invalid. The parser could not parse it correctly.

ENOMEM   There is an error allocating memory for the return value.

On success, the value of the location pointed to by *error* is set to 0.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_get_num_via, sip_get_branchid – get VIA header specific attributes

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

int *sip_get_num_via*(sip_msg_t *sip_msg*,
    int **error*);

char **sip_get_branchid*(sip_msg_t *sip_msg*,
    int **error*);

Description The sip_get_num_via() function returns the number of VIA headers in the SIP message
sip_msg.

The sip_get_branchid() function returns the branch ID value from the topmost VIA header.
The caller is responsible for freeing the returned string.

Return Values The sip_get_num_via() function returns the number of VIA headers on success.

The sip_get_branchid() function returns the branch ID on success and NULL on failure.

The value of errno is not changed by these calls in the event of an error.

Errors If the error is non-null, one of the following values is set:

EINVAL   The *sip_msg* is NULL.

ENOENT   For the *sip_get_branchid* function, there is no VIA header or the VIA header has no
               branch parameter.

EPROTO   For the *sip_sip_get_trans.3sipget_branchid* function, the VIA value is invalid. The
               parser encountered an error or errors while parsing the VIA header.

ENOMEM   For the *sip_get_branchid* function, there is an error in allocating memory for the
               branch ID.

On success, the value of the location pointed to by *error* is set to 0.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_get_param_value, sip_get_params, sip_is_param_present – get parameter information for a SIP header value

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

```
const sip_str_t *sip_get_param_value
    (sip_header_value_t header_value, char *param_name, int *error);
```

```
const sip_param_t *sip_get_params
    (sip_header_value_t header_value, int *error);;
```

```
boolean_t sip_is_param_present
    (const sip_param_t *param_list, char *param_name, int param_len);
```

Description The sip_get_param_value() function returns the value for the parameter name specified by *param_name* from the SIP header value *header_value*.

For functions that return a pointer of type sip_str_t, sip_str_t is supplied by:

```
typedef struct sip_str {
    char    *sip_str_ptr;
    int     sip_str_len;
}sip_str_t;
```

The *sip_str_ptr* parameter points to the start of the returned value and *sip_str_len* supplies the length of the returned value.

The sip_get_params() function returns the parameter list, if any, for the SIP header value *header_value*.

The sip_is_param_present() function returns B_TRUE if the parameter specified by *param_name* of length supplied in *param_len* is present in the parameter list, *param_list*. Otherwise, it returns B_FALSE.

Return Values With the exception of sip_is_param_present(), these functions return the queried value on success and NULL on failure.

The value of errno is not changed by these calls in the event of an error.

Errors If the error is non-null, one of the following values is set:

EINVAL   If any of the required input is NULL or if the specified SIP header value is marked deleted.

EPROTO   If the returned SIP header value is invalid (i.e. the parser encountered errors when parsing the value).

On success, the value of the location pointed to by *error* is set to 0.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_get_request_method, sip_get_response_code, sip_get_response_phrase,
sip_get_sip_version – obtain attributes from the start line in a SIP message

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

sip_method_t *sip_get_request_method*(const sip_msg_t *sip_msg*,
    int **error*);

int *sip_get_response_code*(sip_msg_t *sip_msg*,
    int **error*);

const sip_str_t **sip_get_response_phrase*(sip_msg_t *sip_msg*,
    int **error*);

const sip_str_t**sip_get_sip_version*(sip_msg_t *sip_msg*,
    int **error*);

Description For functions that return a pointer of type sip_str_t, sip_str_t is supplied by:

```
typedef struct sip_str {
    char    *sip_str_ptr;
    int     sip_str_len;
}sip_str_t;
```

The *sip_str_ptr* parameter points to the start of the returned value and *sip_str_len* supplies the
length of the returned value.

For example, given the following request line in a SIP message *sip_msg* that is input to
sip_get_request_uri_str():

FROM : <Alice sip:alice@atlanta.com>;tag=1928301774

the return is a pointer to *sip_str_t* with the *sip_str_ptr* member pointing to "A" of Alice and
sip_str_len being set to 5, the length of Alice.

Access functions for headers that can have multiple values take the value as the input, while
those that can have only one value take the SIP message *sip_msg* as the input.

The sip_get_request_method() function will return the SIP method from the request line in
the SIP message *sip_msg*. The method can be one of the following:

INVITE
ACK
OPTIONS
BYE
CANCEL
REGISTER
REFER
INFO

SUBSCRIBE
NOTIFY
PRACK
UNKNOWN

The sip_get_response_code() function will return the response code *response* from the request line in the SIP message *sip_msg*.

The sip_get_respose_phrase() function will return the response phrase *response* from the request line in the SIP message *sip_msg*.

The sip_get_sip_version() function will return the version of the SIP protocol from the request or the response line in the SIP message *sip_msg*.

Return Values For functions that return a pointer to *sip_str_t*, the return value is the specified value on success or NULL in case of error. For functions that return an integer, the return value is the specified value on success and -1 on error.

The value of errno is not changed by these calls in the event of an error.

Errors These functions take a pointer to an integer *error* as an argument. If the error is non-null, one of the following values is set:

EINVAL   The input SIP message *sip_msg* or the header value is null; or the specified header/header value is deleted.

EPROTO   The header value is not present or invalid. The parser could not parse it correctly.

ENOMEM   There is an error allocating memory for the return value.

On success, the value of the location pointed to by *error* is set to 0.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_get_request_uri_str – return request URI

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

const sip_str_t *sip_get_request_uri_str(sip_msg_t *sip_msg,
    int *error);

Description For functions that return a pointer of type *sip_str_t*, *sip_str_t* is supplied by:

typedef struct sip_str {
    char *sip_str_ptr;
    int sip_str_len;
}sip_str_t;

The *sip_str_ptr* parameter points to the start of the returned value and *sip_str_len* supplies the length of the returned value.

For example, given the following request line in a SIP message input to
sip_get_request_uri_str():

INVITE sip:marconi@radio.org SIP/2.0

the return is a pointer to *sip_str_t* with the *sip_str_ptr* member pointing to "s" of
sip:marconi@radio.org and *sip_str_len* being set to 21, the length of
sip:marconi@radio.org.

The sip_get_request_uri_str() function returns the URI string from the request line in the SIP message *sip_msg*.

Return Values The sip_get_request_uri_str() function returns the URI string. The function returns NULL on failure.

The value of errno is not changed by these calls in the event of an error.

Errors If the error is non-null, one of the following values is set:

EINVAL   For the sip_get_request_uri_str() function, there is no request line is in the SIP message.

EPROTO   For *sip_get_request_uri_str*, the request URI is invalid.

On success, the value of the location pointed to by *error* is set to 0.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also   libsip(3LIB)

Name sip_get_resp_desc – return the response phrase

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

char *sip_get_resp_desc(int *resp_code*);

Description The sip_get_resp_desc() function returns the response phrase for the given response code
in *resp_code*. The response code is not one that is listed in RFC 3261 (Section 21). The
returned string is "UNKNOWN".

Return Values The value of errno is not changed by these calls in the event of an error.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_get_trans – lookup a transaction

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

const struct sip_xaction **sip_get_trans*(sip_msg_t *sip_msg*, int *which*,
    int **error*);

Description The sip_get_trans() transaction for the SIP message *sip_msg*. A transaction is not freed if
there are any references on it.

The transaction type should be specified as one of the following:

SIP_CLIENT_TRANSACTON - lookup a client transaction
SIP_SERVER_TRANSACTON - lookup a server transaction

The sip_get_trans() function matches a transaction to a message as specified in RFC 3261,
sections 17.1.3 and 17.2.3. The sip_get_trans() function holds a reference to the returned
transaction. The caller must release this reference after use.

Return Values The sip_get_trans() function returns the required value on success or NULL on failure.

The value of errno is not changed by these calls in the event of an error.

Errors On success, the value of the location pointed to by *error* is set to 0.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name   sip_get_trans_method, sip_get_trans_state, sip_get_trans_orig_msg,
sip_get_trans_conn_obj, sip_get_trans_resp_msg, sip_get_trans_branchid – get transaction
attributes

Synopsis   cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

sip_method_t *sip_get_trans_method*(sip_transaction_t *sip_trans*,
    int **error*);

int *sip_get_trans_state*(sip_transaction_t *trans*, int **error*);

const struct sip_message **sip_get_trans_orig_msg*
    (sip_transaction_t *sip_trans*, int **error*);

const struct sip_message **sip_get_trans_resp_msg*
    (sip_transaction_t *sip_trans*, int **error*);

const struct sip_conn_object **sip_get_trans_conn_obj*
    (sip_transaction_t *sip_trans*, int **error*);

char **sip_get_trans_branchid*(sip_transaction_t *trans*, int **error*);

Description   The sip_get_trans_method() function returns the method the SIP message that created the
transaction *sip_trans*.

The sip_get_trans_state() function returns the state of the transaction *sip_trans*.

A newly created transaction is in the state:

SIP_NEW_TRANSACTION

A client transaction could be in one of the following states:

SIP_CLNT_CALLING
SIP_CLNT_INV_PROCEEDING
SIP_CLNT_INV_TERMINATED
SIP_CLNT_INV_COMPLETED
SIP_CLNT_TRYING
SIP_CLNT_NONINV_PROCEEDING
SIP_CLNT_NONINV_TERMINATED
SIP_CLNT_NONINV_COMPLETED

A server transaction could be in one of the following states:

SIP_SRV_INV_PROCEEDING
SIP_SRV_INV_COMPLETED
SIP_SRV_CONFIRMED

SIP_SRV_INV_TERMINATED
SIP_SRV_TRYING
SIP_SRV_NONINV_PROCEEDING
SIP_SRV_NONINV_COMPLETED
SIP_SRV_NONINV_TERMINATED

The sip_get_trans_orig_msg() function returns the message that created the transaction *sip_trans*. This could be a request on the client or a response on the server.

The sip_get_trans_resp_msg() function returns the last response that was sent on the transaction *sip_trans*. Typically, this response is used by the transaction layer for retransmissions for unreliable transports or for responding to retransmitted requests. A response that terminates a transaction is not returned.

The sip_get_trans_conn_obj() function returns the cached connection object, if any, in the transaction*sip_trans*.

The sip_get_trans_branchid() function returns the branch ID for the message that created the transaction*sip_trans*. The caller is responsible for freeing the returned string.

Return Values The sip_get_trans_orig_msg(), sip_get_trans_resp_msg(), sip_get_trans_conn_obj(), and sip_get_trans_branchid() functions return the required value on success or NULL on failure.

The sip_get_trans_state() and sip_get_trans_method() functions return the required value on success and -1 on failure.

The value of errno is not changed by these calls in the event of an error.

Errors If the error is non-null, one of the following values is set:

EINVAL   The input transaction *sip_trans* is NULL.

ENOMEM   For sip_get_trans_branchid() there is an error allocating memory for the branch ID string.

On success, the value of the location pointed to by *error* is set to 0.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also   libsip(3LIB)

Name sip_get_uri_parsed – return the parsed URI

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

const struct sip_uri *sip_get_uri_parsed(sip_header_value_t *value*,
    int *error*);

Description The sip_get_uri_parsed() function returns the parsed URI *sip_uri* from the SIP header value specified in *value*.

Return Values The sip_get_uri_parsed() function returns the parsed URI *sip_uri* on success. The function returns NULL on failure.

The value of errno is not changed by these calls in the event of an error.

Errors If the error is non-null, following value is set:

EINVAL   The SIP header value of the SIP message is NULL or there is no URI.

The input URI is null or the requested URI component is invalid. The error flag is set for the requested component.

The URI parameters or headers are requested from a non-SIP[S] URI; or the 'opaque', 'query', 'path', 'reg-name' components are requested from a SIP[S] URI.

On success, the value of the location pointed to by *error* is set to 0.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_guid – generate a random string

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

char *sip_guid*();

Description The sip_guid() function can be used to generate a random string. The caller is responsible for freeing the returned string.

Return Values The sip_guid() function returns a string on success and NULL on failure.

The value of errno is not changed by these calls in the event of an error.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_hold_dialog, sip_release_dialog – hold/release reference on a dialog

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

void *sip_hold_dialog*(sip_dialog_t *dialog*);

void *sip_release_dialog*(sip_dialog_t *dialog*);

Description For functions that return a pointer of type sip_str_t, sip_str_t is supplied by:

```
typedef struct sip_str {
    char    *sip_str_ptr;
    int     sip_str_len;
}sip_str_t;
```

The *sip_str_ptr* parameter points to the start of the returned value and *sip_str_len* supplies the length of the returned value.

The sip_hold_dialog() function is used to hold a reference on the *dialog*. A dialog is not freed if there are any references on it.

The sip_release_dialog() function is used to release a reference in the *dialog*. If the reference in a dialog drops to 0 and it is in SIP_DLG_DESTROYED state, it is freed.

Return Values The value of errno is not changed by these calls in the event of an error.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_hold_msg, sip_free_msg – adds and removes a reference from a SIP message

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

void *sip_hold_msg*(sip_msg_t *sip_msg*);

void *sip_free_msg*(sip_msg_t *sip_msg*);

Description The sip_hold_msg() function adds a reference to the SIP message passed as the argument.
The reference is used to prevent the SIP message from being freed when in use.

The sip_free_msg() function is used to remove an added reference on the SIP message
passed as the argument. If this is the last reference on the SIP message (i.e. the number of
references on the SIP message is 0), the SIP message is destroyed and associated resources
freed. Freeing a SIP message does not set the *sip_msg* pointer to NULL. Applications should not
expect the pointer to a freed SIP message to be NULL.

Return Values The value of errno is not changed by these calls in the event of an error.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_hold_trans, sip_release_trans – hold or release reference on a transaction

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

void *sip_hold_trans*(sip_transaction_t *sip_trans*);

void *sip_release_trans*(sip_transaction_t *sip_trans*);

Description The sip_hold_trans() function is used to hold a reference on the transaction *sip_trans*. A transaction is not freed if there are any references on it.

The sip_release_trans() function is used to release a reference on the transaction *sip_trans*. If the reference falls to 0 and the transaction is in a terminated state, the transaction is freed.

Return Values The value of errno is not changed by these calls in the event of an error.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_init_conn_object, sip_clear_stale_data, sip_conn_destroyed – connection object related functions

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

int *sip_init_conn_object*(sip_conn_object_t *obj*);

void *sip_clear_stale_data*(sip_conn_object_t *obj*);

void *sip_conn_destroyed*(sip_conn_object_t *obj*);

Description The sip_init_conn_object() function initializes the connection object *obj* for use by the stack. The first member of the connection object (a void *) is used by the stack to store connection object specific stack-private data.

The sip_clear_stale_data() function is used to clear any stack-private data in the connection object *obj*.

The sip_conn_destroyed() function is used to intimate the stack of the pending destruction of the connection object *obj*. The stack clean up any stack-private data in *obj* and also removes *obj* from any caches the stack maintains.

Return Values The sip_init_conn_object() function returns 0 on success and the appropriate error value on failure.

The value of errno is not changed by these calls in the event of an error.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name   sip_is_sip_uri, sip_get_uri_scheme, sip_get_uri_host, sip_get_uri_user,
sip_get_uri_password, sip_get_uri_port, sip_get_uri_params, sip_get_uri_headers,
sip_get_uri_opaque, sip_get_uri_query, sip_get_uri_path, sip_get_uri_regname,
sip_is_uri_teluser, sip_get_uri_errflags, sip_uri_errflags_to_str – get URI related attributes

Synopsis   cc [*flag* ...] *file* ... -lsip [*library* ...]
```
#include <sip.h>
```

```
boolean_t sip_is_sip_uri(const struct sip_uri *sip_uri);
```

```
const sip_str_t *sip_get_uri_scheme(const struct sip_uri *sip_uri,
    int *error);
```

```
const sip_str_t *sip_get_uri_user(const struct sip_uri *sip_uri,
    int *error);
```

```
const sip_str_t *sip_get_uri_password(const struct sip_uri *sip_uri,
    int *error);
```

```
const sip_str_t *sip_get_uri_host(const struct sip_uri *sip_uri,
    int *error);
```

```
int sip_get_uri_port(const struct sip_uri *sip_uri,
    int *error);
```

```
const sip_param_t *sip_get_uri_params(const struct sip_uri *sip_uri,
    int *error);
```

```
const sip_str_t *sip_get_uri_headers(const struct sip_uri *sip_uri,
    int *error);
```

```
const sip_str_t *sip_get_uri_opaque(const struct sip_uri *sip_uri,
    int *error);
```

```
const sip_str_t *sip_get_uri_query(const struct sip_uri *sip_uri,
    int *error);
```

```
const sip_str_t *sip_get_uri_path(const struct sip_uri *sip_uri,
    int *error);
```

```
const sip_str_t *sip_get_uri_regname(const struct sip_uri *sip_uri,
    int *error);
```

```
boolean_t sip_is_uri_teluser(const struct sip_uri *sip_uri);
```

```
int sip_get_uri_errflags(const struct sip_uri *sip_uri,
    int *error);
```

```
char *sip_uri_errflags_to_str(int uri_errflags);
```

Description   For functions that return a pointer of type *sip_str_t*, *sip_str_t* is supplied by:

```
typedef struct sip_str {
    char *sip_str_ptr;
    int  sip_str_len;
}sip_str_t;
```

The *sip_str_ptr* parameter points to the start of the returned value and *sip_str_len* supplies the length of the returned value.

For example, given the following request line in a SIP message input to sip_get_request_uri_str():

INVITE sip:marconi@radio.org SIP/2.0

the return is a pointer to *sip_str_t* with the *sip_str_ptr* member pointing to "s" of sip:marconi@radio.org and *sip_str_len* being set to 21, the length of sip:marconi@radio.org.

The sip_is_sip_uri() function takes a parsed URI *sip_uri* and returns B_TRUE if it is a SIP[S] URI and B_FALSE if it is not. A URI is a SIP[S] URI if the scheme in the URI is either "sip" or "sips".

The sip_get_uri_user() function takes a parsed URI *sip_uri* and returns the value of the "user" component, if present.

The sip_get_uri_password() function takes a parsed URI *sip_uri* and returns the value of the "password" component, if present.

The sip_get_uri_host() function takes a parsed URI *sip_uri* and returns the value of the "host" component, if present.

The sip_get_uri_port() function takes a parsed URI *sip_uri* and returns the value of the "port" component, if present.

The sip_get_uri_params() function takes a parsed URI *sip_uri* and returns the list of URI parameters, if present, from a SIP[S] URI.

The sip_get_uri_headers() function takes a parsed URI *sip_uri* and returns 'headers' from a SIP[S] URI.

The sip_get_uri_query() function takes a parsed URI *sip_uri* and returns the value of the 'query' component, if present.

The sip_get_uri_path() function takes a parsed URI *sip_uri* and returns the value of the 'path' component, if present.

The sip_get_uri_regname() function takes a parsed URI *sip_uri* and returns the value of the 'regname' component, if present.

The sip_is_uri_teluser() function returns B_TRUE if the user component is a telephone-subscriber. Otherwise, B_FALSE is returned.

The sip_get_uri_errflags() function returns the error flags from a parsed URI *sip_uri*. The returned value is a bitmask with the appropriate bit set when the parser, sip_parse_uri(), encounters an error. The following are the possible error values that could be set:

Bit value	Error	Comments
0x00000001	SIP_URIERR_SCHEME	invalid scheme
0x00000002	SIP_URIERR_USER	invalid user name
0x00000004	SIP_URIERR_PASS	invalid password
0x00000008	SIP_URIERR_HOST	invalid host
0x00000010	SIP_URIERR_PORT	invalid port number
0x00000020	SIP_URIERR_PARAM	invalid URI parameters
0x00000040	SIP_URIERR_HEADER	invalid URI headers
0x00000080	SIP_URIERR_OPAQUE	invalid opaque
0x00000100	SIP_URIERR_QUERY	invalid query
0x00000200	SIP_URIERR_PATH	invalid path
0x00000400	SIP_URIERR_REGNAME	invalid reg-name

The sip_uri_errflags_to_str() function takes the error flags from a parsed URI *sip_uri* and forms a string with all the error bits that are set. For example, if SIP_URIERR_PASS and SIP_URIERR_PORT are set in a parsed URI *sip_uri*, the sip_uri_errflags_to_str() function returns a string such as:

```
"Error(s) in PASSWORD, PORT part(s)"
```

The caller is responsible for freeing the returned string.

Return Values The sip_get_uri_scheme(), sip_get_uri_user(), sip_get_uri_password(), sip_get_uri_host(), sip_get_uri_params(), sip_get_uri_headers(), sip_get_uri_opaque(), sip_get_uri_query(), sip_get_uri_path(), sip_get_uri_regname(), and sip_uri_errflags_to_str() functions return the requested value on success and NULL on failure.

The sip_get_uri_port() function returns *port* from the URI or 0 if the port is not present. The returned port is in host byte order.

The value of errno is not changed by these calls in the event of an error.

Errors If the error is non-null, the following value is set:

EINVAL   The SIP header value of the SIP message is NULL or there is no URI.

The input URI is null or the requested URI component is invalid. The error flag is set for the requested component.

The URI parameters or headers are requested from a non-SIP[S] URI; or the 'opaque', 'query', 'path', 'reg-name' components are requested from a SIP[S] URI.

On success, the value of the location pointed to by *error* is set to 0.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also   libsip(3LIB)

Name sip_msg_is_request, sip_message_is_response – determine if the SIP message is a request or a response

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

boolean_t *sip_msg_is_request*(const sip_msg_t *sip_msg*,
     int **error*);

boolean_t *sip_msg_is_response*(const sip_msg_t *sip_msg*,
     int **error*);

Description The sip_msg_is_request() function returns B_TRUE if *sip_msg* is a request and B_FALSE otherwise.

The sip_msg_is_response() function returns B_TRUE if *sip_msg* is a response and B_FALSE otherwise.

Return Values For functions that return an integer, the return value is the specified value on success and -1 on error.

The value of errno is not changed by these calls in the event of an error.

Errors These functions take a pointer to an integer *error* as an argument. If the error is non-null, one of the following values is set:

EINVAL   The input SIP message *sip_msg* or the header value is null; or the specified header/header value is deleted.

EPROTO   The header value is not present or invalid. The parser could not parse it correctly.

ENOMEM   There is an error allocating memory for the return value.

On success, the value of the location pointed to by *error* is set to 0.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_msg_to_str, sip_hdr_to_str, sip_reqline_to_str, sip_respline_to_str, sip_sent_by_to_str – return string representations

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

char *sip_msg_to_str(sip_msg_t *sip_msg*,
    int *error*);

char *sip_hdr_to_str(sip_header_t *sip_header*,
    int *error*);

char *sip_reqline_to_str(sip_msg_t *sip_msg*,
    int *error*);

char *sip_respline_to_str(sip_msg_t *sip_msg*,
    int *error*);

char *sip_sent_by_to_str(int *error*);

Description The sip_msg_to_str() function returns the string representation of the SIP message *sip_msg*. Deleted headers are not included in the returned string. The caller is responsible for freeing the returned string.

The sip_hdr_to_str() function returns the string representation of the SIP header *sip_header*. The caller is responsible for freeing the returned string.

The sip_reqline_to_str() function returns the string representation of the request line from the SIP message *sip_msg*. The caller is responsible for freeing the returned string.

The sip_respline_to_str() function returns the string representation of the response line from the SIP message *sip_msg*. The caller is responsible for freeing the returned string.

The sip_sent_by_to_str() function can be used to retrieve the list of sent-by values registered with the stack. The returned string is a comma separated list of sent-by values. The caller is responsible for freeing the returned string.

Return Values The sip_msg_to_str(), sip_hdr_to_str(), sip_reqline_to_str(), sip_respline_to_str(), and sip_sent_by_to_str() functions return the relevant string on success and NULL on failure.

The value of errno is not changed by these calls in the event of an error.

Errors For the sip_msg_to_str(), sip_hdr_to_str(), sip_reqline_to_str(), and sip_respline_to_str(), one of the following values is set if the error is non-null:

EINVAL   Input is null.

ENOMEM   Memory allocation failure.

On success, the value of the location pointed to by *error* is set to 0.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name   sip_new_msg – allocates a new SIP message

Synopsis   cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

sip_msg_t *sip_new_msg*();

Description   The sip_new_msg() function allocates and returns a new SIP message.

Return Values   The sip_new_msg() function returns the newly allocated SIP message on success and NULL on failure.

The value of errno is not changed by these calls in the event of an error.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also   libsip(3LIB)

Name sip_parse_uri, sip_free_parsed_uri – parse a URI and free a parsed URI

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

sip_uri_t *sip_parse_uri*(sip_str_t *uri_str*,
    int *error*);

void *sip_free_parsed_uri*(sip_uri_t *sip_uri*);

Description For functions that return a pointer of type *sip_str_t*, *sip_str_t* is supplied by:

```
typedef struct sip_str {
    char *sip_str_ptr;
    int  sip_str_len;
}sip_str_t;
```

The *sip_str_ptr* parameter points to the start of the returned value and *sip_str_len* supplies the length of the returned value.

For example, given the following request line in a SIP message input to
sip_get_request_uri_str():

INVITE sip:marconi@radio.org SIP/2.0

the return is a pointer to *sip_str_t* with the *sip_str_ptr* member pointing to "s" of
sip:marconi@radio.org and *sip_str_len* being set to 21, the length of
sip:marconi@radio.org.

The sip_parse_uri() function takes a URI string in the form *sip_str_t* and returns a parsed URI *sip_uri*. The syntax of the URI is as specified in RFC 3261, section 25.1. If the parser encounters an error when parsing a component, it sets the appropriate error bit in the error flags and proceeds to the next component, if present.

The sip_free_parsed_uri() function takes a parsed URI *sip_uri*, obtained from
sip_parse_uri(), and frees any associated memory.

Return Values The sip_parse_uri() function returns the parsed URI *sip_uri* on success. It returns a NULL if memory cannot be allocated for the parsed URI.

The value of errno is not changed by these calls in the event of an error.

Errors If the error is non-null, the following values is set:

EINVAL The SIP header value of the SIP message is NULL or there is no URI.

The input URI is null or the requested URI component is invalid. The error flag is set for the requested component.

The URI parameters or headers are requested from a non-SIP[S] URI; or the 'opaque', 'query', 'path', 'reg-name' components are requested from a SIP[S] URI.

On success, the value of the location pointed to by *error* is set to 0.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_process_new_packet – send an inbound message to the SIP stack for processing

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

void *sip_process_new_packet*(sip_conn_object_t *conn_object*,
    void **msgstr*, size_t *msgstr*);

Description The sip_process_new_packet() function receives incoming message, creates a SIP message, processes it and passes it on to the application. For a byte-stream protocol like TCP sip_process_new_packet() also takes care of breaking the byte stream into message boundaries using the CONTENT-LENGTH header in the SIP message. If the SIP message arriving on TCP does not contain a CONTENT-LENGTH header, the behavior is unspecified. sip_process_new_packet() deletes the SIP message on return from the application's receive function, thus if the application wishes to retain the SIP message for future use, it must use sip_hod_msg() so that the message is not freed by sip_process_new_packet().

Return Values The value of errno is not changed by these calls in the event of an error.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_register_sent_by, sip_unregister_sent_by, sip_unregister_all_sent_by – allows registering and un-registering sent-by values

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

int *sip_register_sent_by*(char *val*);

void *sip_unregister_sent_by*(char *val*);

void *sip_unregister_all_sent_by*(int *error*);

Description The sip_register_sent_by() function can be used to register a list of hostnames or IP addresses that the application may add to the VIA headers. The *val* is a comma separated list of such sent-by values. If any value is registered using sip_register_sent_by(), the SIP stack validates incoming responses to check if the sent-by parameter in the topmost VIA header is part of the registered list. If the check fails, the response is dropped. If there are no sent-by values registered, there is no check done on incoming responses.

The sip_unregister_sent_by() and sip_unregister_all_sent_by() functions are used to un-register sent-by values. The *val* for sip_unregister_sent_by() is a comma separated list of sent-by values that need to be un-registered. sip_unregister_all_sent_by() un-registers all the values that have been registered.

Return Values The sip_register_sent_by() function returns 0 on success and the appropriate error value on failure.

The value of errno is not changed by these calls in the event of an error.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_sendmsg – send an outbound SIP message to the SIP stack for processing

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

int *sip_sendmsg*(sip_conn_object_t *obj*, sip_msg_t *sip_msg*,
    sip_dialog_t *dialog*, uint32_t *flags*);

Description The sip_sendmsg() function is used to send an outbound SIP message *sip_msg* to the SIP
stack on its way to the peer. The connection object for the SIP message is passed as *obj*. The
caller also provides the dialog associated with the message, if one exists. The value of flags is
the result of ORing the following, as required:

SIP_SEND_STATEFUL   Send the request or response statefully. This results in the
stack creating and maintaining a transaction for this
request/response. If this flag is not set transactions are not
created for the request/response.

SIP_DIALOG_ON_FORK   When this flag is set, the stack may create multiple dialogs for a
dialog completing response. This may result due to forking of
the dialog creating request. If this flag is not set, the first
response to a dialog creating request creates a dialog, but
subsequent ones do not. It is only meaningful if the stack is
configured to maintain dialogs.

Return Values The sip_sendmsg() function returns 0 on success and the appropriate error on failure.

The value of errno is not changed by these calls in the event of an error.

Errors The sip_sendmsg() function can return one of the following errors on failure:

EINVAL   If a message is being statefully sent and the *branchid* in the VIA header does not
conform to RFC 3261 or when accessing CSEQ header while creating a transaction.

ENOENT   If a message is being statefully sent, error getting the CSEQ header while creating a
transaction.

EPROTO   If a message is being statefully sent, error getting the CSEQ value while creating a
transaction.

ENOMEM   If the message is being statefully sent, error allocating memory for creating or
adding a transaction or during transaction related processing.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also libsip(3LIB)

Name sip_stack_init – initializes SIP stack

Synopsis cc [*flag* ...] *file* ... -lsip [*library* ...]
#include <sip.h>

int *sip_stack_init*(sip_stack_init_t * *stack_val*);

Description The sip_stack_init() function is used to initialize the SIP stack. The stack can be initialized
by a process only once. Any shared library that is linked with a main program or another
library that has already initialized the stack will encounter a failure when trying to initialize
the stack.

The initialization structure is given by:

```
typedef struct sip_stack_init_s {
    int                    sip_version;
    uint32_t               sip_stack_flags;
    sip_io_pointers_t      *sip_io_pointers;
    sip_ulp_pointers_t     *sip_ulp_pointers;
    sip_header_function_t  *sip_function_table;
};
```

sip_version   This must be set to SIP_STACK_VERSION.

sip_stack_flags   If the application wants the SIP stack to maintain dialogs, this flag must be
set to SIP_STACK_DIALOGS. Otherwise, it must be set to 0. If
SIP_STACK_DIALOGS is not set, the stack does not deal with dialogs at all.

Upper Layer Registrations These include callbacks that are invoked to deliver incoming messages or error notification.

The callback functions should not create a thread and invoke a function that could recursively
invoke the callback. For example, the callback function for a transition state change
notification should not create a thread to send a SIP message that results in a change in the
state of the transaction, which would again invoke the callback function.

The registration structure is supplied by:

```
typedef struct sip_ulp_pointers_s {
    void     (*sip_ulp_recv)(const sip_conn_object_t,
                  sip_msg_t, const sip_dialog_t);
    uint_t   (*sip_ulp_timeout)(void *,
                  void (*func)(void *),
                  struct timeval *);
    boolean_t (*sip_ulp_untimeout)(uint_t);
    int      (*sip_ulp_trans_error)
                  (sip_transaction_t, int, void *);
    void     (*sip_ulp_dlg_del)(sip_dialog_t,
                  sip_msg_t, void *);
    void     (*sip_ulp_trans_state_cb)
                  (sip_transaction_t, sip_msg_t,
```

```
                                       int, int);
              void       (*sip_ulp_dlg_state_cb)(sip_dialog_t,
                                sip_msg_t, int, int);
          }sip_io_pointers_t;
```

sip_ulp_recv   This is a mandatory routine that the application registers for the
                            stack to deliver an inbound SIP message. The SIP stack invokes
                            the function with the connection object on which the message
                            arrived, the SIP message, and any associated dialog.

                            The SIP message is freed once the function returns. If the
                            application wishes to use the message beyond that, it has to hold a
                            reference on the message using sip_hold_msg(). Similarly, if the
                            application wishes to cache the dialog, it must hold a reference on
                            the dialog using sip_hold_msg().

sip_ulp_timeout
sip_ulp_untimeout   An application can register these two routines to implement its
                            own routines for the stack timers. Typically, an application should
                            allow the stack to use its own built-in timer routines. The built-in
                            timer routines are used only by the stack and are not available to
                            applications. If the application registers one routine, it must also
                            register the other.

                            These functions must be registered for single-threaded
                            application. Otherwise, the timer thread provided by the stack
                            could result in invoking a registered callback function.

sip_ulp_trans_error   The application can register this routine to be notified of a
                            transaction error. An error can occur when the transaction layer
                            tries to send a message using a cached connection object which
                            results in a failure. If this routine is not registered the transaction
                            is terminated on such a failure. The final argument is for future
                            use. It is always set to NULL.

sip_ulp_dlg_del   An application can register this routine to be notified when a
                            dialog is deleted. The dialog to be deleted is passed along with the
                            SIP message which caused the dialog to be deleted. The final
                            argument is for future use. It is always set to NULL.

sip_ulp_trans_state_cb
sip_ulp_dlg_state_cb   If these callback routines are registered, the stack invokes
                            sip_ulp_trans_state_cb when a transaction changes states and
                            sip_ulp_dlg_state_cb when a dialog changes states.

Connection Manager Interface

The connection manager interfaces must be registered by the application to provide I/O related functionality to the stack. These interfaces act on a connection object that is defined by the application. The application registers the interfaces for the stack to work with the connection object. The connection object is application defined, but the stack requires that the first member of the connection object is a void *, used by the stack to store connection object specific information which is private to the stack.

The connection manager structure is supplied by:

```
typedef struct sip_io_pointers_s {
        int     (*sip_conn_send)(const sip_conn_object_t, char *, int);
        void    (*sip_hold_conn_object)(sip_conn_object_t);
        void    (*sip_rel_conn_object)(sip_conn_object_t);
        boolean_t       (*sip_conn_is_stream)(sip_conn_object_t);
        boolean_t       (*sip_conn_is_reliable)(sip_conn_object_t);
        int     (*sip_conn_remote_address)(sip_conn_object_t, struct sockaddr *,
                    socklen_t *);
        int     (*sip_conn_local_address)(sip_conn_object_t, struct sockaddr *,
                    socklen_t *);
        int     (*sip_conn_transport)(sip_conn_object_t);
        int     (*sip_conn_timer1)(sip_conn_object_t);
        int     (*sip_conn_timer2)(sip_conn_object_t);
        int     (*sip_conn_timer4)(sip_conn_object_t);
        int     (*sip_conn_timerd)(sip_conn_object_t);
}sip_io_pointers_t;
```

sip_conn_send   This function is invoked by the stack after processing an
                             outbound SIP message. This function is responsible for
                             sending the SIP message to the peer. A return of 0 indicates
                             success. The SIP message is passed to the function as a string,
                             along with the length information and the associated
                             connection object.

sip_hold_conn_object
sip_rel_conn_object   The application provides a mechanism for the stack to indicate
                             that a connection object is in use by the stack and must not be
                             freed. The stack uses sip_hold_conn_object to indicate that
                             the connection object is in use and sip_rel_conn_object to
                             indicate that it has been released. The connection object is
                             passed as the argument to these functions. The stack expects
                             that the application will not free the connection object if it is in
                             use by the stack.

sip_conn_is_stream   The stack uses this to determine whether the connection
                             object, passed as the argument, is byte-stream oriented.
                             Byte-stream protocols include TCP while message-based
                             protocols include SCTP and UDP.

sip_conn_is_reliable | The stack uses this to determine whether the connection object, passed as the argument, is reliable. Reliable protocols include TCP and SCTP. Unreliable protocols include UDP.

sip_conn_local_address
sip_conn_remote_address | These two interfaces are used by the stack to obtain endpoint information for a connection object. The sip_conn_local_address provides the local address/port information. The sip_conn_remote_address provides the address/port information of the peer. The caller allocates the buffer and passes its associated length along with it. On return, the length is updated to reflect the actual length.

sip_conn_transport | The stack uses this to determine the transport used by the connection object, passed as the argument. The transport could be TCP, UDP, SCTP.

sip_conn_timer1
sip_conn_timer2
sip_conn_timer4
sip_conn_timerd | These four interfaces may be registered by an application to provide connection object specific timer information. If these are not registered the stack uses default values.

The interfaces provide the timer values for Timer 1 (RTT estimate - default 500 msec), Timer 2 (maximum retransmit interval for non-INVITE request and INVITE response - default 4 secs), Timer 4 (maximum duration a message will remain in the network - default 5 secs) and Timer D (wait time for response retransmit interval - default 32 secs).

Custom SIP headers   In addition to the SIP headers supported by the stack, an application can optionally provide a table of custom headers and associated parsing functions. The table is an array with an entry for each header. If the table includes headers supported by the stack, parsing functions or other application-specific table entries take precedence over libsip supported headers. The header table structure is supplied by:

```
typedef struct header_function_table {
    char      *header_name;
    char      *header_short_name;
    int       (*header_parse_func)
                  (struct sip_header *,
                      struct sip_parsed_header **);
    boolean_t (*header_check_compliance)
                  (struct sip_parsed_header *);
    boolean_t (*header_is_equal)
                  (struct sip_parsed_header *,
```

```
                          struct sip_parsed_header *);
        void     (*header_free)
                 (struct sip_parsed_header *);
}
```

header_name	The full name of the header. The application must ensure that he name does not conflict with existing headers. If it does, the one registered by the application takes precedence.
header_short_name	Compact name, if any, for the header.
header_parse_func	The parsing function for the header. The parser will set the second argument to the resulting parsed structure. A return value of 0 indicates success.
header_free	The function that frees the parsed header
header_check_compliance	An application can optionally provide this function that will check if the header is compliant or not. The compliance for a custom header will be defined by the application.
header_is_equal	An application can optionally provide this function to determine whether two input headers are equivalent. The equivalence criteria is defined by the application.

Return Values On success sip_stack_init() returns 0. Otherwise, the function returns the error value.

The value of errno is not changed by these calls in the event of an error.

Errors On failure, the sip_stack_init() function returns the following error value:

EINVAL   If the stack version is incorrect, or if any of the mandatory functions is missing.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also libsip(3LIB)

Name slp_api – Service Location Protocol Application Programming Interface

Synopsis cc [*flag* ...] *file* ... -lslp [*library* ...]
#include <slp.h>

Description The slp_api is a C language binding that maps directly into the Service Location Protocol ("SLP") defined by *RFC 2614*. This implementation requires minimal overhead. With the exception of the SLPDereg() and SLPDelAttrs() functions, which map into different uses of the SLP deregister request, there is one C language function per protocol request. Parameters are for the most part character buffers. Memory management is kept simple because the client allocates most memory and client callback functions are required to copy incoming parameters into memory allocated by the client code. Any memory returned directly from the API functions is deallocated using the SLPFree() function.

To conform with standard C practice, all character strings passed to and returned through the API are null-terminated, even though the SLP protocol does not use null-terminated strings. Strings passed as parameters are UTF-8 but they may still be passed as a C string (a null-terminated sequence of bytes.) Escaped characters must be encoded by the API client as UTF-8. In the common case of US-ASCII, the usual one byte per character C strings work. API functions assist in escaping and unescaping strings.

Unless otherwise noted, parameters to API functions and callbacks are non-NULL. Some parameters may have other restrictions. If any parameter fails to satisfy the restrictions on its value, the operation returns a PARAMETER_BAD error.

Syntax for String Parameters Query strings, attribute registration lists, attribute deregistration lists, scope lists, and attribute selection lists follow the syntax described in *RFC 2608*. The API reflects the strings passed from clients directly into protocol requests, and reflects out strings returned from protocol replies directly to clients. As a consequence, clients are responsible for formatting request strings, including escaping and converting opaque values to escaped byte-encoded strings. Similarly, on output, clients are required to unescape strings and convert escaped string-encoded opaques to binary. The SLPEscape() and SLPUnescape() functions can be used for escaping SLP reserved characters, but they perform no opaque processing.

Opaque values consist of a character buffer that contains a UTF-8-encoded string, the first characters of which are the non UTF-8 encoding "\ff". Subsequent characters are the escaped values for the original bytes in the opaque. The escape convention is relatively simple. An escape consists of a backslash followed by the two hexadecimal digits encoding the byte. An example is "\2c" for the byte 0x2c. Clients handle opaque processing themselves, since the algorithm is relatively simple and uniform.

System Properties The system properties established in slp.conf(4), the configuration file, are accessible through the SLPGetProperty() and SLPSetProperty() functions. The SLPSetProperty() function modifies properties only in the running process, not in the configuration file. Errors are checked when the property is used and, as with parsing the configuration file, are logged at the LOG_INFO priority. Program execution continues without interruption by substituting the default for the erroneous parameter. In general, individual agents should rarely be required to

override these properties, since they reflect properties of the SLP network that are not of concern to individual agents. If changes are required, system administrators should modify the configuration file.

Properties are global to the process, affecting all threads and all handles created with SLPOpen().

Memory Management   The only API functions that return memory specifically requiring deallocation on the part of the client are SLPParseSrvURL(), SLPFindScope(), SLPEscape(), and SLPUnescape(). Free this memory with SLPFree() when it is no longer needed. Do not free character strings returned by means of the SLPGetProperty() function.

Any memory passed to callbacks belongs to the library, and it must not be retained by the client code. Otherwise, crashes are possible. Clients must copy data out of the callback parameters. No other use of the memory in callback parameters is allowed.

Asynchronous and Incremental Return Semantics   If a handle parameter to an API function is opened asynchronously, the API function calls on the handle to check the other parameters, opens the appropriate operation, and returns immediately. If an error occurs in the process of starting the operation, the error code is returned. If the handle parameter is opened synchronously, the function call is blocked until all results are available, and it returns only after the results are reported through the callback function. The return code indicates whether any errors occurred during the operation.

The callback function is called whenever the API library has results to report. The callback code is required to check the error code parameter before looking at the other parameters. If the error code is not SLP_OK, the other parameters may be NULL or otherwise invalid. The API library can terminate any outstanding operation on which an error occurs. The callback code can similarly indicate that the operation should be terminated by passing back SLP_FALSE to indicate that it is not interested in receiving more results. Callback functions are not permitted to recursively call into the API on the same SLPHandle. If an attempt is made to call into the API, the API function returns SLP_HANDLE_IN_USE. Prohibiting recursive callbacks on the same handle simplifies implementation of thread safe code, since locks held on the handle will not be in place during a second outcall on the handle.

The total number of results received can be controlled by setting the net.slp.maxResults parameter.

On the last call to a callback, whether asynchronous or synchronous, the status code passed to the callback has value SLP_LAST_CALL. There are four reasons why the call can terminate:

DA reply received   A reply from a DA has been received and therefore nothing more is expected.

Multicast terminated   The multicast convergence time has elapsed and the API library multicast code is giving up.

Multicast null results   Nothing new has been received during multicast for awhile and the API library multicast code is giving up on that (as an optimization).

<table>
<tr><td>Maximum results</td><td>The user has set the net.slp.maxResults property and that number of replies has been collected and returned.</td></tr>
</table>

Configuration Files The API library reads slp.conf(4), the default configuration file, to obtain the operating parameters. You can specify the location of this file with the SLP_CONF_FILE environment variable. If you do not set this variable, or the file it refers to is invalid, the API will use the default configuration file at /etc/inet/slp.conf instead.

Data Structures The data structures used by the SLP API are as follows:

The URL Lifetime Type

```
typedef enum {
    SLP_LIFETIME_DEFAULT = 10800,
    SLP_LIFETIME_MAXIMUM = 65535
} SLPURLLifetime;
```

The enumeration SLPURLLifetime contains URL lifetime values, in seconds, that are frequently used. SLP_LIFETIME_DEFAULT is 3 hours, while SLP_LIFETIME_MAXIMUM is 18 hours, which corresponds to the maximum size of the lifetime field in SLP messages. Note that on registration SLP_LIFETIME_MAXIMUM causes the advertisement to be continually reregistered until the process exits.

The SLPBoolean Type

```
typedef enum {
    SLP_FALSE = 0,
    SLP_TRUE = 1
} SLPBoolean;
```

The enumeration SLPBoolean is used as a Boolean flag.

The Service URL Structure

```
typedef struct srvurl {
    char *s_pcSrvType;
    char *s_pcHost;
    int   s_iPort;
    char *s_pcNetFamily;
    char *s_pcSrvPart;
} SLPSrvURL;
```

The SLPSrvURL structure is filled in by the SLPParseSrvURL() function with information parsed from a character buffer containing a service URL. The fields correspond to different parts of the URL, as follows:

s_pcSrvType   A pointer to a character string containing the service type name, including naming authority.

s_pcHost	A pointer to a character string containing the host identification information.
s_iPort	The port number, or zero, if none. The port is only available if the transport is IP.
s_pcNetFamily	A pointer to a character string containing the network address family identifier. Possible values are "ipx" for the IPX family, "at" for the Appletalk family, and "", the empty string, for the IP address family.
s_pcSrvPart	The remainder of the URL, after the host identification.
	The host and port should be sufficient to open a socket to the machine hosting the service; the remainder of the URL should allow further differentiation of the service.

The SLPHandle

```
typedef void* SLPHandle;
```

The SLPHandle type is returned by SLPOpen() and is a parameter to all SLP functions. It serves as a handle for all resources allocated on behalf of the process by the SLP library. The type is opaque.

Callbacks
Include a function pointer to a callback function specific to a particular API operation in the parameter list when the API function is invoked. The callback function is called with the results of the operation in both the synchronous and asynchronous cases. When the callback function is invoked, the memory included in the callback parameters is owned by the API library, and the client code in the callback must copy out the contents if it wants to maintain the information longer than the duration of the current callback call.

Each callback parameter list contains parameters for reporting the results of the operation, as well as an error code parameter and a cookie parameter. The error code parameter reports the error status of the ongoing (for asynchronous) or completed (for synchronous) operation. The cookie parameter allows the client code that starts the operation by invoking the API function to pass information down to the callback without using global variables. The callback returns an SLPBoolean to indicate whether the API library should continue processing the operation. If the value returned from the callback is SLP_TRUE, asynchronous operations are terminated. Synchronous operations ignore the return since the operation is already complete.

SLPRegReport()

```
typedef void SLPRegReport(SLPHandle hSLP,
    SLPError errCode,
    void *pvCookie);
```

SLPRegReport() is the callback function to the SLPReg(), SLPDereg(), and SLPDelAttrs() functions. The SLPRegReport() callback has the following parameters:

hSLP	The SLPHandle() used to initiate the operation.
errCode	An error code indicating if an error occurred during the operation.
pvCookie	Memory passed down from the client code that called the original API function, starting the operation. It may be NULL.

SLPSrvTypeCallback()

```
typedef SLPBoolean SLPSrvTypeCallback(SLPHandle hSLP,
    const char* pcSrvTypes,
    SLPError errCode,
    void *pvCookie);
```

The SLPSrvTypeCallback() type is the type of the callback function parameter to the SLPFindSrvTypes() function. The results are collated when the *hSLP* handle is opened either synchronously or asynchronously. The SLPSrvTypeCallback() callback has the following parameters:

hSLP	The SLPHandle used to initiate the operation.
pcSrvTypes	A character buffer containing a comma-separated, null-terminated list of service types.
errCode	An error code indicating if an error occurred during the operation. The callback should check this error code before processing the parameters. If the error code is other than SLP_OK, then the API library may choose to terminate the outstanding operation.
pvCookie	emory passed down from the client code that called the original API function, starting the operation. It can be NULL.

SLPSrvURLCallback

```
typedef SLPBoolean SLPSrvURLCallback(SLPHandle hSLP,
    const char* pcSrvURL,
    unsigned short usLifetime,
    SLPError errCode,
    void *pvCookie);
```

The SLPSrvURLCallback() type is the type of the callback function parameter to the SLPFindSrvs() function. The results are collated, regardless of whether the *hSLP* was opened collated or uncollated. The SLPSrvURLCallback() callback has the following parameters:

hSLP	The SLPHandle used to initiate the operation.
pcSrvURL	A character buffer containing the returned service URL.
usLifetime	An unsigned short giving the life time of the service advertisement. The value must be an unsigned integer less than or equal to SLP_LIFETIME_MAXIMUM.

errCode   An error code indicating if an error occurred during the operation. The
                callback should check this error code before processing the parameters. If the
                error code is other than `SLP_OK`, then the API library may choose to terminate
                the outstanding operation.

pvCookie   Memory passed down from the client code that called the original API
                function, starting the operation. It can be `NULL`.

SLPAttrCallback

```
typedef SLPBoolean SLPAttrCallback(SLPHandle hSLP,
    const char* pcAttrList,
    SLPError errCode,
    void *pvCookie);
```

The `SLPAttrCallback()` type is the type of the callback function parameter to the
`SLPFindAttrs()` function.

The behavior of the callback differs depending upon whether the attribute request was by URL
or by service type. If the `SLPFindAttrs()` operation was originally called with a URL, the
callback is called once, in addition to the last call, regardless of whether the handle was opened
asynchronously or synchronously. The *pcAttrList* parameter contains the requested attributes
as a comma-separated list. It is empty if no attributes match the original tag list.

If the `SLPFindAttrs()` operation was originally called with a service type, the value of
pcAttrList and the calling behavior depend upon whether the handle was opened
asynchronously or synchronously. If the handle was opened asynchronously, the callback is
called every time the API library has results from a remote agent. The *pcAttrList* parameter is
collated between calls, and contains a comma-separated list of the results from the agent that
immediately returned. If the handle was opened synchronously, the results are collated from
all returning agents, the callback is called once, and the *pcAttrList* parameter is set to the
collated result.

`SLPAttrCallback()` callback has the following parameters:

hSLP   The `SLPHandle` used to initiate the operation.

pcAttrList   A character buffer containing a comma-separated and null-terminated list of
                attribute id/value assignments, in SLP wire format.

errCode   An error code indicating if an error occurred during the operation. The callback
                should check this error code before processing the parameters. If the error code
                is other than `SLP_OK`, then the API library may choose to terminate the
                outstanding operation.

pvCookie   Memory passed down from the client code that called the original API function,
                starting the operation. It can be `NULL`.

Errors An interface that is part of the SLP API may return one of the following values.

SLP_LAST_CALL   The SLP_LAST_CALL code is passed to callback functions
                              when the API library has no more data for them and
                              therefore no further calls will be made to the callback on
                              the currently outstanding operation. The callback uses
                              this to signal the main body of the client code that no
                              more data will be forthcoming on the operation, so that
                              the main body of the client code can break out of data
                              collection loops. On the last call of a callback during both
                              a synchronous and asynchronous call, the error code
                              parameter has value SLP_LAST_CALL, and the other
                              parameters are all NULL. If no results are returned by an
                              API operation, then only one call is made, with the error
                              parameter set to SLP_LAST_CALL.

SLP_OK   The SLP_OK code indicates that the no error occurred
                              during the operation.

SLP_LANGUAGE_NOT_SUPPORTED   No DA or SA has service advertisement information in
                              the language requested, but at least one DA or SA might
                              have information for that service in another language.

SLP_PARSE_ERROR   The SLP message was rejected by a remote SLP agent. The
                              API returns this error only when no information was
                              retrieved, and at least one SA or DA indicated a protocol
                              error. The data supplied through the API may be
                              malformed or damaged in transit.

SLP_INVALID_REGISTRATION   The API may return this error if an attempt to register a
                              service was rejected by all DAs because of a malformed
                              URL or attributes.SLP does not return the error if at least
                              one DA accepts the registration.

SLP_SCOPE_NOT_SUPPORTED   The API returns this error if the UA or SA has been
                              configured with the net.slp.useScopes list of scopes
                              and the SA request did not specify one or more of these
                              allowable scopes, and no others. It may also be returned
                              by a DA if the scope included in a request is not supported
                              by a DA.

SLP_AUTHENTICATION_ABSENT   This error arises when the UA or SA failed to send an
                              authenticator for requests or registrations when security
                              is enabled and thus required.

SLP_AUTHENTICATION_FAILED   This error arises when a authentication on an SLP
                              message received from a remote SLP agent failed.

SLP_INVALID_UPDATE	An update for a nonexisting registration was issued, or the update includes a service type or scope different than that in the initial registration.
SLP_REFRESH_REJECTED	The SA attempted to refresh a registration more frequently than the minimum refresh interval. The SA should call the appropriate API function to obtain the minimum refresh interval to use.
SLP_NOT_IMPLEMENTED	An outgoing request overflowed the maximum network MTU size. The request should be reduced in size or broken into pieces and tried again.
SLP_BUFFER_OVERFLOW	An outgoing request overflowed the maximum network MTU size. The request should be reduced in size or broken into pieces and tried again.
SLP_NETWORK_TIMED_OUT	When no reply can be obtained in the time specified by the configured timeout interval, this error is returned.
SLP_NETWORK_INIT_FAILED	If the network cannot initialize properly, this error is returned.
SLP_MEMORY_ALLOC_FAILED	If the API fails to allocate memory, the operation is aborted and returns this.
SLP_PARAMETER_BAD	If a parameter passed into an interface is bad, this error is returned.
SLP_NETWORK_ERROR	The failure of networking during normal operations causes this error to be returned.
SLP_INTERNAL_SYSTEM_ERROR	A basic failure of the API causes this error to be returned. This occurs when a system call or library fails. The operation could not recover.
SLP_HANDLE_IN_USE	In the C API, callback functions are not permitted to recursively call into the API on the same SLPHandle, either directly or indirectly. If an attempt is made to do so, this error is returned from the called API function

List Of Routines

SLPOpen()	open an SLP handle
SLPClose()	close an open SLP handle
SLPReg()	register a service advertisement
SLPDereg()	deregister a service advertisement
SLPDelAttrs()	delete attributes

SLPFindSrvTypes()	return service types
SLPFindSrvs()	return service URLs
SLPFindAttrs()	return service attributes
SLPGetRefreshInterval()	return the maximum allowed refresh interval for SAs
SLPFindScopes()	return list of configured and discovered scopes
SLPParseSrvURL()	parse service URL
SLPEscape()	escape special characters
SLPUnescape()	translate escaped characters into UTF-8
SLPGetProperty()	return SLP configuration property
SLPSetProperty()	set an SLP configuration property
slp_strerror()	map SLP error code to message
SLPFree()	free memory

Environment Variables When SLP_CONF_FILE is set, use this file for configuration.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp
CSI	CSI-enabled
MT-Level	Safe

See Also slpd(1M), slp.conf(4), slpd.reg(4), attributes(5)

Oracle Solaris Administration: Network Services

Guttman, E., Perkins, C., Veizades, J., and Day, M. *RFC 2608, Service Location Protocol, Version 2*. The Internet Society. June 1999.

Kempf, J. and Guttman, E. *RFC 2614, An API for Service Location*. The Internet Society. June 1999.

Name SLPClose – close an open SLP handle

Synopsis cc [*flag*...] *file*... -lslp [*library*...]
#include <slp.h>

void SLPClose(SLPHandle *phSLP*);

Description The SLPClose() function frees all resources associated with the handle. If the handle is invalid, the function returns silently. Any outstanding synchronous or asynchronous operations are cancelled, so that their callback functions will not be called any further

Parameters *phSLP*   An SLPHandle handle returned from a call to SPLOpen().

Errors This function or its callback may return any SLP error code. See the ERRORS section in slp_api(3SLP).

Examples EXAMPLE 1   Using SLPClose()

The following example will free all resources associated the handle:

SLPHandle hslp
    SLPCLose(hslp);

Environment SLP_CONF_FILE   When set, use this file for configuration.
Variables

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp

See Also slpd(1M), slp_api(3SLP), slp.conf(4), slpd.reg(4), attributes(5)

Oracle Solaris Administration: Network Services

Kempf, J. and Guttman, E. *RFC 2614, An API for Service Location*. The Internet Society. June 1999.

Name   SLPDelAttrs – delete attributes

Synopsis   cc [*flag*...] *file*... -lslp [*library*...]
#include <slp.h>

SLPError SLPDelAttrs(SLPHandle *hSLP*, const char *pcURL*,
     const char *pcAttrs*, SLPRegReport *callback*, void *pvCookie*);

Description   The SLPDelAttrs() function deletes the selected attributes in the locale of the SLPHandle. If
no error occurs, the return value is 0. Otherwise, one of the SLPError codes is returned.

Parameters   *hSLP*   The language specific SLPHandle to use to delete attributes. It cannot be NULL.

pcURL   The URL of the advertisement from which the attributes should be deleted. It
              cannot be NULL.

pcAttrs   A comma-separated list of attribute ids for the attributes to deregister.

callback   A callback to report the operation's completion status. It cannot be NULL.

pvCookie   Memory passed to the callback code from the client. It cannot be NULL.

Errors   This function or its callback may return any SLP error code. See the ERRORS section in
slp_api(3SLP).

Examples   EXAMPLE 1   Deleting Attributes

Use the following example to delete the location and dpi attributes for the URL
service:printer:lpr://serv/queve1

```
SLPHandle hSLP;
SLPError err;
SLPRegReport report;

err = SLPDelAttrs(hSLP, "service:printer:lpr://serv/queue1",
    "location,dpi", report, NULL);
```

Environment   SLP_CONF_FILE   When set, use this file for configuration.
Variables
Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp

See Also   slpd(1M), slp_api(3SLP), slp.conf(4), slpd.reg(4), attributes(5)

Oracle Solaris Administration: Network Services

Kempf, J. and Guttman, E. *RFC 2614, An API for Service Location*. The Internet Society. June
1999.

Name SLPDereg – deregister the SLP advertisement

Synopsis cc [*flag...*] *file...* -lslp [*library...*]
#include <slp.h>

SLPError SLPDereg(SLPHandle *hSLP*, const char **pcURL*,
    SLPRegReport *callback*, void **pvCookie*);

Description The SLPDereg() function deregisters the advertisement for URL *pcURL* in all scopes where the service is registered and in all language locales, not just the locale of the SLPHandle. If no error occurs, the return value is 0. Otherwise, one of the SLPError codes is returned.

Parameters *hSLP* The language specific SLPHandle to use for deregistering. *hSLP* cannot be NULL.

pcURL The URLto deregister. The value of *pcURL* cannot be NULL.

callback A callback to report the operation completion status. *callback* cannot be NULL.

pvCookie Memory passed to the callback code from the client. *pvCookie* can be NULL.

Errors This function or its callback may return any SLP error code. See the ERRORS section in slp_api(3SLP).

Examples EXAMPLE 1 Using SLPDereg()

Use the following example to deregister the advertisement for the URL "service:ftp://csserver":

```
SLPerror err;
SLPHandle hSLP;
SLPRegReport regreport;

err = SLPDereg(hSLP, "service:ftp://csserver", regreport, NULL);
```

Environment Variables SLP_CONF_FILE   When set, use this file for configuration.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp

See Also slpd(1M), slp_api(3SLP), slp.conf(4), slpd.reg(4), attributes(5)

Oracle Solaris Administration: Network Services

Guttman, E., Perkins, C., Veizades, J., and Day, M. *RFC 2608, Service Location Protocol, Version 2*. The Internet Society. June 1999.

Kempf, J. and Guttman, E., *RFC 2614, An API for Service Location*, The Internet Society, June 1999.

Name SLPEscape – escapes SLP reserved characters

Synopsis cc [*flag...*] *file...* -lslp [*library...*]
#include <slp.h>

SLPError SLPEscape(const char *pcInBuf, char** *ppcOutBuf*,
    SLPBoolean *isTag*);

Description The SLPEscape() function processes the input string in *pcInbuf* and escapes any SLP reserved characters. If the *isTag* parameter is SLPTrue, it then looks for bad tag characters and signals an error if any are found by returning the SLP_PARSE_ERROR code. The results are put into a buffer allocated by the API library and returned in the *ppcOutBuf* parameter. This buffer should be deallocated using SLPFree(3SLP) when the memory is no longer needed.

Parameters *pcInBuf*   Pointer to the input buffer to process for escape characters.

ppcOutBuf   Pointer to a pointer for the output buffer with the SLP reserved characters escaped. It must be freed using SLPFree() when the memory is no longer needed.

isTag   When true, checks the input buffer for bad tag characters.

Errors This function or its callback may return any SLP error code. See the ERRORS section in slp_api(3SLP).

Examples EXAMPLE 1   Converting Attribute Tags

The following example shows how to convert the attribute tag ,tag-example, to on the wire format:

```
SLPError err;
char* escaped Chars;

err = SLPEscape(",tag-example,", &escapedChars, SLP_TRUE);
```

Environment SLP_CONF_FILE   When set, use this file for configuration.
Variables
Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp

See Also slpd(1M), slp_api(3SLP), SLPFree(3SLP), slp.conf(4), slpd.reg(4), attributes(5)

Oracle Solaris Administration: Network Services

Guttman, E., Perkins, C., Veizades, J., and Day, M. *RFC 2608, Service Location Protocol, Version 2*. The Internet Society. June 1999.

Kempf, J. and Guttman, E. *RFC 2614, An API for Service Location*. The Internet Society. June 1999.

Name SLPFindAttrs – return service attributes

Synopsis cc [*flag...*] *file...* -lslp [*library...*]
```
#include <slp.h>

SLPError SLPFindAttrs(SLPHandle hSLP, const char *pcURL,
    const char *pcScopeList, const char *pcAttrIds,
    SLPAttrCallback *callback, void *pvCookie);
```

Description The SLPFindAttrs() function returns service attributes matching the attribute tags for the indicated full or partial URL. If *pcURL* is a complete URL, the attribute information returned is for that particular service in the language locale of the SLPHandle. If *pcURL* is a service type, then all attributes for the service type are returned, regardless of the language of registration. Results are returned through the *callback* parameter.

The result is filtered with an SLP attribute request filter string parameter, the syntax of which is described in *RFC 2608*. If the filter string is the empty string, "", all attributes are returned.

If an error occurs in starting the operation, one of the SLPError codes is returned.

Parameters *hSLP*   The language-specific SLPHandle on which to search for attributes. It cannot be NULL.

pcURL   The full or partial URL. See *RFC 2608* for partial URL syntax. It cannot be NULL.

pcScopeList   A pointer to a char containing a comma-separated list of scope names. It cannot be NULL or an empty string, "".

pcAttrIds   The filter string indicating which attribute values to return. Use empty string "" to indicate all values. Wildcards matching all attribute ids having a particular prefix or suffix are also possible. It cannot be NULL.

callback   A callback function through which the results of the operation are reported. It cannot be NULL.

pvCookie   Memory passed to the callback code from the client. It may be NULL.

Errors This function or its callback may return any SLP error code. See the ERRORS section in slp_api(3SLP).

Examples EXAMPLE 1   Returning Service Attributes for a Specific URL

Use the following example to return the attributes "location" and "dpi" for the URL "service:printer:lpr://serv/queue1" through the callback attrReturn:

```
SLPHandle hSLP;
SLPAttrCallback attrReturn;
SLPError err;

err = SLPFindAttrs(hSLP "service:printer:lpr://serv/queue1",
```

EXAMPLE 1 Returning Service Attributes for a Specific URL *(Continued)*

```
"default", "location,dpi", attrReturn, err);
```

EXAMPLE 2 Returning Service Attributes for All URLs of a Specific Type

Use the following example to return the attributes "location" and "dpi" for all service URLs having type "service:printer:lpr":

```
err = SLPFindAttrs(hSLP, "service:printer:lpr",
    "default", "location, pi",
    attrReturn, NULL);
```

Environment Variables SLP_CONF_FILE When set, use this file for configuration.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp

See Also slpd(1M), slp_api(3SLP), slp.conf(4), slpd.reg(4), attributes(5)

Oracle Solaris Administration: Network Services

Kempf, J. and Guttman, E. *RFC 2614, An API for Service Location*. The Internet Society. June 1999.

Name SLPFindScopes – return list of configured and discovered scopes

Synopsis cc [*flag*...] *file*... -lslp [*library*...]
#include <slp.h>

SLPError SLPFindScopes(SLPHandle *hSLP*, char** *ppcScopes*);

Description The SLPFindScopes() function sets the *ppcScopes* parameter to a pointer to a comma-separated list including all available scope names. The list of scopes comes from a variety of sources: the configuration file, the net.slp.useScopes property and the net.slp.DAAddresses property, DHCP, or through the DA discovery process. If there is any order to the scopes, preferred scopes are listed before less desirable scopes. There is always at least one string in the array, the default scope, DEFAULT.

If no error occurs, SLPFindScopes() returns SLP_OK, otherwise, it returns the appropriate error code.

Parameters *hSLP* The SLPHandle on which to search for scopes. *hSLP* cannot be NULL.

ppcScopes A pointer to a char pointer into which the buffer pointer is placed upon return. The buffer is null-terminated. The memory should be freed by calling SLPFree(). See SLPFree(3SLP)

Errors This function or its callback may return any SLP error code. See the ERRORS section in slp_api(3SLP).

Examples EXAMPLE 1 Finding Configured or Discovered Scopes

Use the following example to find configured or discovered scopes:

```
SLPHandle hSLP;
char *ppcScopes;
SLPError err;

error = SLPFindScopes(hSLP, & ppcScopes);
```

Environment Variables SLP_CONF_FILE When set, use this file for configuration.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp

See Also slpd(1M), slp_api(3SLP), SLPFree(3SLP), slp.conf(4), slpd.reg(4), attributes(5)

Oracle Solaris Administration: Network Services

Guttman, E., Perkins, C., Veizades, J., and Day, M. *RFC 2608, Service Location Protocol, Version 2*. The Internet Society. June 1999.

Kempf, J. and Guttman, E. *RFC 2614, An API for Service Location*. The Internet Society. June 1999.

Name SLPFindSrvs – return service URLs

Synopsis cc [*flag*...] *file*... -lslp [*library*...]
#include <slp.h>

SLPError SLPFindSrvs(SLPHandle *hSLP*, const char **pcServiceType*,
    const char **pcScopeList*, const char **pcSearchFilter*,
    SLPSrvURLCallback **callback*, void **pvCookie*);

Description The SLPFindSrvs() function issues a request for SLP services. The query is for services on a language-specific SLPHandle. It returns the results through the *callback*. The parameters will determine the results.

If an error occurs in starting the operation, one of the SLPError codes is returned.

Parameters *hSLP*   The language-specific SLPHandle on which to search for services. It cannot be NULL.

pcServiceType   The service type string for the request. The *pcServiceType* can be discovered by a call to SLPSrvTypes(). Examples of service type strings include

"service:printer:lpr"

or

"service:nfs"

pcServiceType cannot be NULL.

pcScopeList   A pointer to a char containing a comma-separated list of scope names. It cannot be NULL or an empty string, "".

pcSearchFilter A query formulated of attribute pattern matching expressions in the form of a LDAPv3 search filter. See *RFC 2254*. If this filter is empty, "", all services of the requested type in the specified scopes are returned. It cannot be NULL.

callback   A callback through which the results of the operation are reported. It cannot be NULL.

pvCookie   Memory passed to the callback code from the client. It can be NULL.

Errors This function or its callback may return any SLP error code. See the ERRORS section in slp_api(3SLP).

Examples EXAMPLE 1 Using SLPFindSrvs()

The following example finds all advertisements for printers supporting the LPR protocol with the dpi attribute 300 in the default scope:

```
SLPError err;
SLPHandle hSLP;
SLPSrvURLCallback srvngst;
```

EXAMPLE 1 Using SLPFindSrvs()   *(Continued)*

```
err = SLPFindSrvs(hSLP,
                  "service:printer:lpr",
                  "default",
                  "(dpi=300)",
                  srvngst,
                  NULL);
```

Environment Variables   SLP_CONF_FILE   When set, use this file for configuration.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp

See Also   slpd(1M), slp_api(3SLP), slp.conf(4), slpd.reg(4), attributes(5)

Oracle Solaris Administration: Network Services

Howes, T. *RFC 2254, The String Representation of LDAP Search Filters*. The Internet Society. 1997.

Guttman, E., Perkins, C., Veizades, J., and Day, M. *RFC 2608, Service Location Protocol, Version 2*. The Internet Society. June 1999.

Kempf, J. and Guttman, E. *RFC 2614, An API for Service Location*. The Internet Society. June 1999.

Name SLPFindSrvTypes – find service types

Synopsis cc [*flag...*] *file...* -lslp [*library...*]
#include <slp.h>

SLPError SLPFindSrvTypes(SLPHandle *hSLP*, const char **pcNamingAuthority*,
    const char **pcScopeList*, SLPSrvTypeCallback **callback*, void **pvCookie*);

Description The SLPFindSrvTypes() function issues an SLP service type request for service types in the
scopes indicated by the pcScopeList. The results are returned through the *callback*
parameter. The service types are independent of language locale, but only for services
registered in one of the scopes and for the indicated naming authority.

If the naming authority is "*", then results are returned for all naming authorities. If the
naming authority is the empty string, "", then the default naming authority, IANA, is used.
IANA is not a valid naming authority name. The SLP_PARAMETER_BAD error code will be
returned if you include it explicitly.

The service type names are returned with the naming authority included in the following
format:

service-type "." naming-authority

unless the naming authority is the default, in which case, just the service type name is
returned.

If an error occurs in starting the operation, one of the SLPError codes is returned.

Parameters

hSLP	The SLPHandle on which to search for types. It cannot be NULL.
pcNamingAuthority	The naming authority to search. Use "*"to search all naming authorties; use the empty string "" to search the default naming authority. It cannot be NULL.
pcScopeList	A pointer to a char containing a comma-separated list of scope names to search for service types. It cannot be NULL or an empty string, "".
callback	A callback through which the results of the operation are reported. It cannot be NULL.
pvCookie	Memory passed to the callback code from the client. It can be NULL.

Errors This function or its callback may return any SLP error code. See the ERRORS section in
slp_api(3SLP).

Examples EXAMPLE 1 Using SLPFindSrvTypes()

The following example finds all service type names in the default scope and default naming
authority:

EXAMPLE 1   Using SLPFindSrvTypes()   *(Continued)*

```
SLPError err;
SLPHandle hSLP;
SLPSrvTypeCallback findsrvtypes;

err = SLPFindSrvTypes(hSLP, "", "default", findsrvtypes, NULL);
```

Environment Variables   SLP_CONF_FILE   When set, use this file for configuration.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp

See Also   slpd(1M), slp_api(3SLP), slp.conf(4), slpd.reg(4), attributes(5)

Oracle Solaris Administration: Network Services

Guttman, E., Perkins, C., Veizades, J., and Day, M. *RFC 2608, Service Location Protocol, Version 2*. The Internet Society. June 1999.

Howes, T. *RFC 2254, The String Representation of LDAP Search Filters*. The Internet Society. 1997.

Kempf, J. and Guttman, E. *RFC 2614, An API for Service Location*. The Internet Society. June 1999.

Name   SLPFree – frees memory

Synopsis   cc [*flag*...] *file*... -lslp [*library*...]
          #include <slp.h>

          SLPError SLPFree(void **pvMem*);

Description   The SLPFree() function frees memory returned from SLPParseSrvURL(), SLPFindScopes(),
          SLPEscape(), and SLPUnescape().

Parameters   *pvMem*   A pointer to the storage allocated by the SLPParseSrvURL(), SLPFindScopes(),
                      SLPEscape(), and SLPUnescape() functions. *pvMem* is ignored if its value is NULL.

Errors   This function or its callback may return any SLP error code. See the ERRORS section in
          slp_api(3SLP).

Examples   EXAMPLE 1   Using SLPFree()

          The following example illustrates how to call SLPFree(). It assumes that SrvURL contains
          previously allocated memory.

          SLPerror err;

          err = SLPFree((void*) SrvURL);

Environment   SLP_CONF_FILE   When set, use this file for configuration.
Variables
Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp

See Also   slpd(1M), SLPEscape(3SLP), SLPFindScopes(3SLP), SLPParseSrvURL(3SLP),
          SLPUnescape(3SLP), slp_api(3SLP), slp.conf(4), slpd.reg(4), attributes(5)

          Oracle Solaris Administration: Network Services

          Guttman, E., Perkins, C., Veizades, J., and Day, M. *RFC 2608, Service Location Protocol,
          Version 2*. The Internet Society. June 1999.

          Kempf, J. and Guttman, E. *RFC 2614, An API for Service Location*. The Internet Society. June
          1999.

Name   SLPGetProperty – return SLP configuration property

Synopsis cc [*flag...*] *file...* -lslp [*library...*]
#include <slp.h>

const char* SLPGetProperty(const char* *pcName*);

Description The SLPGetProperty() function returns the value of the corresponding SLP property name, or NULL, if none. If there is no error, SLPGetProperty() returns a pointer to the property value. If the property was not set, it returns the empty string, "". If an error occurs, SLPGetProperty() returns NULL. The returned string should not be freed.

Parameters *pcName*   A null-terminated string with the property name. *pcName* cannot be NULL.

Errors This function or its callback may return any SLP error code. See the ERRORS section in slp_api(3SLP).

Examples EXAMPLE 1   Using SLPGetProperty()

Use the following example to return a list of configured scopes:

```
const char* useScopes

useScopes = SLPGetProperty("net.slp.useScopes");
```

Environment SLP_CONF_FILE   When set, use this file for configuration.
Variables
Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp

See Also slpd(1M), slp_api(3SLP), slp.conf(4), slpd.reg(4), attributes(5)

Oracle Solaris Administration: Network Services

Kempf, J. and Guttman, E. *RFC 2614, An API for Service Location*. The Internet Society. June 1999.

Name SLPGetRefreshInterval – return the maximum allowed refresh interval

Synopsis cc [*flag*...] *file*... -lslp [*library*...]#include <slp.h>

int SLPGetRefreshInterval(void)

Description The SLPGetRefreshInterval() function returns the maximum across all DAs of the min-refresh-interval attribute. This value satisfies the advertised refresh interval bounds for all DAs. If this value is used by the SA, it assures that no refresh registration will be rejected. If no DA advertises a min-refresh-interval attribute, a value of 0 is returned. If an error occurs, an SLP error code is returned.

Errors This function or its callback may return any SLP error code. See the ERRORS section in slp_api(3SLP).

Examples EXAMPLE 1 Using SLPGetRefreshInterval()

Use the following example to return the maximum valid refresh interval for SA:

int minrefresh

minrefresh = SLPGetRefreshInterval( );

Environment Variables SLP_CONF_FILE   When set, use this file for configuration.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp

See Also slpd(1M), slp_api(3SLP), slp.conf(4), slpd.reg(4), attributes(5)

Oracle Solaris Administration: Network Services

Kempf, J. and Guttman, E. *RFC 2614, An API for Service Location*. The Internet Society. June 1999.

Name SLPOpen – open an SLP handle

Synopsis cc [*flag...*] *file...* -lslp [*library...*]
#include <slp.h>

SLPError SLPOpen(const char *pcLang*, SLPBoolean *isAsync*, SLPHandle *phSLP*);

Description The SLPOpen() function returns a SLPHandle handle in the *phSLP* parameter for the language locale passed in as the *pcLang* parameter. The client indicates if operations on the handle are to be synchronous or asynchronous through the *isAsync* parameter. The handle encapsulates the language locale for SLP requests issued through the handle, and any other resources required by the implementation. SLP properties are not encapsulated by the handle, they are global. The return value of the function is an SLPError code indicating the status of the operation. Upon failure, the *phSLP* parameter is NULL.

An SLPHandle can only be used for one SLP API operation at a time. If the original operation was started asynchronously, any attempt to start an additional operation on the handle while the original operation is pending results in the return of an SLP_HANDLE_IN_USE error from the API function. The SLPClose() function terminates any outstanding calls on the handle.

Parameters *pcLang* A pointer to an array of characters containing the language tag set forth in *RFC 1766* for the natural language locale of requests issued on the handle. This parameter cannot be NULL.

isAsync An SLPBoolean indicating whether or not the SLPHandle should be opened for an asynchronous operation.

phSLP A pointer to an SLPHandle in which the open SLPHandle is returned. If an error occurs, the value upon return is NULL.

Errors This function or its callback may return any SLP error code. See the ERRORS section in slp_api(3SLP).

Examples EXAMPLE 1 Using SLPOpen()

Use the following example to open a synchronous handle for the German ("de") locale:

SLPHandle HSLP; SLPError err; err = SLPOpen("de", SLP_FALSE, &hSLP)

Environment Variables SLP_CONF_FILE When set, use this file for configuration.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp

See Also slpd(1M), slp_api(3SLP), slp.conf(4), slpd.reg(4), attributes(5)

Oracle Solaris Administration: Network Services

Alvestrand, H. *RFC 1766, Tags for the Identification of Languages.* Network Working Group. March 1995.

Kempf, J. and Guttman, E. *RFC 2614, An API for Service Location.* The Internet Society. June 1999.

Name SLPParseSrvURL – parse service URL

Synopsis cc [*flag...*] *file...* -lslp [*library...*]
#include <slp.h>

SLPError SLPParseSrvURL(const char *_pcSrvURL_, SLPSrvURL** _ppSrvURL_);

Description The SLPParseSrvURL() routine parses the URL passed in as the argument into a service URL structure and returns it in the *ppSrvURL* pointer. If a parser error occurs, returns SLP_PARSE_ERROR. The structure returned in *ppSrvURL* should be freed with SLPFree(). If the URL has no service part, the s_pcSrvPart string is the empty string, "", that is, it is not NULL. If *pcSrvURL* is not a service: URL, then the s_pcSrvType field in the returned data structure is the URL's scheme, which might not be the same as the service type under which the URL was registered. If the transport is IP, the s_pcNetFamily field is the empty string.

If no error occurs, the return value is the SLP_OK. Otherwise, if an error occurs, one of the SLPError codes is returned.

Parameters *pcSrvURL*   A pointer to a character buffer containing the null terminated URL string to parse. It is destructively modified to produce the output structure. It may not be NULL.

ppSrvURL   A pointer to a ponter for the SLPSrvURL structure to receive the parsed URL. It may not be NULL.

Errors This function or its callback may return any SLP error code. See the ERRORS section in slp_api(3SLP).

Examples EXAMPLE 1   Using SLPParseSrvURL()

The following example uses the SLPParseSrvURL() function to parse the service URL service:printer:lpr://serv/queue1:

```
SLPSrvURL* surl;
SLPError err;

err = SLPParseSrvURL("service:printer:lpr://serv/queue1", &surl);
```

Environment Variables SLP_CONF_FILE   When set, use this file for configuration.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp

See Also slpd(1M), slp_api(3SLP), slp.conf(4), slpd.reg(4), attributes(5)

Oracle Solaris Administration: Network Services

Guttman, E., Perkins, C., Veizades, J., and Day, M. *RFC 2608, Service Location Protocol, Version 2*. The Internet Society. June 1999.

Kempf, J. and Guttman, E. *RFC 2614, An API for Service Location*. The Internet Society. June 1999.

Name   SLPReg – register an SLP advertisement

Synopsis   cc [*flag...*] *file...* -lslp [*library...*]
#include <slp.h>

SLPError SLPReg(SLPHandle *hSLP*, const char **pcSrvURL*,
    const unsigned short *usLifetime*, const char **pcSrvType*,
    const char **pcAttrs*, SLPBoolean *fresh*,
    SLPRegReport *callback*, void **pvCookie*);

Description   The SLPReg() function registers the URL in *pcSrvURL* having the lifetime *usLifetime* with the attribute list in *pcAttrs*. The *pcAttrs* list is a comma-separated list of attribute assignments in on-the-wire format (including escaping of reserved characters). The *sLifetime* parameter must be nonzero and less than or equal to SLP_LIFETIME_MAXIMUM. If the fresh flag is SLP_TRUE, then the registration is new, the SLP protocol *fresh* flag is set, and the registration replaces any existing registrations.

The *pcSrvType* parameter is a service type name and can be included for service URLs that are not in the service: scheme. If the URL is in the service: scheme, the *pcSrvType* parameter is ignored. If the fresh flag is SLP_FALSE, then an existing registration is updated. Rules for new and updated registrations, and the format for *pcAttrs* and *pcScopeList*, can be found in *RFC 2608*. Registrations and updates take place in the language locale of the *hSLP* handle.

The API library is required to perform the operation in all scopes obtained through configuration.

Parameters   *hSLP*   The language specific SLPHandle on which to register the advertisement. *hSLP* cannot be NULL.

pcSrvURL   The URL to register. The value of *pcSrvURL* cannot be NULL or the empty string.

usLifetime   An unsigned short giving the life time of the service advertisement, in seconds. The value must be an unsigned integer less than or equal to SLP_LIFETIME_MAXIMUM.

pcSrvType   The service type. If *pURL* is a service: URL, then this parameter is ignored. *pcSrvType* cannot be NULL.

pcAttrs   A comma-separated list of attribute assignment expressions for the attributes of the advertisement. *pcAttrs* cannot be NULL. Use the empty string, "", to indicate no attributes.

fresh   An SLPBoolean that is SLP_TRUE if the registration is new or SLP_FALSE if it is a reregistration.

callback   A callback to report the operation completion status. *callback* cannot be NULL.

pvCookie   Memory passed to the callback code from the client. *pvCookie* can be NULL.

Errors This function or its callback may return any SLP error code. See the ERRORS section in
slp_api(3SLP).

Examples EXAMPLE 1   An Initial Registration

The following example shows an initial registration for the "service:video://bldg15"
camera service for three hours:

```
SLPError err;
SLPHandle hSLP;
SLPRegReport regreport;
err = SLPReg(hSLP, "service:video://bldg15",
    10800, "", "(location=B15-corridor),
    (scan-rate=100)", SLP_TRUE,
    regRpt, NULL);
```

Environment SLP_CONF_FILE   When set, use this file for configuration.
Variables

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp

See Also slpd(1M), slp_api(3SLP), slp.conf(4), slpd.reg(4), attributes(5)

Oracle Solaris Administration: Network Services

Guttman, E., Perkins, C., Veizades, J., and Day, M.,*RFC 2608, Service Location Protocol,
Version 2*. The Internet Society. June 1999.

Kempf, J. and Guttman, E. *RFC 2614, An API for Service Location*. The Internet Society. June
1999.

Name SLPSetProperty – set an SLP configuration property

Synopsis cc [*flag*...] *file*... -lslp [*library*...]
#include <slp.h>

void SLPSetProperty(const char *pcName, const char *pcValue);

Description The SLPSetProperty() function sets the value of the SLP property to the new value. The *pcValue* parameter contains the property value as a string.

Parameters *pcName*   A null-terminated string with the property name. *pcName* cannot be NULL.

pcValue   A null-terminated string with the property value. *pcValue* cannot be NULL

Errors This function or its callback may return any SLP error code. See the ERRORS section in slp_api(3SLP).

Examples EXAMPLE 1   Setting a Configuration Property

The following example shows to set the property net.slp.typeHint to service:ftp:

SLPSetProperty ("net.slp.typeHint" "service:ftp");

Environment Variables SLP_CONF_FILE   When set, use this file for configuration.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp

See Also slpd(1M), slp_api(3SLP), slp.conf(4), slpd.reg(4), attributes(5)

Oracle Solaris Administration: Network Services

Kempf, J. and Guttman, E. *RFC 2614, An API for Service Location*. The Internet Society. June 1999.

Name slp_strerror – map SLP error codes to messages

Synopsis #include <slp.h>

const char* slp_strerror(SLPError *err_code*);

Description The slp_strerror() function maps err_code to a string explanation of the error. The returned string is owned by the library and must not be freed.

Parameters *err_code* An SLP error code.

Errors This function or its callback may return any SLP error code. See the ERRORS section in slp_api(3SLP).

Examples EXAMPLE 1 Using slp_sterror()

The following example returns the message that corresponds to the error code:

```
SLPError error;
const char* msg;
msg = slp_streerror(err);
```

Environment Variables SLP_CONF_FILE When set, use this file for configuration.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp

See Also slpd(1M), slp_api(3SLP), slp.conf(4), slpd.reg(4), attributes(5)

Oracle Solaris Administration: Network Services

Kempf, J. and Guttman, E. *RFC 2614, An API for Service Location*. The Internet Society. June 1999.

Name SLPUnescape – translate escaped characters into UTF-8

Synopsis cc [*flag*...] *file*... -lslp [*library*...]
#include <slp.h>

```
SLPError SLPUnescape(const char *pcInBuf, char** ppcOutBuf,
    SLPBoolean isTag);
```

Description The SLPUnescape() function processes the input string in *pcInbuf* and unescapes any SLP
reserved characters. If the *isTag* parameter is SLPTrue, then look for bad tag characters and
signal an error if any are found with the SLP_PARSE_ERROR code. No transformation is
performed if the input string is an opaque. The results are put into a buffer allocated by the
API library and returned in the *ppcOutBuf* parameter. This buffer should be deallocated using
SLPFree(3SLP) when the memory is no longer needed.

Parameters *pcInBuf*   Pointer to the input buffer to process for escape characters.

ppcOutBuf   Pointer to a pointer for the output buffer with the SLP reserved characters
escaped. Must be freed using SLPFree(3SLP) when the memory is no longer
needed.

isTag   When true, the input buffer is checked for bad tag characters.

Errors This function or its callback may return any SLP error code. See the ERRORS section in
slp_api(3SLP).

Examples EXAMPLE 1 Using SLPUnescape()

The following example decodes the representation for ",tag,":

```
char* pcOutBuf;
SLPError err;

err = SLPUnescape("\\2c tag\\2c", &pcOutbuf, SLP_TRUE);
```

Environment   SLP_CONF_FILE   When set, use this file for configuration.
Variables
Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	service/network/slp

See Also slpd(1M), SLPFree(3SLP), slp_api(3SLP), slp.conf(4), slpd.reg(4), attributes(5)

Oracle Solaris Administration: Network Services

Guttman, E., Perkins, C., Veizades, J., and Day, M. *RFC 2608, Service Location Protocol,
Version 2*. The Internet Society. June 1999.

Kempf, J. and Guttman, E. *RFC 2614, An API for Service Location*. The Internet Society. June 1999.

Name sockatmark – determine whether a socket is at the out-of-band mark

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <sys/socket.h>

int sockatmark(int *s*);

Description The sockatmark() function determines whether the socket specified by the descriptor *s* is at the out-of-band data mark. If the protocol for the socket supports out-of-band data by marking the stream with an out-of-band data mark, the sockatmark() function returns 1 when all data preceding the mark has been read and the out-of-band data mark is the first element in the receive queue. The sockatmark() function does not remove the mark from the stream.

Return Values Upon successful completion, the sockatmark() function returns a value indicating whether the socket is at an out-of-band data mark. If the protocol has marked the data stream and all data preceding the mark has been read, the return value is 1. If there is no mark, or if data precedes the mark in the receive queue, the sockatmark() function returns 0. Otherwise, it returns −1 and sets errno to indicate the error.

Errors The sockatmark() function will fail if:

EBADF   The *s* argument is not a valid file descriptor.

ENOTTY   The *s* argument does not specify a descriptor for a socket.

Usage The use of this function between receive operations allows an application to determine which received data precedes the out-of-band data and which follows the out-of-band data.

There is an inherent race condition in the use of this function. On an empty receive queue, the current read of the location might well be at the "mark", but the system has no way of knowing that the next data segment that will arrive from the network will carry the mark, and sockatmark() will return false, and the next read operation will silently consume the mark.

Hence, this function can only be used reliably when the application already knows that the out-of-band data has been seen by the system or that it is known that there is data waiting to be read at the socket, either by SIGURG or select(3C).

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See standards(5).

See Also   recv(3XNET), recvmsg(3XNET), select(3C), attributes(5), standards(5)

Name   socket – create an endpoint for communication

Synopsis   cc [*flag* ...] *file* ... -lsocket -lnsl [*library* ...]
#include <sys/types.h>
#include <sys/socket.h>

int socket(int *domain*, int *type*, int *protocol*);

Description   The socket() function creates an endpoint for communication and returns a descriptor.

The *domain* argument specifies the protocol family within which communication takes place.
The protocol family is generally the same as the address family for the addresses supplied in
later operations on the socket. These families are defined in <sys/socket.h>.

The currently supported protocol families are:

PF_UNIX   UNIX system internal protocols

PF_INET   Internet Protocol Version 4 (IPv4)

PF_INET6   Internet Protocol Version 6 (IPv6)

PF_NCA   Network Cache and Accelerator (NCA) protocols

The socket has the indicated *type*, which specifies the communication semantics. Currently
defined types are:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM

There must be an entry in the netconfig(4) file for at least each protocol family and type
required. If a non-zero protocol has been specified but no exact match for the protocol family,
type, and protocol is found, then the first entry containing the specified family and type with a
protocol value of zero will be used.

A SOCK_STREAM type provides sequenced, reliable, two-way connection-based byte streams.
An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket
supports datagrams (connectionless, unreliable messages of a fixed (typically small)
maximum length). A SOCK_SEQPACKET socket may provide a sequenced, reliable, two-way
connection-based data transmission path for datagrams of fixed maximum length; a
consumer may be required to read an entire packet with each read system call. This facility is
protocol specific, and presently not implemented for any protocol family. SOCK_RAW sockets
provide access to internal network interfaces. The types SOCK_RAW, which is available only to a
user with the net_rawaccess privilege, and SOCK_RDM, for which no implementation currently
exists, are not described here.

The *protocol* parameter is a protocol-family-specific value which specifies a particular protocol to be used with the socket. Normally this value is zero, as commonly only a single protocol exists to support a particular socket type within a given protocol family. However, multiple protocols may exist, in which case a particular protocol may be specified in this manner.

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a connect(3SOCKET) call. Once connected, data may be transferred using read(2) and write(2) calls or some variant of the send(3SOCKET) and recv(3SOCKET) calls. When a session has been completed, a close(2) may be performed. Out-of-band data may also be transmitted as described on the send(3SOCKET) manual page and received as described on the recv(3SOCKET) manual page.

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with −1 returns and with ETIMEDOUT as the specific code in the global variable errno. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (for instance 5 minutes). A SIGPIPE signal is raised if a thread sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_SEQPACKET sockets employ the same system calls as SOCK_STREAM sockets. The only difference is that read(2) calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded.

SOCK_DGRAM and SOCK_RAW sockets allow datagrams to be sent to correspondents named in sendto(3SOCKET) calls. Datagrams are generally received with recvfrom(3SOCKET), which returns the next datagram with its return address.

An fcntl(2) call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives. It can also enable non-blocking I/O.

The operation of sockets is controlled by socket level *options*. These options are defined in the file <sys/socket.h>. setsockopt(3SOCKET) and getsockopt(3SOCKET) are used to set and get options, respectively.

Return Values Upon successful completion, a descriptor referencing the socket is returned. Otherwise, -1 is returned and errno is set to indicate the error.

Errors The socket() function will fail if:

EACCES   Permission to create a socket of the specified type or protocol is denied.

EAGAIN	There were insufficient resources available to complete the operation.
EAFNOSUPPORT	The specified address family is not supported by the protocol family.
EMFILE	The per-process descriptor table is full.
ENOMEM	Insufficient user memory is available.
ENOSR	There were insufficient STREAMS resources available to complete the operation.
EPFNOSUPPORT	The specified protocol family is not supported.
EPROTONOSUPPORT	The protocol type is not supported by the address family.
EPROTOTYPE	The socket type is not supported by the protocol.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also close(2), fcntl(2), ioctl(2), read(2), write(2), accept(3SOCKET), bind(3SOCKET), connect(3SOCKET), getsockname(3SOCKET), getsockopt(3SOCKET), in.h(3HEAD), listen(3SOCKET), recv(3SOCKET), setsockopt(3SOCKET), send(3SOCKET), shutdown(3SOCKET), socket.h(3HEAD), socketpair(3SOCKET), attributes(5), nca(7d)

Notes Historically, AF_* was commonly used in places where PF_* was meant. New code should be careful to use PF_* as necessary.

Name   socket – create an endpoint for communication

Synopsis   cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <sys/socket.h>

int socket(int *domain*, int *type*, int *protocol*);

Description   The socket() function creates an unbound socket in a communications domain, and returns a file descriptor that can be used in later function calls that operate on sockets.

The <sys/socket.h> header defines at least the following values for the *domain* argument:

AF_UNIX   File system pathnames.

AF_INET   Internet Protocol version 4 (IPv4) address.

AF_INET6   Internet Protocol version 6 (IPv6) address.

The *type* argument specifies the socket type, which determines the semantics of communication over the socket. The socket types supported by the system are implementation-dependent. Possible socket types include:

SOCK_STREAM   Provides sequenced, reliable, bidirectional, connection-mode byte streams, and may provide a transmission mechanism for out-of-band data.

SOCK_DGRAM   Provides datagrams, which are connectionless-mode, unreliable messages of fixed maximum length.

SOCK_SEQPACKET   Provides sequenced, reliable, bidirectional, connection-mode transmission path for records. A record can be sent using one or more output operations and received using one or more input operations, but a single operation never transfers part of more than one record. Record boundaries are visible to the receiver via the MSG_EOR flag.

If the *protocol* argument is non-zero, it must specify a protocol that is supported by the address family. The protocols supported by the system are implementation-dependent.

The process may need to have appropriate privileges to use the socket() function or to create some sockets.

Parameters   The function takes the following arguments:

domain   Specifies the communications domain in which a socket is to be created.

type   Specifies the type of socket to be created.

protocol   Specifies a particular protocol to be used with the socket. Specifying a *protocol* of 0 causes socket() to use an unspecified default protocol appropriate for the requested socket type.

The *domain* argument specifies the address family used in the communications domain. The address families supported by the system are implementation-dependent.

Usage The documentation for specific address families specify which protocols each address family supports. The documentation for specific protocols specify which socket types each protocol supports.

The application can determine if an address family is supported by trying to create a socket with *domain* set to the protocol in question.

Return Values Upon successful completion, socket() returns a nonnegative integer, the socket file descriptor. Otherwise a value of -1 is returned and errno is set to indicate the error.

Errors The socket() function will fail if:

EAFNOSUPPORT   The implementation does not support the specified address family.

EMFILE   No more file descriptors are available for this process.

ENFILE   No more file descriptors are available for the system.

EPROTONOSUPPORT   The protocol is not supported by the address family, or the protocol is not supported by the implementation.

EPROTOTYPE   The socket type is not supported by the protocol.

The socket() function may fail if:

EACCES   The process does not have appropriate privileges.

ENOBUFS   Insufficient resources were available in the system to perform the operation.

ENOMEM   Insufficient memory was available to fulfill the request.

ENOSR   There were insufficient STREAMS resources available for the operation to complete.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also accept(3XNET), bind(3XNET), connect(3XNET), getsockname(3XNET), getsockopt(3XNET), listen(3XNET), recv(3XNET), recvfrom(3XNET), recvmsg(3XNET), send(3XNET), sendmsg(3XNET), setsockopt(3XNET), shutdown(3XNET), socketpair(3XNET), attributes(5), standards(5)

Name socketpair – create a pair of connected sockets

Synopsis cc [*flag* ...] *file* ... -lsocket -lnsl [*library* ...]
#include <sys/types.h>
#include <sys/socket.h>

int socketpair(int *domain*, int *type*, int *protocol*, int *sv*[2]);

Description The socketpair() library call creates an unnamed pair of connected sockets in the specified address family *domain*, of the specified *type*, that uses the optionally specified *protocol*. The descriptors that are used in referencing the new sockets are returned in *sv*[0] and *sv*[1]. The two sockets are indistinguishable.

Return Values socketpair() returns −1 on failure and 0 on success.

Errors The call succeeds unless:

EAFNOSUPPORT	The specified address family is not supported on this machine.
EMFILE	Too many descriptors are in use by this process.
ENOMEM	There was insufficient user memory for the operation to complete.
ENOSR	There were insufficient STREAMS resources for the operation to complete.
EOPNOTSUPP	The specified protocol does not support creation of socket pairs.
EPROTONOSUPPORT	The specified protocol is not supported on this machine.
EACCES	The process does not have appropriate privileges.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also pipe(2), read(2), write(2), socket.h(3HEAD), attributes(5)

Notes This call is currently implemented only for the AF_UNIX address family.

Name socketpair – create a pair of connected sockets

Synopsis cc [*flag* ...] *file* ... -lxnet [*library* ...]
#include <sys/socket.h>

int socketpair(int *domain*, int *type*, int *protocol*, int *socket_vector*[2]);

Description The socketpair() function creates an unbound pair of connected sockets in a specified *domain*, of a specified type, under the protocol optionally specified by the *protocol* argument. The two sockets are identical. The file descriptors used in referencing the created sockets are returned in *socket_vector*0 and *socket_vector*1.

The *type* argument specifies the socket type, which determines the semantics of communications over the socket. The socket types supported by the system are implementation-dependent. Possible socket types include:

SOCK_STREAM Provides sequenced, reliable, bidirectional, connection-mode byte streams, and may provide a transmission mechanism for out-of-band data.

SOCK_DGRAM Provides datagrams, which are connectionless-mode, unreliable messages of fixed maximum length.

SOCK_SEQPACKET Provides sequenced, reliable, bidirectional, connection-mode transmission path for records. A record can be sent using one or more output operations and received using one or more input operations, but a single operation never transfers part of more than one record. Record boundaries are visible to the receiver via the MSG_EOR flag.

If the *protocol* argument is non-zero, it must specify a protocol that is supported by the address family. The protocols supported by the system are implementation-dependent.

The process may need to have appropriate privileges to use the socketpair() function or to create some sockets.

Parameters *domain* Specifies the communications domain in which the sockets are to be created.

type Specifies the type of sockets to be created.

protocol Specifies a particular protocol to be used with the sockets. Specifying a *protocol* of 0 causes socketpair() to use an unspecified default protocol appropriate for the requested socket type.

socket_vector Specifies a 2-integer array to hold the file descriptors of the created socket pair.

Usage The documentation for specific address families specifies which protocols each address family supports. The documentation for specific protocols specifies which socket types each protocol supports.

The socketpair() function is used primarily with UNIX domain sockets and need not be supported for other domains.

Return Values Upon successful completion, this function returns 0. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The socketpair() function will fail if:

EAFNOSUPPORT	The implementation does not support the specified address family.
EMFILE	No more file descriptors are available for this process.
ENFILE	No more file descriptors are available for the system.
EOPNOTSUPP	The specified protocol does not permit creation of socket pairs.
EPROTONOSUPPORT	The protocol is not supported by the address family, or the protocol is not supported by the implementation.
EPROTOTYPE	The socket type is not supported by the protocol.

The socketpair() function may fail if:

EACCES	The process does not have appropriate privileges.
ENOBUFS	Insufficient resources were available in the system to perform the operation.
ENOMEM	Insufficient memory was available to fulfill the request.
ENOSR	There were insufficient STREAMS resources available for the operation to complete.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See standards(5).

See Also socket(3XNET), attributes(5), standards(5)

Name spray – scatter data in order to test the network

Synopsis cc [*flag* ...] *file* ... -lsocket -lnsl [*library* ...]
#include <rpcsvc/spray.h>

bool_t xdr_sprayarr(XDR **xdrs*, sprayarr **objp*);

bool_t xdr_spraycumul(XDR **xdrs*, spraycumul **objp*);

Description The spray program sends packets to a given machine to test communications with that
machine.

The spray program is not a C function interface, per se, but it can be accessed using the generic
remote procedure calling interface clnt_call(). See rpc_clnt_calls(3NSL). The program
sends a packet to the called host. The host acknowledges receipt of the packet. The program
counts the number of acknowledgments and can return that count.

The spray program currently supports the following procedures, which should be called in the
order given:

SPRAYPROC_CLEAR   This procedure clears the counter.

SPRAYPROC_SPRAY   This procedure sends the packet.

SPRAYPROC_GET   This procedure returns the count and the amount of time since the last
                    SPRAYPROC_CLEAR.

Examples **EXAMPLE 1** Using spray()

The following code fragment demonstrates how the spray program is used:

```
#include <rpc/rpc.h>
#include <rpcsvc/spray.h>
 . . .
    spraycumul    spray_result;
    sprayarr     spray_data;
    char         buf[100];        /* arbitrary data */
    int          loop = 1000;
    CLIENT    *clnt;
    struct timeval timeout0 = {0, 0};
    struct timeval timeout25 = {25, 0};
    spray_data.sprayarr_len = (uint_t)100;
    spray_data.sprayarr_val = buf;
    clnt = clnt_create("somehost", SPRAYPROG, SPRAYVERS, "netpath");
    if (clnt == (CLIENT *)NULL) {
        /* handle this error */
    }
    if (clnt_call(clnt, SPRAYPROC_CLEAR,
        xdr_void, NULL, xdr_void, NULL, timeout25)) {
            /* handle this error */
    }
```

EXAMPLE 1 Using spray() *(Continued)*

```
while (loop— > 0) {
    if (clnt_call(clnt, SPRAYPROC_SPRAY,
        xdr_sprayarr, &spray_data, xdr_void, NULL, timeout0)) {
            /* handle this error */
    }
}
if (clnt_call(clnt, SPRAYPROC_GET,
    xdr_void, NULL, xdr_spraycumul, &spray_result, timeout25)) {
        /* handle this error */
}
printf("Acknowledged %ld of 1000 packets in %d secs %d usecs\n",
    spray_result.counter,
    spray_result.clock.sec,
    spray_result.clock.usec);
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also spray(1M), rpc_clnt_calls(3NSL), attributes(5)

Notes This interface is unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

A spray program is not useful as a networking benchmark as it uses unreliable connectionless transports, for example, udp. It can report a large number of packets dropped, when the drops were caused by the program sending packets faster than they can be buffered locally, that is, before the packets get to the network medium.

Name t_accept – accept a connection request

Synopsis #include <xti.h>

int t_accept(int *fd*, int *resfd*, const struct t_call **call*);

Description This routine is part of the XTI interfaces that evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, a different header file, tiuser.h, must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

This function is issued by a transport user to accept a connection request. The parameter *fd* identifies the local transport endpoint where the connection indication arrived; *resfd* specifies the local transport endpoint where the connection is to be established, and *call* contains information required by the transport provider to complete the connection. 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 *call*, *addr* is the protocol address of the calling transport user, *opt* indicates any options associated with the connection, *udata* points to any user data to be returned to the caller, and *sequence* is the value returned by t_listen(3NSL) that uniquely associates the response with a previously received connection indication. The address of the caller, *addr* may be null (length zero). Where *addr* is not null then it may optionally be checked by XTI.

A transport user may accept a connection on either the same, or on a different, local transport endpoint than the one on which the connection indication arrived. Before the connection can be accepted on the same endpoint (*resfd==fd*), the user must have responded to any previous connection indications received on that transport endpoint by means of t_accept() or t_snddis(3NSL). Otherwise, t_accept() will fail and set t_errno to TINDOUT.

If a different transport endpoint is specified (*resfd!=fd*), then the user may or may not choose to bind the endpoint before the t_accept() is issued. If the endpoint is not bound prior to the t_accept(), the endpoint must be in the T_UNBND state before the t_accept() is issued, and the transport provider will automatically bind it to an address that is appropriate for the protocol concerned. If the transport user chooses to bind the endpoint it must be bound to a protocol address with a *qlen* of zero and must be in the T_IDLE state before the t_accept() is issued.

Responding endpoints should be supplied to t_accept() in the state T_UNBND.

The call to t_accept() may fail with t_errno set to TLOOK if there are indications (for example connect or disconnect) waiting to be received on endpoint *fd*. Applications should be prepared for such a failure.

The *udata* argument enables the called transport user to send user data to the caller and the amount of user data must not exceed the limits supported by the transport provider as returned in the *connect* field of the *info* argument of t_open(3NSL) or t_getinfo(3NSL). If the *len* field of *udata* is zero, no data will be sent to the caller. All the *maxlen* fields are meaningless.

When the user does not indicate any option (*call→opt.len* = 0) the connection shall be accepted with the option values currently set for the responding endpoint *resfd*.

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.

Valid States fd: T_INCON

resfd (fd!=resfd): T_IDLE, T_UNBND

Errors 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	Either an invalid sequence number was specified, or a valid sequence number was specified but the connection request was aborted by the peer. In the latter case, its T_DISCONNECT event will be received on the listening endpoint.
TINDOUT	The function was called with *fd==resfd* but there are outstanding connection indications on the endpoint. Those other connection indications must be handled either by rejecting them by means of t_snddis(3NSL) or accepting them on a different endpoint by means of t_accept.
TLOOK	An asynchronous event has occurred on the transport endpoint referenced by *fd* and requires immediate attention.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TOUTSTATE	The communications endpoint referenced by *fd* or *resfd* is not in one of the states in which a call to this function is valid.

TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (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* != *fd*) was bound to a protocol address with a *qlen* that is greater than zero.
TSYSERR	A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

```
#include <tiuser.h>
```

Error Description Values The t_errno values that can be set by the XTI interface and cannot be set by the TLI interface are:

TPROTO

TINDOUT

TPROVMISMATCH

TRESADDR

TRESQLEN

Option Buffer The format of the options in an opt buffer is dictated by the transport provider. Unlike the XTI interface, the TLI interface does not specify the buffer format.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also t_connect(3NSL), t_getinfo(3NSL), t_getstate(3NSL), t_listen(3NSL), t_open(3NSL), t_optmgmt(3NSL), t_rcvconnect(3NSL), t_snddis(3NSL), attributes(5)

Warnings   There may be transport provider-specific restrictions on address binding.

Some transport providers do not differentiate between a connection indication and the connection itself. If the connection has already been established after a successful return of t_listen(3NSL), t_accept() will assign the existing connection to the transport endpoint specified by *resfd*.

Name   t_alloc – allocate a library structure

Synopsis   #include <xti.h>

void *t_alloc(int *fd*, int *struct_type*, int *fields*);

Description   This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI
represents the future evolution of these interfaces. However, TLI interfaces are supported for
compatibility. When using a TLI routine that has the same name as an XTI routine, a different
header file, tiuser.h, must be used. Refer to the section, TLI COMPATIBILITY, for a description
of differences between the two interfaces.

The t_alloc() function dynamically allocates memory for the various transport function
argument structures as specified below. This function will allocate memory for the specified
structure, and will also allocate memory for buffers referenced by the structure.

The structure to allocate is specified by *struct_type* and must 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
```

where each of these structures may subsequently be used as an argument to one or more
transport functions.

Each of the above structures, except T_INFO, contains at least one field of type struct netbuf.
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* argument of t_open(3NSL) or t_getinfo(3NSL). The relevant fields of
the *info* argument are described in the following list. The *fields* argument specifies which
buffers to allocate, where the argument 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 structures.

T_OPT   The *opt* field of the t_optmgmt, t_call, t_unitdata or t_uderr 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 *fd* will not be allocated.

For each relevant field specified in *fields*, t_alloc() will allocate memory for the buffer
associated with the field, and initialize the *len* field to zero and the *buf* pointer and *maxlen* field
accordingly. Irrelevant or unknown values passed in fields are ignored. Since the length of the
buffer allocated will be based on the same size information that is returned to the user on a call
to t_open(3NSL) and t_getinfo(3NSL), *fd* must refer to the transport endpoint through

which the newly allocated structure will be passed. In the case where a T_INFO structure is to be allocated, *fd* may be set to any value. In this way the appropriate size information can be accessed. If the size value associated with any specified field is T_INVALID, t_alloc() will be unable to determine the size of the buffer to allocate and will fail, setting t_errno to TSYSERR and errno to EINVAL. See t_open(3NSL) or t_getinfo(3NSL). If the size value associated with any specified field is T_INFINITE, then the behavior of t_alloc() is implementation-defined. For any field not specified in *fields*, *buf* will be set to the null pointer and *len* and *maxlen* will be set to zero. See t_open(3NSL) or t_getinfo(3NSL).

The pointer returned if the allocation succeeds is suitably aligned so that it can be assigned to a pointer to any type of object and then used to access such an object or array of such objects in the space allocated.

Use of t_alloc() to allocate structures will help ensure the compatibility of user programs with future releases of the transport interface functions.

Return Values On successful completion, t_alloc() returns a pointer to the newly allocated structure. On failure, a null pointer is returned.

Valid States ALL - apart from T_UNINIT

Errors On failure, t_errno is set to one of the following:

TBADF   struct_type is other than T_INFO and the specified file descriptor does not refer to a transport endpoint.

TNOSTRUCTYPE   Unsupported *struct_type* requested. This can include a request for a structure type which is inconsistent with the transport provider type specified, that is, connection-mode or connectionless-mode.

TPROTO   This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).

TSYSERR   A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

#include <tiuser.h>

Error Description Values The t_errno values that can be set by the XTI interface and cannot be set by the TLI interface are:

TPROTO

TNOSTRUCTYPE

Special Buffer Sizes   Assume that the value associated with any field of struct t_info (argument returned by t_open() or t_getinfo()) that describes buffer limits is –1. Then the underlying service provider can support a buffer of unlimited size. If this is the case, t_alloc() will allocate a buffer with the default size 1024 bytes, which may be handled as described in the next paragraph.

If the underlying service provider supports a buffer of unlimited size in the netbuf structure (see t_connect(3NSL)), t_alloc() will return a buffer of size 1024 bytes. If a larger size buffer is required, it will need to be allocated separately using a memory allocation routine such as malloc(3C). The buf and maxlen fields of the netbuf data structure can then be updated with the address of the new buffer and the 1024 byte buffer originally allocated by t_alloc() can be freed using free(3C).

Assume that the value associated with any field of struct t_info (argument returned by t_open() or t_getinfo()) that describes nbuffer limits is –2. Then t_alloc() will set the buffer pointer to NULL and the buffer maximum size to 0, and then will return success (see t_open(3NSL) or t_getinfo(3NSL)).

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also   free(3C), malloc(3C), t_connect(3NSL), t_free(3NSL), t_getinfo(3NSL), t_open(3NSL), attributes(5)

Name   t_bind – bind an address to a transport endpoint

Synopsis   `#include <xti.h>`

`int t_bind(int` *fd*`, const struct t_bind *`*req*`, struct t_bind *`*ret*`);`

Description   This routine is part of the XTI interfaces that evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the `tiuser.h`header file must be used. Refer to the `TLI COMPATIBILITY` section for a description of differences between the two interfaces.

This function associates a protocol address with the transport endpoint specified by *fd* 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* arguments point to a t_bind structure containing the following members:

```
struct netbuf    addr;
unsigned    qlen;
```

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 connection indications.

The parameter *req* is used to request that an address, represented by the netbuf structure, be bound to the given transport endpoint. The parameter *len* specifies the number of bytes in the address, and *buf* points to the address buffer. The parameter *maxlen* has no meaning for the *req* argument. On return, *ret* contains an encoding for the address that the transport provider actually bound to the transport endpoint; if an address was specified in *req*, this will be an encoding of the same address. In *ret*, the user specifies *maxlen,* which is the maximum size of the address buffer, and *buf* which points to the buffer where the address is to be placed. On return, *len* specifies the number of bytes in the bound address, and *buf* points to the bound address. If *maxlen* equals zero, no address is returned. If *maxlen* is greater than zero and less than the length of the address, t_bind() fails with t_errno set to TBUFOVFLW.

If the requested address is not available, t_bind() will return –1 with t_errno set as appropriate. If no address is specified in *req* (the *len* field of *addr* in *req* is zero or *req* is NULL), the transport provider will assign an appropriate address to be bound, and will return that address in the *addr* field of *ret*. If the transport provider could not allocate an address, t_bind() will fail with t_errno set to TNOADDR.

The parameter *req* may be a null pointer if the user does not wish to specify an address to be bound. Here, the value of *qlen* is assumed to be zero, and the transport provider will assign an address to the transport endpoint. Similarly, *ret* 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 *qlen*. It

is valid to set *req* and *ret* 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.

The *qlen* field has meaning only when initializing a connection-mode service. It specifies the number of outstanding connection indications that the transport provider should support for the given transport endpoint. An outstanding connection indication is one that has been passed to the transport user by the transport provider but which has not been accepted or rejected. A value of *qlen* greater than zero is only meaningful when issued by a passive transport user that expects other users to call it. The value of *qlen* will be negotiated by the transport provider and may be changed if the transport provider cannot support the specified number of outstanding connection indications. However, this value of *qlen* will never be negotiated from a requested value greater than zero to zero. This is a requirement on transport providers; see WARNINGS below. On return, the *qlen* field in *ret* will contain the negotiated value.

If *fd* refers to a connection-mode service, this function allows more than one transport endpoint to be bound to the same protocol address. but it is not possible to bind more than one protocol address to the same transport endpoint. However, the transport provider must also support this capability. If a user binds more than one transport endpoint to the same protocol address, only one endpoint can be used to listen for connection indications associated with that protocol address. In other words, only one t_bind() for a given protocol address may specify a value of *qlen* greater than zero. In this way, the transport provider can identify which transport endpoint should be notified of an incoming connection indication. If a user attempts to bind a protocol address to a second transport endpoint with a value of *qlen* 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(3NSL) or t_close(3NSL) 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 connection indications.

If *fd* refers to connectionless mode service, this function allows for more than one transport endpoint to be associated with a protocol address, where the underlying transport provider supports this capability (often in conjunction with value of a protocol-specific option). If a user attempts to bind a second transport endpoint to an already bound protocol address when such capability is not supported for a transport provider, t_bind() will return –1 and set t_errno to TADDRBUSY.

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.

Valid States	T_UNBND
Errors	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.
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 provider's state will change to T_IDLE and the information to be returned in *ret* will be discarded.
TOUTSTATE	The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.
TNOADDR	The transport provider could not allocate an address.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).
TSYSERR	A system error has occurred during execution of this function.

Tli Compatibility   The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header   The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

#include <tiuser.h>

Address Bound   The user can compare the addresses in *req* and *ret* to determine whether the transport provider bound the transport endpoint to a different address than that requested.

Error Description Values   The t_errno values TPROTO and TADDRBUSY can be set by the XTI interface but cannot be set by the TLI interface.

A t_errno value that this routine can return under different circumstances than its XTI counterpart is TBUFOVFLW. It can be returned even when the maxlen field of the corresponding buffer has been set to zero.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also t_accept(3NSL), t_alloc(3NSL), t_close(3NSL), t_connect(3NSL), t_unbind(3NSL), attributes(5)

Warnings The requirement that the value of *qlen* never be negotiated from a requested value greater than zero to zero implies that transport providers, rather than the XTI implementation itself, accept this restriction.

An implementation need not allow an application explicitly to bind more than one communications endpoint to a single protocol address, while permitting more than one connection to be accepted to the same protocol address. That means that although an attempt to bind a communications endpoint to some address with *qlen=0* might be rejected with TADDRBUSY, the user may nevertheless use this (unbound) endpoint as a responding endpoint in a call to t_accept(3NSL). To become independent of such implementation differences, the user should supply unbound responding endpoints to t_accept(3NSL).

The local address bound to an endpoint may change as result of a t_accept(3NSL) or t_connect(3NSL) call. Such changes are not necessarily reversed when the connection is released.

Name t_close – close a transport endpoint

Synopsis #include <xti.h>

int t_close(int *fd*);

Description This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

The t_close() function informs the transport provider that the user is finished with the transport endpoint specified by *fd*, and frees any local library resources associated with the endpoint. In addition, t_close() closes the file associated with the transport endpoint.

The function t_close() should be called from the T_UNBND state. See t_getstate(3NSL). However, this function 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, close(2) will be issued for that file descriptor; if there are no other descriptors in this process or in another process which references the communication endpoint, any connection that may be associated with that endpoint is broken. The connection may be terminated in an orderly or abortive manner.

A t_close() 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.

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.

Valid States T_UNBND

Errors On failure, t_errno is set to 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 XTI and the transport provider for which there is no other suitable XTI error (t_errno).

TSYSERR   A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

#include <tiuser.h>

Error Description Values The t_errno value that can be set by the XTI interface and cannot be set by the TLI interface is:

TPROTO

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also close(2), t_getstate(3NSL), t_open(3NSL), t_unbind(3NSL), attributes(5)

Name t_connect – establish a connection with another transport user

Synopsis #include <xti.h>

```
int t_connect(int fd, const struct t_call *sndcall,
    struct t_call *rcvcall);
```

Description This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces. This function enables a transport user to request a connection to the specified destination transport user.

This function can only be issued in the T_IDLE state. The parameter *fd* identifies the local transport endpoint where communication will be established, while *sndcall* and *rcvcall* point to a t_call structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The parameter *sndcall* specifies information needed by the transport provider to establish a connection and *rcvcall* specifies information that is associated with the newly established connection.

In *sndcall*, *addr* specifies the protocol address of the destination transport user, *opt* presents any protocol-specific information that might be needed by the transport provider, *udata* points to optional user data that may be passed to the destination transport user during connection establishment, and *sequence* has no meaning for this function.

On return, in *rcvcall*, *addr* contains the protocol address associated with the responding transport endpoint, *opt* represents any 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, and *sequence* has no meaning for this function.

The *opt* argument permits users to define the options that may be passed to the transport provider. The user may choose not to negotiate protocol options by setting the *len* field of *opt* to zero. In this case, the provider uses the option values currently set for the communications endpoint.

If used, *sndcall→opt.buf* must point to a buffer with the corresponding options, and *sndcall→opt.len* must specify its length. The *maxlen* and *buf* fields of the netbuf structure pointed by *rcvcall→addr* and *rcvcall→opt* must be set before the call.

The *udata* argument 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* argument of t_open(3NSL) or t_getinfo(3NSL). If the *len* of *udata* is zero in *sndcall*, no data will be sent to the destination transport user.

On return, the *addr*, *opt* and *udata* fields of *rcvcall* will be updated to reflect values associated with the connection. Thus, the *maxlen* field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *maxlen* can be set to zero, in which case no information to this specific argument is given to the user on the return from t_connect(). If maxlen is greater than zero and less than the length of the value, t_connect() fails with t_errno set to TBUFOVFLW. If *rcvcall* is set to NULL, no information at all is returned.

By default, t_connect() executes in synchronous mode, and will wait for the destination user's response before returning control to the local user. A successful return (that is, return value of zero) indicates that the requested connection has been established. However, if O_NONBLOCK is set by means of t_open(3NSL) or fcntl(2), t_connect() executes in asynchronous mode. In this case, the call will not wait for the remote user's response, but will return control immediately to the local user and return –1 with t_errno 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 connection request to the destination transport user. The t_rcvconnect(3NSL) function is used in conjunction with t_connect() 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 t_rcvconnect(3NSL), t_rcvdis(3NSL) or t_snddis(3NSL). When an asynchronous t_connect() call is interrupted by the arrival of a signal, the state of the corresponding transport endpoint is 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.

Valid States T_IDLE

Errors 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 argument *(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_DATAXFER, and the information to be returned in *rcvcall* is discarded.	
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.	
TNODATA	O_NONBLOCK 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 communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.	
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).	
TSYSERR	A system error has occurred during execution of this function.	

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

```
#include <tiuser.h>
```

Error Description Values The TPROTO and TADDRBUSY t_errno values can be set by the XTI interface but not by the TLI interface.

A t_errno value that this routine can return under different circumstances than its XTI counterpart is TBUFOVFLW. It can be returned even when the maxlen field of the corresponding buffer has been set to zero.

Option Buffers   The format of the options in an opt buffer is dictated by the transport provider. Unlike the
XTI interface, the TLI interface does not fix the buffer format.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also   fcntl(2), t_accept(3NSL), t_alloc(3NSL), t_getinfo(3NSL), t_listen(3NSL),
t_open(3NSL), t_optmgmt(3NSL), t_rcvconnect(3NSL), t_rcvdis(3NSL), t_snddis(3NSL),
attributes

Name t_errno – XTI error return value

Synopsis #include <xti.h>

Description This error return value is part of the XTI interfaces that evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI interface that has the same name as an XTI interfaces, a different headerfile, <tiuser.h>, must be used. Refer the the TLI COMPATIBILITY section for a description of differences between the two interfaces.

t_errno is used by XTI functions to return error values.

XTI functions provide an error number in t_errno which has type *int* and is defined in <xti.h>. The value of t_errno will be defined only after a call to a XTI function for which it is explicitly stated to be set and until it is changed by the next XTI function call. The value of t_errno should only be examined when it is indicated to be valid by a function's return value. Programs should obtain the definition of t_errno by the inclusion of <xti.h>. The practice of defining t_errno in program as extern int t_errno is obsolescent. No XTI function sets t_errno to 0 to indicate an error.

It is unspecified whether t_errno is a macro or an identifier with external linkage. It represents a modifiable lvalue of type *int*. If a macro definition is suppressed in order to access an actual object or a program defines an identifier with name *t_errno*, the behavior is undefined.

The symbolic values stored in t_errno by an XTI function are defined in the ERRORS sections in all relevant XTI function definition pages.

Tli Compatibility t_errno is also used by TLI functions to return error values.

The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, <xti.h>. TLI interfaces should *not* use this header. They should use the header:

#include <tiuser.h>

Error Description Values The t_errno values that can be set by the XTI interface but cannot be set by the TLI interface are:

TNOSTRUCTYPE
TBADNAME
TBADQLEN
TADDRBUSY

```
TINDOUT
TPROVMISMATCH
TRESADDR
TQFULL
TPROTO
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also attributes(5)

Name t_error – produce error message

Synopsis #include <xti.h>

int t_error(const char *errmsg);

Description This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

The t_error() function produces a message on the standard error output which describes the last error encountered during a call to a transport function. The argument string *errmsg* is a user-supplied error message that gives context to the error.

The error message is written as follows: first (if *errmsg* is not a null pointer and the character pointed to be *errmsg* is not the null character) the string pointed to by *errmsg* followed by a colon and a space; then 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 errno followed by a newline.

The language for error message strings written by t_error() is that of the current locale. If it is English, the error message string describing the value in t_errno may be derived from the comments following the t_errno codes defined in xti.h. The contents of the error message strings describing the value in errno are the same as those returned by the strerror(3C) function with an argument of errno.

The error number, t_errno, is only set when an error occurs and it is not cleared on successful calls.

Examples If a t_connect(3NSL) function 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.

Return Values Upon completion, a value of 0 is returned.

Valid States All - apart from T_UNINIT

Errors No errors are defined for the t_error() function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

```
#include <tiuser.h>
```

Error Description Values The t_errno value that can be set by the XTI interface and cannot be set by the TLI interface is:

TPROTO

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also t_errno(3NSL)strerror(3C), attributes(5)

Name t_free – free a library structure

Synopsis #include <xti.h>

int t_free(void *ptr, int struct_type);

Description This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

The t_free() function frees memory previously allocated by t_alloc(3NSL). This function will free memory for the specified structure, and will also free memory for buffers referenced by the structure.

The argument *ptr* points to one of the seven structure types described for t_alloc(3NSL), and *struct_type* identifies the type of that structure which must 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
```

where each of these structures is used as an argument to one or more transport functions.

The function t_free() will check the *addr*, *opt* and *udata* fields of the given structure, as appropriate, and free the buffers pointed to by the *buf* field of the netbuf structure. If *buf* is a null pointer, t_free() will not attempt to free memory. After all buffers are freed, t_free() will free the memory associated with the structure pointed to by *ptr*.

Undefined results will occur if *ptr* or any of the *buf* pointers points to a block of memory that was not previously allocated by t_alloc(3NSL).

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.

Valid States ALL - apart from T_UNINIT.

Errors On failure, t_errno is set to the following:

TNOSTRUCTYPE   Unsupported *struct_type* requested.

TPROTO   This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error *(t_errno)*.

TSYSERR   A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

    #include <tiuser.h>

Error Description Values The t_errno value that can be set by the XTI interface and cannot be set by the TLI interface is:

    TPROTO

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also t_alloc(3NSL), attributes(5)

Name t_getinfo – get protocol-specific service information

Synopsis `#include <xti.h>`

`int t_getinfo(int `*fd*`, struct t_info *`*info*`);`

Description This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the `tiuser.h` header file must be used. Refer to the `TLI COMPATIBILITY` section for a description of differences between the two interfaces.

This function returns the current characteristics of the underlying transport protocol and/or transport connection associated with file descriptor *fd*. The *info* pointer is used to return the same information returned by `t_open(3NSL)`, although not necessarily precisely the same values. This function enables a transport user to access this information during any phase of communication.

This argument points to a `t_info` structure which contains the following members:

```
t_scalar_t addr;      /*max size in octets of the transport protocol address*/
t_scalar_t options;   /*max number of bytes of protocol-specific options  */
t_scalar_t tsdu;      /*max size in octets of a transport service data unit */
t_scalar_t etsdu;     /*max size in octets of an expedited transport service*/
                      /*data unit (ETSDU) */
t_scalar_t connect;   /*max number of octets allowed on connection */
                      /*establishment functions */
t_scalar_t discon;    /*max number of octets of data allowed on t_snddis()  */
                      /*and t_rcvdis() functions */
t_scalar_t servtype;  /*service type supported by the transport provider */
t_scalar_t 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 `T_INVALID` (-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 `T_INVALID` (−2) specifies that the transport provider does not support user-settable options.

tsdu A value greater than zero specifies the maximum size in octets of a transport service data unit (TSDU); a value of `T_NULL` (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 `T_INFINITE` (−1) specifies that there is no limit on the size in octets of a TSDU; and a value of `T_INVALID` (−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 in octets of an expedited transport service data unit (ETSDU); a value of T_NULL (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 T_INFINITE (–1) specifies that there is no limit on the size (in octets) of an ETSDU; and a value of T_INVALID (–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 number of octets that may be associated with connection establishment functions and a value of T_INVALID (–2) specifies that the transport provider does not allow data to be sent with connection establishment functions.

discon If the T_ORDRELDATA bit in flags is clear, a value greater than zero specifies the maximum number of octets that may be associated with the t_snddis(3NSL) and t_rcvdis(3NSL) functions, and a value of T_INVALID (–2) specifies that the transport provider does not allow data to be sent with the abortive release functions. If the T_ORDRELDATA bit is set in flags, a value greater than zero specifies the maximum number of octets that may be associated with the t_sndreldata(), t_rcvreldata(), t_snddis(3NSL) and t_rcvdis(3NSL) functions.

servtype This field specifies the service type supported by the transport provider, as described below.

flags This is a bit field used to specify other information about the communications provider. If the T_ORDRELDATA bit is set, the communications provider supports sending user data with an orderly release. If the T_SENDZERO bit is set in flags, this indicates that 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(3NSL) function 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 may change as a result of protocol option negotiation during connection establishment (the t_optmgmt(3NSL) call has no effect on the values returned by t_getinfo()). These values will only change from the values presented to t_open(3NSL) after the endpoint enters the T_DATAXFER state.

The *servtype* field of *info* specifies 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, t_open(3NSL) will return T_INVALID (−1) for *etsdu*, *connect* and *discon*.

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.

Valid States ALL - apart from T_UNINIT.

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 XTI and the transport provider for which there is no other suitable XTI error (t_errno).
TSYSERR	A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

```
#include <tiuser.h>
```

Error Description Values The t_errno value TPROTO can be set by the XTI interface but not by the TLI interface.

The t_info Structure For TLI , the t_info structure referenced by *info* lacks the following structure member:

```
t_scalar_t flags;     /* other info about the transport provider */
```

This member was added to struct t_info in the XTI interfaces.

When a value of −1 is observed as the return value in various t_info structure members, it signifies that the transport provider can handle an infinite length buffer for a corresponding attribute, such as address data, option data, TSDU (octet size), ETSDU (octet size), connection data, and disconnection data. The corresponding structure members are addr, options, tsdu, estdu, connect, and discon, respectively.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also   t_alloc(3NSL), t_open(3NSL), t_optmgmt(3NSL), t_rcvdis(3NSL), t_snddis(3NSL), attributes(5)

Name t_getprotaddr – get the protocol addresses

Synopsis #include <xti.h>

int t_getprotaddr(int *fd*, struct t_bind **boundaddr*,
    struct t_bind **peeraddr*);

Description This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

The t_getprotaddr() function returns local and remote protocol addresses currently associated with the transport endpoint specified by *fd*. In *boundaddr* and *peeraddr* the user specifies *maxlen*, which is the maximum size (in bytes) of the address buffer, and *buf* which 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*, and the *len* field 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*. The *buf* field of *peeraddr* points to the address, if any, currently connected to *fd*, and the *len* field specifies the length of the address. If the transport endpoint is not in the T_DATAXFER, T_INREL, T_OUTCON or T_OUTREL states, zero is returned in the *len* field of *peeraddr*. If the *maxlen* field of *boundaddr* or *peeraddr* is set to zero, no address is returned.

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 the error.

Valid States ALL - apart from T_UNINIT.

Errors 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.

TPROTO   This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).

TSYSERR   A system error has occurred during execution of this function.

Tli Compatibility In the TLI interface definition, no counterpart of this routine was defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also   t_bind(3NSL), attributes(5)

Name   t_getstate – get the current state

Synopsis   #include <xti.h>

int t_getstate(int *fd*);

Description   This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

The t_getstate() function returns the current state of the provider associated with the transport endpoint specified by *fd*.

Return Values   State is returned upon successful completion. Otherwise, a value of –1 is returned and 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 direction orderly release sent.

T_INREL   Incoming direction orderly release received.

If the provider is undergoing a state transition when t_getstate() is called, the function will fail.

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 XTI and the transport provider for which there is no other suitable XTI error (t_errno).

TSTATECHNG   The transport provider is undergoing a transient state change.

TSYSERR   A system error has occurred during execution of this function.

Tli Compatibility   The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

    #include <tiuser.h>

Error Description Values The t_errno value that can be set by the XTI interface and cannot be set by the TLI interface is:

    TPROTO

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also t_open(3NSL), attributes(5)

Name t_listen – listen for a connection indication

Synopsis #include <xti.h>

int t_listen(int *fd*, struct t_call **call*);

Description This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

This function listens for a connection indication from a calling transport user. The argument *fd* identifies the local transport endpoint where connection indications arrive, and on return, *call* contains information describing the connection 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 *call*, *addr* returns the protocol address of the calling transport user. This address is in a format usable in future calls to t_connect(3NSL). Note, however that t_connect(3NSL) may fail for other reasons, for example TADDRBUSY. *opt* returns options associated with the connection indication, *udata* returns any user data sent by the caller on the connection request, and *sequence* is a number that uniquely identifies the returned connection indication. The value of *sequence* enables the user to listen for multiple connection indications before responding to any of them.

Since this function returns values for the *addr*, *opt* and *udata* fields of *call*, the *maxlen* field of each must be set before issuing the t_listen() to indicate the maximum size of the buffer for each. If the *maxlen* field of *call→addr*, *call→opt* or *call→udata* is set to zero, no information is returned for this parameter.

By default, t_listen() executes in synchronous mode and waits for a connection indication to arrive before returning to the user. However, if O_NONBLOCK is set via t_open(3NSL) or fcntl(2), t_listen() executes asynchronously, reducing to a poll for existing connection indications. If none are available, it returns –1 and sets t_errno to TNODATA.

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.

Valid States T_IDLE, T_INCON

Errors 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 argument *qlen* of the endpoint referenced by *fd* is zero.

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. The
               provider's state, as seen by the user, changes to T_INCON, and the connection
               indication information to be returned in *call* is discarded. The value of
               sequence returned can be used to do a t_snddis(3NSL).

TLOOK   An asynchronous event has occurred on this transport endpoint and
               requires immediate attention.

TNODATA   O_NONBLOCK was set, but no connection indications had been queued.

TNOTSUPPORT   This function is not supported by the underlying transport provider.

TOUTSTATE   The communications endpoint referenced by *fd* is not in one of the states in
               which a call to this function is valid.

TPROTO   This error indicates that a communication problem has been detected
               between XTI and the transport provider for which there is no other suitable
               XTI error (t_errno).

TQFULL   The maximum number of outstanding connection indications has been
               reached for the endpoint referenced by *fd*. Note that a subsequent call to
               t_listen() may block until another incoming connection indication is
               available. This can only occur if at least one of the outstanding connection
               indications becomes no longer outstanding, for example through a call to
               t_accept(3NSL).

TSYSERR   A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files.
This, and other semantic differences between the two interfaces are described in the
subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They
should use the header:

    #include <tiuser.h>

Error Description   The t_errno values TPROT0, TBADQLEN, and TQFULL can be set by the XTI interface but not by
Values   the TLI interface.

A t_errno value that this routine can return under different circumstances than its XTI
counterpart is TBUFOVFLW. It can be returned even when the maxlen field of the corresponding
buffer has been set to zero.

Option Buffers The format of the options in an `opt` buffer is dictated by the transport provider. Unlike the XTI interface, the TLI interface does not fix the buffer format.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also `fcntl`(2), `t_accept`(3NSL), `t_alloc`(3NSL), `t_bind`(3NSL), `t_connect`(3NSL), `t_open`(3NSL), `t_optmgmt`(3NSL), `t_rcvconnect`(3NSL), `t_snddis`(3NSL), `attributes`(5)

Warnings Some transport providers do not differentiate between a connection indication and the connection itself. If this is the case, a successful return of `t_listen()` indicates an existing connection.

Name   t_look – look at the current event on a transport endpoint

Synopsis   #include <xti.h>

```
int t_look(int fd);
```

Description   This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

This function returns the current event on the transport endpoint specified by *fd*. This function enables a transport provider to notify a transport user of an asynchronous event when the user is calling 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 function to be executed.

This function also enables a transport user to poll a transport endpoint periodically for asynchronous events.

Return Values   Upon success, t_look() 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	Disconnection 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.

Valid States   ALL - apart from T_UNINIT.

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
          XTI and the transport provider for which there is no other suitable XTI error
          (t_errno).

TSYSERR   A system error has occurred during execution of this function.

Tli Compatibility   The XTI and TLI interface definitions have common names but use different header files.
This, and other semantic differences between the two interfaces are described in the
subsections below.

Interface Header   The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They
should use the header:

    #include <tiuser.h>

Return Values   The return values that are defined by the XTI interface and cannot be returned by the TLI
interface are:

    T_GODATA
    T_GOEXDATA

Error Description   The t_errno value that can be set by the XTI interface and cannot be set by the TLI interface
Values   is:

    TPROTO

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also   t_open(3NSL), t_snd(3NSL), t_sndudata(3NSL), attributes(5)

Name t_open – establish a transport endpoint

Synopsis #include <xti.h>
#include <fcntl.h>

int t_open(const char *name, int *oflag*, struct t_info *info*);

Description This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI
represents the future evolution of these interfaces. However, TLI interfaces are supported for
compatibility. When using a TLI routine that has the same name as an XTI routine, the
tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description
of differences between the two interfaces.

The t_open() function must be called as the first step in the initialization of a transport
endpoint. This function establishes a transport endpoint by supplying a transport provider
identifier that indicates a particular transport provider, that is, transport protocol, and
returning a file descriptor that identifies that endpoint.

The argument *name* points to a transport provider identifier and *oflag* identifies any open
flags, as in open(2). The argument *oflag* is constructed from O_RDWR optionally bitwise
inclusive-OR'ed with O_NONBLOCK. These flags are defined by the header <fcntl.h>. The file
descriptor returned by t_open() will be used by all subsequent functions to identify the
particular local transport endpoint.

This function also returns various default characteristics of the underlying transport protocol
by setting fields in the t_info structure. This argument points to a t_info which contains the
following members:

```
t_scalar_t addr;        /* max size of the transport protocol address */
t_scalar_t options;     /* max number of bytes of  */
                            /* protocol-specific options  */
t_scalar_t tsdu;        /* max size of a transport service data  */
                            /* unit (TSDU)  */
t_scalar_t etsdu;       /* max size of an expedited transport  */
                            /* service data unit (ETSDU)  */
t_scalar_t connect;     /* max amount of data allowed on  */
                        /* connection establishment functions  */
t_scalar_t discon;      /* max amount of data allowed on  */
                        /* t_snddis() and t_rcvdis() functions  */
t_scalar_t servtype;    /* service type supported by the  */
                            /* transport provider  */
t_scalar_t flags;       /* other info about the transport provider  */
```

The values of the fields have the following meanings:

addr A value greater than zero (T_NULL) indicates the maximum size of a transport
protocol address and a value of –2 (T_INVALID) specifies that the transport
provider does not provide user access to transport protocol addresses.

options   A value greater than zero (T_NULL) indicates the maximum number of bytes of protocol-specific options supported by the provider, and a value of –2 (T_INVALID) specifies that the transport provider does not support user-settable options.

tsdu   A value greater than zero (T_NULL specifies the maximum size of a transport service data unit (TSDU); a value of zero (T_NULL) 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 (T_INFINITE) specifies that there is no limit to the size of a TSDU; and a value of –2 (T_INVALID) specifies that the transfer of normal data is not supported by the transport provider.

etsdu   A value greater than zero (T_NULL) specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero (T_NULL) 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 (T_INFINITE) specifies that there is no limit on the size of an ETSDU; and a value of –2 (T_INVALID) 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 (T_NULL) specifies the maximum amount of data that may be associated with connection establishment functions, and a value of –2 (T_INVALID) specifies that the transport provider does not allow data to be sent with connection establishment functions.

discon   If the T_ORDRELDATA bit in flags is clear, a value greater than zero (T_NULL) specifies the maximum amount of data that may be associated with the t_snddis(3NSL) and t_rcvdis(3NSL) functions, and a value of –2 (T_INVALID) specifies that the transport provider does not allow data to be sent with the abortive release functions. If the T_ORDRELDATA bit is set in flags, a value greater than zero (T_NULL) specifies the maximum number of octets that may be associated with the t_sndreldata(), t_rcvreldata(), t_snddis(3NSL) and t_rcvdis(3NSL) functions.

servtype   This field specifies the service type supported by the transport provider, as described below.

flags   This is a bit field used to specify other information about the communications provider. If the T_ORDRELDATA bit is set, the communications provider supports user data to be sent with an orderly release. 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(3NSL) function may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function.

The *servtype* field of *info* specifies 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, t_open() will return –2 (T_INVALID) for *etsdu*, *connect* and *discon*.

A single transport endpoint may support only one of the above services at one time.

If *info* is set to a null pointer by the transport user, no protocol information is returned by t_open().

Return Values A valid file descriptor is returned upon successful completion. Otherwise, a value of –1 is returned and t_errno is set to indicate an error.

Valid States T_UNINIT.

Errors On failure, t_errno is set to the following:

TBADFLAG   An invalid flag is specified.

TBADNAME   Invalid transport provider name.

TPROTO   This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).

TSYSERR   A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the xti.h TLI interfaces should *not* use this header. They should use the header:

#include <tiuser.h>

Error Description Values   The t_errno values TPROTO and TBADNAME can be set by the XTI interface but cannot be set by the TLI interface.

Notes   For TLI , the t_info structure referenced by *info* lacks the following structure member:

t_scalar_t flags;   /* other info about the transport provider */

This member was added to struct t_info in the XTI interfaces.

When a value of –1 is observed as the return value in various t_info structure members, it signifies that the transport provider can handle an infinite length buffer for a corresponding attribute, such as address data, option data, TSDU (octet size), ETSDU (octet size), connection data, and disconnection data. The corresponding structure members are addr, options, tsdu, estdu, connect, and discon, respectively.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also   open(2), attributes(5)

Name   t_optmgmt – manage options for a transport endpoint

Synopsis   #include <xti.h>

int t_optmgmt(int *fd*, const struct t_optmgmt *req*, struct t_optmgmt *ret*);

Description   This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

The t_optmgmt() function enables a transport user to retrieve, verify or negotiate protocol options with the transport provider. The argument *fd* identifies a transport endpoint.

The *req* and *ret* arguments point to a t_optmgmt structure containing the following members:

```
struct netbuf opt;
t_scalar_t     flags;
```

The *opt* field identifies protocol options and the *flags* field is used to specify the action to take with those options.

The options are represented by a netbuf structure in a manner similar to the address in t_bind(3NSL). The argument *req* is used to request a specific action of the provider and to send options to the provider. The argument *len* specifies the number of bytes in the options, *buf* points to the options buffer, and *maxlen* has no meaning for the *req* argument. The transport provider may return options and flag values to the user through *ret*. For *ret*, *maxlen* specifies the maximum size of the options buffer and *buf* points to the buffer where the options are to be placed. If *maxlen* in *ret* is set to zero, no options values are returned. On return, *len* specifies the number of bytes of options returned. The value in *maxlen* has no meaning for the *req* argument, but must be set in the *ret* argument to specify the maximum number of bytes the options buffer can hold.

Each option in the options buffer is of the form struct t_opthdr possibly followed by an option value.

The *level* field of struct t_opthdr identifies the XTI level or a protocol of the transport provider. The *name* field identifies the option within the level, and *len* contains its total length; that is, 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, the *status* field of the returned options contains information about the success or failure of a negotiation.

Several options can be concatenated. The option user has, however to ensure that each options header and value part starts at a boundary appropriate for the architecture-specific alignment rules. The macros T_OPT_FIRSTHDR(nbp), T_OPT_NEXTHDR (nbp,tohp), T_OPT_DATA(tohp) are provided for that purpose.

T_OPT_DATA(nhp)   If argument is a pointer to a t_opthdr structure, this macro
                          returns an unsigned character pointer to the data associated
                          with the t_opthdr.

T_OPT_NEXTHDR(nbp, tohp)   If the first argument is a pointer to a netbuf structure
                          associated with an option buffer and second argument is a
                          pointer to a t_opthdr structure within that option buffer,
                          this macro returns a pointer to the next t_opthdr structure
                          or a null pointer if this t_opthdr is the last t_opthdr in the
                          option buffer.

T_OPT_FIRSTHDR(tohp)   If the argument is a pointer to a netbuf structure associated
                          with an option buffer, this macro returns the pointer to the
                          first t_opthdr structure in the associated option buffer, or a
                          null pointer if there is no option buffer associated with this
                          netbuf or if it is not possible or the associated option buffer
                          is too small to accommodate even the first aligned option
                          header.

                          T_OPT_FIRSTHDR is useful for finding an appropriately
                          aligned start of the option buffer. T_OPT_NEXTHDR is useful
                          for moving to the start of the next appropriately aligned
                          option in the option buffer. Note that OPT_NEXTHDR is also
                          available for backward compatibility requirements.
                          T_OPT_DATA is useful for finding the start of the data part in
                          the option buffer where the contents of its values start on an
                          appropriately aligned boundary.

                          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 will fail with TBADOPT. If the
                          error is detected, some options have possibly been
                          successfully negotiated. The transport user can check the
                          current status by calling t_optmgmt() with the T_CURRENT
                          flag set.

                          The *flags* field of *req* must specify one of the following
                          actions:

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 by *req→opt.buf* and *req→opt.len*. The negotiated option values are returned in the buffer pointed to by *ret->opt.buf*. The *status* field of each returned option is set to indicate the result of the negotiation. The value is T_SUCCESS if the proposed value was negotiated, T_PARTSUCCESS if a degraded value was negotiated, T_FAILURE if 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, and 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 *ret→flags*.

This field contains the worst single result, whereby the rating is done according to the order T_NOTSUPPORT, T_READONLY, T_FAILURE, T_PARTSUCCESS, T_SUCCESS. The value T_NOTSUPPORT is the worst result and T_SUCCESS is the best.

For each level, the option T_ALLOPT can be requested on input. No value is given with this option; only the t_opthdr part is specified. This input requests to negotiate all supported options of this level to their default values. The result is returned option by option in *ret→opt.buf*. Note that depending on the state of the transport endpoint, not all requests to negotiate the default value may be successful.

T_CHECK   This action enables the user to verify whether the options specified in *req* are supported by the transport provider. If an option is specified with no option value (it consists only of a t_opthdr structure), the option is returned with its *status* field set to T_SUCCESS if it is supported, T_NOTSUPPORT if it is not or needs additional user privileges, and T_READONLY if it is read-only (in the current XTI 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 *ret→flags*. This field contains the worst single result of the option checks, whereby the rating is the same as for T_NEGOTIATE .

Note that no negotiation takes place. All currently effective option values remain unchanged.

T_DEFAULT This action enables the transport user to retrieve the default option values. The user specifies the options of interest in *req→opt.buf*. 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 *ret→opt.buf*.

The *status* field returned is 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, and set to T_SUCCESS in all other cases. The overall result of the request is returned in *ret→flags*. This field contains the worst single result, whereby the rating is the same as for T_NEGOTIATE .

For each level, the option T_ALLOPT can be requested on input. All supported options of this level with their default values are then returned. In this case, *ret→opt.maxlen* must be given at least the value *info→options* before the call. See t_getinfo(3NSL) and t_open(3NSL).

T_CURRENT This action enables the transport user to retrieve the currently effective option values. The user specifies the options of interest in *req→opt.buf*. The option values are irrelevant and will be ignored; it is sufficient to specifiy the t_opthdr part of an option only. The currently effective values are then returned in *req→opt.buf*.

The *status* field returned is 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, and set to T_SUCCESS in all other cases. The overall result of the request is returned in *ret→flags*. This field contains the worst single result, whereby the rating is the same as for T_NEGOTIATE .

For each level, the option T_ALLOPT can be requested on input. All supported options of this level with their currently effective values are then returned.

The option T_ALLOPT can only be used with t_optmgmt() and the actions T_NEGOTIATE, T_DEFAULT and T_CURRENT. It can be used with any supported level and addresses all supported options of this level. The option has no value; it consists of a t_opthdr only. Since in a t_optmgmt() call only options of one level may be addressed, this option should not be requested together with other options. The function 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 function t_optmgmt() may block under various circumstances and depending on the implementation. The function will block, for instance, if the protocol addressed by the call resides on a separate controller. It may also block due to flow control constraints; that is, if data sent previously across this transport endpoint has not yet been fully processed. If the function is interrupted by a signal, the option negotiations that have been done so far may remain valid. The behavior of the function is not changed if O_NONBLOCK is set.

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.

Valid States ALL - apart from T_UNINIT.

Errors On failure, t_errno is set to one of the following:

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.
TNOTSUPPORT	This action is not supported by the transport provider.
TOUTSTATE	The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).
TSYSERR	A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

```
#include <tiuser.h>
```

Error Description Values The t_errno value TPROTO can be set by the XTI interface but not by the TLI interface.

The t_errno values that this routine can return under different circumstances than its XTI counterpart are TACCES and TBUFOVFLW.

TACCES	can be returned to indicate that the user does not have permission to negotiate the specified options.
TBUFOVFLW	can be returned even when the maxlen field of the corresponding buffer has been set to zero.

Option Buffers The format of the options in an opt buffer is dictated by the transport provider. Unlike the XTI interface, the TLI interface does not fix the buffer format. The macros T_OPT_DATA, T_OPT_NEXTHDR, and T_OPT_FIRSTHDR described for XTI are not available for use by TLI interfaces.

Actions The semantic meaning of various action values for the flags field of *req* differs between the TLI and XTI interfaces. TLI interface users should heed the following descriptions of the actions:

T_NEGOTIATE	This action enables the user to negotiate the values of the options specified in *req* with the transport provider. The provider will evaluate the requested options and negotiate the values, returning the negotiated values through *ret*.

T_CHECK   This action enables the user to verify whether the options specified in *req* are
                    supported by the transport provider. On return, the flags field of *ret* will
                    have either T_SUCCESS or T_FAILURE set to indicate to the user whether the
                    options are supported. These flags are only meaningful for the T_CHECK
                    request.

T_DEFAULT   This action enables a user to retrieve the default options supported by the
                    transport provider into the opt field of *ret*. In *req*, the len field of opt must
                    be zero and the buf field may be NULL.

Connectionless Mode If issued as part of the connectionless mode service, t_optmgmt() may block due to flow
control constraints. The function will not complete until the transport provider has processed
all previously sent data units.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also close(2), poll(2), select(3C), t_accept(3NSL), t_alloc(3NSL), t_bind(3NSL),
t_close(3NSL), t_connect(3NSL), t_getinfo(3NSL), t_listen(3NSL), t_open(3NSL),
t_rcv(3NSL), t_rcvconnect(3NSL), t_rcvudata(3NSL), t_snddis(3NSL), attributes(5)

Name t_rcv – receive data or expedited data sent over a connection

Synopsis #include <xti.h>

int t_rcv(int *fd*, void **buf*, unsigned int *nbytes*, int **flags*);

Description This function is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI function that has the same name as an XTI function, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

This function receives either normal or expedited data. The argument *fd* identifies the local transport endpoint through which data will arrive, *buf* points to a receive buffer where user data will be placed, and *nbytes* specifies the size of the receive buffer. The argument *flags* may be set on return from t_rcv() and specifies optional flags as described below.

By default, t_rcv() operates in synchronous mode and will wait for data to arrive if none is currently available. However, if O_NONBLOCK is set by means of t_open(3NSL) or fcntl(2), t_rcv() will execute in asynchronous mode and will fail if no data is available. See TNODATA below.

On return from the call, if T_MORE is set in *flags*, this 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, or under unusual conditions (for example, the arrival of a signal or T_EXDATA event), 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() with the T_MORE flag set indicates that another t_rcv() 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* argument on return from t_open(3NSL) or t_getinfo(3NSL), the T_MORE flag is not meaningful and should be ignored. If *nbytes* is greater than zero on the call to t_rcv(), t_rcv() will return 0 only if the end of a TSDU is being returned to the user.

On return, the data is expedited if T_EXPEDITED is set in flags. If T_MORE is also set, it indicates that the number of expedited bytes exceeded nbytes, a signal has interrupted the call, or that an entire ETSDU was not available (only for transport protocols that support fragmentation of ETSDUs). The rest of the ETSDU will be returned by subsequent calls to t_rcv() which will return with T_EXPEDITED set in flags. The end of the ETSDU is identified by the return of a t_rcv() call with T_EXPEDITED set and T_MORE cleared. If the entire ETSDU is not available it is possible for normal data fragments to be returned between the initial and final fragments of an ETSDU.

If a signal arrives, t_rcv() returns, giving the user any data currently available. If no data is available, t_rcv() returns –1, sets t_errno to TSYSERR and errno to EINTR. If some data is available, t_rcv() returns the number of bytes received and T_MORE is set in flags.

In synchronous mode, the only way for the user to be notified of the arrival of normal or expedited data is to issue this function or check for the T_DATA or T_EXDATA events using the t_look(3NSL) function. Additionally, the process can arrange to be notified by means of the EM interface.

Return Values On successful completion, t_rcv() returns the number of bytes received. Otherwise, it returns −1 on failure and t_errno is set to indicate the error.

Valid States T_DATAXFER, T_OUTREL.

Errors 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 function is not supported by the underlying transport provider.

TOUTSTATE   The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.

TPROTO   This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).

TSYSERR   A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

#include <tiuser.h>

Error Description
Values
The t_errno value that can be set by the XTI interface and cannot be set by the TLI interface is:

TPROTO

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe
Standard	See standards(5).

See Also   fcntl(2), t_getinfo(3NSL), t_look(3NSL), t_open(3NSL), t_snd(3NSL), attributes(5), standards(5)

Name   t_rcvconnect – receive the confirmation from a connection request

Synopsis   #include <xti.h>

int t_rcvconnect(int *fd*, struct t_call **call);

Description   This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

This function enables a calling transport user to determine the status of a previously sent connection request and is used in conjunction with t_connect(3NSL) to establish a connection in asynchronous mode, and to complete a synchronous t_connect(3NSL) call that was interrupted by a signal. The connection will be established on successful completion of this function.

The argument *fd* identifies the local transport endpoint where communication will be established, and *call* contains information associated with the newly established connection. The argument *call* points to a t_call structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

In *call*, *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, and *sequence* has no meaning for this function.

The *maxlen* field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *maxlen* can be set to zero, in which case no information to this specific argument is given to the user on the return from t_rcvconnect(). If *call* is set to NULL, no information at all is returned. By default, t_rcvconnect() 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 by means of t_open(3NSL) or fcntl(2), t_rcvconnect() executes in asynchronous mode, and reduces to a poll for existing connection confirmations. If none are available, t_rcvconnect() fails and returns immediately without waiting for the connection to be established. See TNODATA below. In this case, t_rcvconnect() must be called again to complete the connection establishment phase and retrieve the information returned in *call*.

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.

Valid States T_OUTCON.

Errors 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 connection information to be returned in *call* will be discarded. The provider's state, as seen by the user, will be changed to T_DATAXFER.

TLOOK   An asynchronous event has occurred on this transport connection and requires immediate attention.

TNODATA   O_NONBLOCK was set, but a connection confirmation has not yet arrived.

TNOTSUPPORT   This function is not supported by the underlying transport provider.

TOUTSTATE   The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.

TPROTO   This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).

TSYSERR   A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

#include<tiuser.h>

Error Description Values The t_errno value TPROTO can be set by the XTI interface but not by the TLI interface.

A t_errno value that this routine can return under different circumstances than its XTI counterpart is TBUFOVFLW. It can be returned even when the maxlen field of the corresponding buffer has been set to zero.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also   fcntl(2), t_accept(3NSL), t_alloc(3NSL), t_bind(3NSL), t_connect(3NSL),
t_listen(3NSL), t_open(3NSL), t_optmgmt(3NSL), attributes(5)

Name t_rcvdis – retrieve information from disconnection

Synopsis #include <xti.h>

int t_rcvdis(int *fd*, struct t_discon **discon*);

Description This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

This function is used to identify the cause of a disconnection and to retrieve any user data sent with the disconnection. The argument *fd* identifies the local transport endpoint where the connection existed, and *discon* points to a t_discon structure containing the following members:

```
struct netbuf udata;
int reason;
int sequence;
```

The field *reason* specifies the reason for the disconnection through a protocol-dependent reason code, *udata* identifies any user data that was sent with the disconnection, and *sequence* may identify an outstanding connection indication with which the disconnection is associated. The field *sequence* is only meaningful when t_rcvdis() is issued by a passive transport user who has executed one or more t_listen(3NSL) functions and is processing the resulting connection indications. If a disconnection indication occurs, *sequence* can be used to identify which of the outstanding connection indications is associated with the disconnection.

The *maxlen* field of *udata* may be set to zero, if the user does not care about incoming data. If, in addition, the user does not need to know the value of *reason* or *sequence*, *discon* may be set to NULL and any user data associated with the disconnection indication shall be discarded. However, if a user has retrieved more than one outstanding connection indication by means of t_listen(3NSL), and *discon* is a null pointer, the user will be unable to identify with which connection indication the disconnection is associated.

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.

Valid States T_DATAXFER, T_OUTCON, T_OUTREL, T_INREL, T_INCON(ocnt > 0).

Errors 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 *fd* is a passive endpoint with *ocnt* > 1, it remains in state T_INCON; otherwise, the endpoint state is set to T_IDLE.

TNODIS	No disconnection indication currently exists on the specified transport endpoint.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TOUTSTATE	The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).
TSYSERR	A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

#include <tiuser.h>

Error Description Values The t_errno values TPROTO and TOUTSTATE can be set by the XTI interface but not by the TLI interface.

A failure return, and a t_errno value that this routine can set under different circumstances than its XTI counterpart is TBUFOVFLW. It can be returned even when the maxlen field of the corresponding buffer has been set to zero.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also t_alloc(3NSL), t_connect(3NSL), t_listen(3NSL), t_open(3NSL), t_snddis(3NSL), attributes(5)

Name t_rcvrel – acknowledge receipt of an orderly release indication

Synopsis #include <xti.h>

int t_rcvrel(int *fd*);

Description This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

This function is used to receive an orderly release indication for the incoming direction of data transfer. The argument *fd* identifies the local transport endpoint where the connection exists. After receipt of this indication, the user may not attempt to receive more data by means of t_rcv(3NSL) or t_rcvv(). Such an attempt will fail with *t_error* set to TOUTSTATE. However, the user may continue to send data over the connection if t_sndrel(3NSL) 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 service type T_COTS_ORD on t_open(3NSL) or t_getinfo(3NSL). Any user data that may be associated with the orderly release indication is discarded when t_rcvrel() is called.

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.

Valid States T_DATAXFER, T_OUTREL.

Errors 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 function is not supported by the underlying transport provider.
TOUTSTATE	The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).
TSYSERR	A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

#include <tiuser.h>

Error Description Values The t_errno values that can be set by the XTI interface and cannot be set by the TLI interface are:

TPROTO
TOUTSTATE

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also t_getinfo(3NSL), t_open(3NSL), t_sndrel(3NSL), attributes(5)

Name t_rcvreldata – receive an orderly release indication or confirmation containing user data

Synopsis #include <xti.h>

int t_rcvreldata(int *fd*, struct t_discon **discon*);

Description This function is used to receive an orderly release indication for the incoming direction of data transfer and to retrieve any user data sent with the release. The argument *fd* identifies the local transport endpoint where the connection exists, and *discon* points to a t_discon structure containing the following members:

```
struct netbuf udata;
int reason;
int sequence;
```

After receipt of this indication, the user may not attempt to receive more data by means of t_rcv(3NSL) or t_rcvv(3NSL) Such an attempt will fail with *t_error* set to TOUTSTATE. However, the user may continue to send data over the connection if t_sndrel(3NSL) or t_sndreldata (3N) has not been called by the user.

The field *reason* specifies the reason for the disconnection through a protocol-dependent *reason code*, and *udata* identifies any user data that was sent with the disconnection; the field *sequence* is not used.

If a user does not care if there is incoming data and does not need to know the value of *reason*, *discon* may be a null pointer, and any user data associated with the disconnection will be discarded.

If *discon→udata.maxlen* is greater than zero and less than the length of the value, t_rcvreldata() fails with t_errno set to TBUFOVFLW.

This function is an optional service of the transport provider, only supported by providers of service type T_COTS_ORD. The flag T_ORDRELDATA in the *info→flag* field returned by t_open(3NSL) or t_getinfo(3NSL) indicates that the provider supports orderly release user data; when the flag is not set, this function behaves like t_rcvrel(3NSL) and no user data is returned.

This function may not be available on all systems.

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.

Valid States T_DATAXFER, T_OUTREL.

Errors 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, and the disconnection information to be returned in *discon* will be discarded. The provider state, as seen by the user, will be changed as if the data was successfully retrieved.
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	Orderly release is not supported by the underlying transport provider.
TOUTSTATE	The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).
TSYSERR	A system error has occurred during execution of this function.

Tli Compatibility In the TLI interface definition, no counterpart of this routine was defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also t_getinfo(3NSL), t_open(3NSL), t_sndreldata(3NSL), t_rcvrel(3NSL), t_sndrel(3NSL), attributes(5)

Notes The interfaces t_sndreldata(3NSL) and t_rcvreldata() are only for use with a specific transport called "minimal OSI," which is not available on the Solaris platform. These interfaces are not available for use in conjunction with Internet Transports (TCP or UDP).

Name   t_rcvudata – receive a data unit

Synopsis   `#include <xti.h>`

     `int t_rcvudata(int `*fd*`, struct t_unitdata *`*unitdata*`, int *`*flags*`);`

Description   This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the `tiuser.h` header file must be used. Refer to the `TLI COMPATIBILITY` section for a description of differences between the two interfaces.

This function is used in connectionless-mode to receive a data unit from another transport user. The argument *fd* identifies the local transport endpoint through which data will be received, *unitdata* holds information associated with the received data unit, and *flags* is set on return to indicate that the complete data unit was not received. The argument *unitdata* points to a `t_unitdata` structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

The *maxlen* field of *addr*, *opt* and *udata* must be set before calling this function to indicate the maximum size of the buffer for each. If the *maxlen* field of *addr* or *opt* is set to zero, no information is returned in the *buf* field of this parameter.

On return from this call, *addr* specifies the protocol address of the sending user, *opt* identifies options that were associated with this data unit, and *udata* specifies the user data that was received.

By default, t_rcvudata() operates in synchronous mode and will wait for a data unit to arrive if none is currently available. However, if `O_NONBLOCK` is set by means of t_open(3NSL) or fcntl(2), t_rcvudata() will execute in asynchronous mode and will fail 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 `T_MORE` will be set in *flags* on return to indicate that another t_rcvudata() should be called to retrieve the rest of the data unit. Subsequent calls to t_rcvudata() will return zero for the length of the address and options until the full data unit has been received.

If the call is interrupted, t_rcvudata() will return `EINTR` and no datagrams will have been removed from the endpoint.

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.

Valid States T_IDLE.

Errors 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 *unitdata* will be discarded.
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNODATA	O_NONBLOCK was set, but no data units are currently available from the transport provider.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TOUTSTATE	The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).
TSYSERR	A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

#include<tiuser.h>

Error Description Values The t_errno values that can be set by the XTI interface and cannot be set by the TLI interface are:

TPROTO
TOUTSTATE

A t_errno value that this routine can return under different circumstances than its XTI counterpart is TBUFOVFLW. It can be returned even when the maxlen field of the corresponding buffer has been set to zero.

Option Buffers   The format of the options in an opt buffer is dictated by the transport provider. Unlike the XTI interface, the TLI interface does not fix the buffer format.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also   fcntl(2), t_alloc(3NSL), t_open(3NSL), t_rcvuderr(3NSL), t_sndudata(3NSL), attributes(5)

Name t_rcvuderr – receive a unit data error indication

Synopsis #include <xti.h>

int t_rcvuderr(int *fd*, struct t_uderr **uderr*);

Description This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

This function 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. The argument *fd* identifies the local transport endpoint through which the error report will be received, and *uderr* points to a t_uderr structure containing the following members:

struct netbuf addr;
struct netbuf opt;
t_scalar_t error;

The *maxlen* field of *addr* and *opt* must be set before calling this function to indicate the maximum size of the buffer for each. If this field is set to zero for *addr* or *opt*, no information is returned in the *buf* field of this parameter.

On return from this call, the *addr* structure specifies the destination protocol address of the erroneous data unit, the *opt* structure identifies options that were associated with the data unit, and 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 t_rcvuderr() will simply clear the error indication without reporting any information to the user.

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.

Valid States T_IDLE.

Errors 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 error information to be returned in *uderr* will be discarded.

TNOTSUPPORT   This function is not supported by the underlying transport provider.

TNOUDERR	No unit data error indication currently exists on the specified transport endpoint.
TOUTSTATE	The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).
TSYSERR	A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

    #include <tiuser.h>

Error Description Values The t_errno values TPROTO and TOUTSTATE can be set by the XTI interface but not by the TLI interface.

A t_errno value that this routine can return under different circumstances than its XTI counterpart is TBUFOVFLW. It can be returned even when the maxlen field of the corresponding buffer has been set to zero.

Option Buffers The format of the options in an opt buffer is dictated by the transport provider. Unlike the XTI interface, the TLI interface does not fix the buffer format.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also t_rcvudata(3NSL), t_sndudata(3NSL), attributes(5)

Name t_rcvv – receive data or expedited data sent over a connection and put the data into one or more non-contiguous buffers

Synopsis #include <xti.h>

int t_rcvv(int *fd*, struct t_iovec **iov*, unsigned int *iovcount*, int *flags*);

Description This function receives either normal or expedited data. The argument *fd* identifies the local transport endpoint through which data will arrive, *iov* points to an array of buffer address/buffer size pairs (*iov_base*, *iov_len*). The t_rcvv() function receives data into the buffers specified by *iov*0.*iov_base*, *iov*1.*iov_base*, through *iov [iovcount-1].iov_base*, always filling one buffer before proceeding to the next.

Note that the limit on the total number of bytes available in all buffers passed:

iov(0).iov_len + . . + iov(iovcount-1).iov_len

may be constrained by implementation limits. If no other constraint applies, it will be limited by INT_MAX. In practice, the availability of memory to an application is likely to impose a lower limit on the amount of data that can be sent or received using scatter/gather functions.

The argument iovcount contains the number of buffers which is limited to T_IOV_MAX, which is an implementation-defined value of at least 16. If the limit is exceeded, the function will fail with TBADDATA.

The argument flags may be set on return from t_rcvv() and specifies optional flags as described below.

By default, t_rcvv() operates in synchronous mode and will wait for data to arrive if none is currently available. However, if O_NONBLOCK is set by means of t_open(3NSL) or fcntl(2), t_rcvv() will execute in asynchronous mode and will fail if no data is available. See TNODATA below.

On return from the call, if T_MORE is set in flags, this 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_rcvv() or t_rcv(3NSL) calls. In the asynchronous mode, or under unusual conditions (for example, the arrival of a signal or T_EXDATA event), the T_MORE flag may be set on return from the t_rcvv() call even when the number of bytes received is less than the total size of all the receive buffers. Each t_rcvv() with the T_MORE flag set indicates that another t_rcvv() must follow to get more data for the current TSDU. The end of the TSDU is identified by the return of a t_rcvv() 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* argument on return from t_open(3NSL) or t_getinfo(3NSL), the T_MORE flag is not meaningful and should be ignored. If the amount of buffer space passed in *iov* is greater than zero on the call to t_rcvv(), then t_rcvv() will return 0 only if the end of a TSDU is being returned to the user.

On return, the data is expedited if T_EXPEDITED is set in flags. If T_MORE is also set, it indicates that the number of expedited bytes exceeded nbytes, a signal has interrupted the call, or that an entire ETSDU was not available (only for transport protocols that support fragmentation of

ETSDUs). The rest of the ETSDU will be returned by subsequent calls to t_rcvv() which will return with T_EXPEDITED set in flags. The end of the ETSDU is identified by the return of a t_rcvv() call with T_EXPEDITED set and T_MORE cleared. If the entire ETSDU is not available it is possible for normal data fragments to be returned between the initial and final fragments of an ETSDU.

If a signal arrives, t_rcvv() returns, giving the user any data currently available. If no data is available, t_rcvv() returns −1, sets t_errno to TSYSERR and errno to EINTR. If some data is available, t_rcvv() returns the number of bytes received and T_MORE is set in flags.

In synchronous mode, the only way for the user to be notified of the arrival of normal or expedited data is to issue this function or check for the T_DATA or T_EXDATA events using the t_look(3NSL) function. Additionally, the process can arrange to be notified via the EM interface.

Return Values On successful completion, t_rcvv() returns the number of bytes received. Otherwise, it returns −1 on failure and t_errno is set to indicate the error.

Valid States T_DATAXFER, T_OUTREL.

Errors On failure, t_errno is set to one of the following:

TBADDATA	*iovcount* is greater than T_IOV_MAX.
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 function is not supported by the underlying transport provider.
TOUTSTATE	The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).
TSYSERR	A system error has occurred during execution of this function.

Tli Compatibility In the TLI interface definition, no counterpart of this routine was defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also   fcntl(2), t_getinfo(3NSL), t_look(3NSL), t_open(3NSL), t_rcv(3NSL), t_snd(3NSL), t_sndv(3NSL), attributes(5)

Name   t_rcvvudata – receive a data unit into one or more noncontiguous buffers

Synopsis   #include <xti.h>

```
int t_rcvvudata(int fd, struct t_unitdata *unitdata, struct t_iovec *iov,
    unsigned int iovcount, int *flags);
```

Description   This function is used in connectionless mode to receive a data unit from another transport user. The argument *fd* identifies the local transport endpoint through which data will be received, *unitdata* holds information associated with the received data unit, *iovcount* contains the number of non-contiguous udata buffers which is limited to T_IOV_MAX, which is an implementation-defined value of at least 16, and *flags* is set on return to indicate that the complete data unit was not received. If the limit on *iovcount* is exceeded, the function fails with TBADDATA. The argument *unitdata* points to a t_unitdata structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

The *maxlen* field of *addr* and *opt* must be set before calling this function to indicate the maximum size of the buffer for each. The *udata* field of t_unitdata is not used. The *iov_len* and *iov_base* fields of "*iov0*" through *iov [iovcount-1]* must be set before calling t_rcvvudata() to define the buffer where the userdata will be placed. If the maxlen field of *addr* or *opt* is set to zero then no information is returned in the *buf* field for this parameter.

On return from this call, *addr* specifies the protocol address of the sending user, *opt* identifies options that were associated with this data unit, and *iov*[0].*iov_base* through *iov [iovcount-1].iov_base* contains the user data that was received. The return value of t_rcvvudata() is the number of bytes of user data given to the user.

Note that the limit on the total number of bytes available in all buffers passed:

iov(0).iov_len + . . + iov(iovcount-1).iov_len)

may be constrained by implementation limits. If no other constraint applies, it will be limited by INT_MAX. In practice, the availability of memory to an application is likely to impose a lower limit on the amount of data that can be sent or received using scatter/gather functions.

By default, t_rcvvudata() operates in synchronous mode and waits for a data unit to arrive if none is currently available. However, if O_NONBLOCK is set by means of t_open(3NSL) or fcntl(2), t_rcvvudata() executes in asynchronous mode and fails if no data units are available.

If the buffers defined in the *iov[]* array are not large enough to hold the current data unit, the buffers will be filled and T_MORE will be set in flags on return to indicate that another t_rcvvudata() should be called to retrieve the rest of the data unit. Subsequent calls to t_rcvvudata() will return zero for the length of the address and options, until the full data unit has been received.

Return Values On successful completion, t_rcvvudata() returns the number of bytes received. Otherwise, it returns –1 on failure and t_errno is set to indicate the error.

Valid States T_IDLE.

Errors On failure, t_errno is set to one of the following:

TBADDATA	*iovcount* is greater than T_IOV_MAX.
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 *unitdata* will be discarded.
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNODATA	O_NONBLOCK was set, but no data units are currently available from the transport provider.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TOUTSTATE	The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).
TSYSERR	A system error has occurred during execution of this function.

Tli Compatibility In the TLI interface definition, no counterpart of this routine was defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also fcntl(2), t_alloc(3NSL), t_open(3NSL), t_rcvudata(3NSL), t_rcvuderr(3NSL), t_sndudata(3NSL), t_sndvudata(3NSL), attributes(5)

Name   t_snd – send data or expedited data over a connection

Synopsis   #include <xti.h>

int t_snd(int *fd*, void *buf*, unsigned int *nbytes*, int *flags*);

Description   This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

This function is used to send either normal or expedited data. The argument *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, and *flags* specifies any optional flags described below:

T_EXPEDITED   If set in *flags*, the data will be sent as expedited data and will be subject to the interpretations of the transport provider.

T_MORE   If set in *flags*, this 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() with the T_MORE flag set indicates that another t_snd() 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* argument on return from t_open(3NSL) or t_getinfo(3NSL), the T_MORE flag is not meaningful and will be 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; that is, when the T_MORE flag is not set. Some transport providers also forbid zero-length TSDUs and ETSDUs.

T_PUSH   If set in *flags*, requests that the provider transmit all data that it has accumulated but not sent. The request is a local action on the provider and does not affect any similarly named protocol flag (for example, the TCP PUSH flag). This effect of setting this flag is protocol-dependent, and it may be ignored entirely by transport providers which do not support the use of this feature.

Note that the communications provider is free to collect data in a send buffer until it accumulates a sufficient amount for transmission.

By default, t_snd() 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 O_NONBLOCK is set by means of t_open(3NSL) or fcntl(2), t_snd() will execute in asynchronous mode, and will fail immediately if there are flow control restrictions. The process can arrange to be informed when the flow control restrictions are cleared by means of either t_look(3NSL) or the EM interface.

On successful completion, t_snd() returns the number of bytes (octets) accepted by the communications provider. Normally this will equal the number of octets specified in nbytes. However, if O_NONBLOCK is set or the function is interrupted by a signal, it is possible that only part of the data has actually been accepted by the communications provider. In this case, t_snd() returns a value that is less than the value of nbytes. If t_snd() is interrupted by a signal before it could transfer data to the communications provider, it returns –1 with t_errno set to TSYSERR and errno set to EINTR.

If nbytes is zero and sending of zero bytes is not supported by the underlying communications service, t_snd() returns –1 with t_errno set to TBADDATA.

The size of each TSDU or ETSDU must not exceed the limits of the transport provider as specified by the current values in the TSDU or ETSDU fields in the *info* argument returned by t_getinfo(3NSL).

The error TLOOK is returned for asynchronous events. It is required only for an incoming disconnect event but may be returned for other events.

Return Values   On successful completion, t_snd() 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 if the number of bytes accepted by the communications provider is less than the number of bytes requested, this may either indicate that O_NONBLOCK is set and the communications provider is blocked due to flow control, or that O_NONBLOCK is clear and the function was interrupted by a signal.

Errors   On failure, t_errno is set to one of the following:

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.

> - 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 WARNINGS, below.

TBADF   The specified file descriptor does not refer to a transport endpoint.

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.

TLOOK   An asynchronous event has occurred on this transport endpoint.

TNOTSUPPORT   This function is not supported by the underlying transport provider.

TOUTSTATE   The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.

TPROTO   This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).

TSYSERR   A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

#include <tiuser.h>

Error Description Values The t_errno values that can be set by the XTI interface and cannot be set by the TLI interface are:

TPROTO
TLOOK
TBADFLAG
TOUTSTATE

The t_errno values that this routine can return under different circumstances than its XTI counterpart are:

TBADDATA

In the TBADDATA error cases described above, TBADDATA is returned, only for illegal zero byte TSDU (ETSDU) send attempts.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also fcntl(2), t_getinfo(3NSL), t_look(3NSL), t_open(3NSL), t_rcv(3NSL), attributes(5)

Warnings 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 XTI. 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 XTI, t_snd() fails with TBADDATA.

Name t_snddis – send user-initiated disconnection request

Synopsis #include <xti.h>

int t_snddis(int *fd*, const struct t_call **call);

Description This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

This function is used to initiate an abortive release on an already established connection, or to reject a connection request. The argument *fd* identifies the local transport endpoint of the connection, and *call* specifies information associated with the abortive release. The argument *call* 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 *call* have different semantics, depending on the context of the call to t_snddis(). When rejecting a connection request, *call* must be non-null and contain a valid value of *sequence* to uniquely identify the rejected connection 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 *call* are ignored. In all other cases, *call* need only be used when data is being sent with the disconnection 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 *call* may be a null pointer.

The *udata* structure 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 *discon* field, of the *info* argument of t_open(3NSL) or t_getinfo(3NSL). If the *len* field of *udata* is zero, no data will be sent to the remote user.

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.

Valid States T_DATAXFER, T_OUTCON, T_OUTREL, T_INREL, T_INCON(ocnt > 0).

Errors On failure, t_errno is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TBADDATA	The amount of user data specified was not within the bounds allowed by the transport provider.
TBADSEQ	An invalid sequence number was specified, or a null *call* pointer was specified, when rejecting a connection request.

TLOOK	An asynchronous event, which requires attention, has occurred.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TOUTSTATE	The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).
TSYSERR	A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

```
#include <tiuser.h>
```

Error Description Values The t_errno value TPROTO can be set by the XTI interface but not by the TLI interface.

Option Buffers The format of the options in an opt buffer is dictated by the transport provider. Unlike the XTI interface, the TLI interface does not fix the buffer format.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also t_connect(3NSL), t_getinfo(3NSL), t_listen(3NSL), t_open(3NSL), t_snd(3NSL), attributes(5)

Warnings t_snddis() is an abortive disconnection. Therefore a t_snddis() issued on a connection endpoint may cause data previously sent by means of t_snd(3NSL), or data not yet received, to be lost, even if an error is returned.

Name t_sndrel – initiate an orderly release

Synopsis #include <xti.h>

int t_sndrel(int *fd*);

Description This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

For transport providers of type T_COTS_ORD, this function is used to initiate an orderly release of the outgoing direction of data transfer and indicates to the transport provider that the transport user has no more data to send. The argument *fd* identifies the local transport endpoint where the connection exists. After calling t_sndrel(), 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. For transport providers of types other than T_COTS_ORD, this function fails with error TNOTSUPPORT.

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.

Valid States T_DATAXFER, T_INREL.

Errors 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 function at this time.
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TOUTSTATE	The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).
TSYSERR	A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header   The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They
should use the header:

#include <tiuser.h>

Error Description   The t_errno values that can be set by the XTI interface and cannot be set by the TLI interface
Values   are:

TPROTO
TLOOK
TOUTSTATE

Notes   Whenever this function fails with t_error set to TFLOW, O_NONBLOCK must have been set.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also   t_error(3NSL), t_getinfo(3NSL), t_open(3NSL), t_rcvrel(3NSL), attributes(5)

Name t_sndreldata – initiate or respond to an orderly release with user data

Synopsis #include <xti.h>

int t_sndreldata(int *fd*, struct t_discon **discon);

Description This function is used to initiate an orderly release of the outgoing direction of data transfer and to send user data with the release. The argument *fd* identifies the local transport endpoint where the connection exists, and *discon* points to a t_discon structure containing the following members:

```
struct netbuf udata;
int reason;
int sequence;
```

After calling t_sndreldata(), 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.

The field *reason* specifies the reason for the disconnection through a protocol-dependent *reason code*, and *udata* identifies any user data that is sent with the disconnection; the field *sequence* is not used.

The *udata* structure 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 *discon* field of the *info* argument of t_open(3NSL) or t_getinfo(3NSL). If the *len* field of *udata* is zero or if the provider did not return T_ORDRELDATA in the t_open(3NSL) flags, no data will be sent to the remote user.

If a user does not wish to send data and reason code to the remote user, the value of *discon* may be a null pointer.

This function is an optional service of the transport provider, only supported by providers of service type T_COTS_ORD. The flag T_ORDRELDATA in the *info→flag* field returned by t_open(3NSL) or t_getinfo(3NSL) indicates that the provider supports orderly release user data.

This function may not be available on all systems.

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.

Valid States T_DATAXFER, T_INREL.

Errors 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, or user data was supplied and the provider did not return T_ORDRELDATA in the t_open(3NSL) flags.

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 function at this time.
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNOTSUPPORT	Orderly release is not supported by the underlying transport provider.
TOUTSTATE	The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).
TSYSERR	A system error has occurred during execution of this function.

Tli Compatibility In the TLI interface definition, no counterpart of this routine was defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also t_getinfo(3NSL), t_open(3NSL), t_rcvrel(3NSL), t_rcvreldata(3NSL), t_sndrel(3NSL), attributes(5)

Notes The interfaces t_sndreldata() and t_rcvreldata(3NSL) are only for use with a specific transport called "minimal OSI," which is not available on the Solaris platform. These interfaces are not available for use in conjunction with Internet Transports (TCP or UDP).

Name   t_sndudata – send a data unit

Synopsis   #include <xti.h>

int t_sndudata(int *fd*, const struct t_unitdata **unitdata*);

Description   This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

This function is used in connectionless-mode to send a data unit to another transport user. The argument *fd* identifies the local transport endpoint through which data will be sent, and *unitdata* points to a t_unitdata structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

In *unitdata*, *addr* specifies the protocol address of the destination user, *opt* identifies options that the user wants associated with this request, and *udata* specifies the user data to be sent. 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 uses the option values currently set for the communications endpoint.

If the *len* field of *udata* is zero, and sending of zero octets is not supported by the underlying transport service, the t_sndudata() will return –1 with t_errno set to TBADDATA.

By default, t_sndudata() 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 O_NONBLOCK is set by means of t_open(3NSL) or fcntl(2), t_sndudata() will execute in asynchronous mode and will fail under such conditions. The process can arrange to be notified of the clearance of a flow control restriction by means of either t_look(3NSL) or the EM interface.

If the amount of data specified in *udata* exceeds the TSDU size as returned in the *tsdu* field of the *info* argument of t_open(3NSL) or t_getinfo(3NSL), a TBADDATA error will be generated. If t_sndudata() is called before the destination user has activated its transport endpoint (see t_bind(3NSL)), 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 *t_rcvuderr*. Therefore, an application must be prepared to receive these errors in both of these ways.

If the call is interrupted, t_sndudata() will return EINTR and the datagram will not be sent.

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.

Valid States T_IDLE.

Errors 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* argument, 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 this transport endpoint.

TNOTSUPPORT   This function is not supported by the underlying transport provider.

TOUTSTATE   The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.

TPROTO   This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).

TSYSERR   A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

#include <tiuser.h>

Error Description Values The t_errno values that can be set by the XTI interface and cannot be set by the TLI interface are:

TPROTO
TBADADDR

TBADOPT
TLOOK
TOUTSTATE

Notes   Whenever this function fails with t_error set to TFLOW, O_NONBLOCK must have been set.

Option Buffers   The format of the options in an opt buffer is dictated by the transport provider. Unlike the XTI interface, the TLI interface does not fix the buffer format.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also   fcntl(2), t_alloc(3NSL), t_bind(3NSL), t_error(3NSL), t_getinfo(3NSL), t_look(3NSL), t_open(3NSL), t_rcvudata(3NSL), t_rcvuderr(3NSL), attributes(5)

Name   t_sndv – send data or expedited data, from one or more non-contiguous buffers, on a
connection

Synopsis   #include <xti.h>

int t_sndv(int *fd*, const struct t_iovec **iov*, unsigned int *iovcount*, int *flags*);

Description   This function is used to send either normal or expedited data. The argument *fd* identifies the
local transport endpoint over which data should be sent, *iov* points to an array of buffer
address/buffer length pairs. t_sndv() sends data contained in buffers *iov*0 , *iov*1 , through *iov*
[iovcount-1] . *iovcount* contains the number of non-contiguous data buffers which is limited
to T_IOV_MAX, an implementation-defined value of at least 16. If the limit is exceeded, the
function fails with TBADDATA.

iov(0).iov_len + . . + iov(iovcount-1).iov_len)

Note that the limit on the total number of bytes available in all buffers passed:

may be constrained by implementation limits. If no other constraint applies, it will be limited
by INT_MAX. In practice, the availability of memory to an application is likely to impose a lower
limit on the amount of data that can be sent or received using scatter/gather functions.

The argument *flags* specifies any optional flags described below:

T_EXPEDITED   If set in *flags*, the data will be sent as expedited data and will be subject to the
interpretations of the transport provider.

T_MORE   If set in *flags*, this 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_sndv() calls. Each t_sndv() with
the T_MORE flag set indicates that another t_sndv() or t_snd(3NSL) will
follow with more data for the current TSDU (or ETSDU).

The end of the TSDU (or ETSDU) is identified by a t_sndv() 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* argument on return from
t_open(3NSL) or t_getinfo(3NSL), the T_MORE flag is not meaningful and will be 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, that is, when the T_MORE flag is not set. Some
transport providers also forbid zero-length TSDUs and ETSDUs.

If set in *flags*, requests that the provider transmit all data that it has accumulated but not sent.
The request is a local action on the provider and does not affect any similarly named protocol

flag (for example, the TCP PUSH flag). This effect of setting this flag is protocol-dependent, and it may be ignored entirely by transport providers which do not support the use of this feature.

The communications provider is free to collect data in a send buffer until it accumulates a sufficient amount for transmission.

By default, t_sndv() 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 O_NONBLOCK is set by means of t_open(3NSL) or fcntl(2), t_sndv() executes in asynchronous mode, and will fail immediately if there are flow control restrictions. The process can arrange to be informed when the flow control restrictions are cleared via either t_look(3NSL) or the EM interface.

On successful completion, t_sndv() returns the number of bytes accepted by the transport provider. Normally this will equal the total number of bytes to be sent, that is,

*(iov*0.iov_len + .. + iov[iovcount-1].iov_len)

However, the interface is constrained to send at most INT_MAX bytes in a single send. When t_sndv() has submitted INT_MAX (or lower constrained value, see the note above) bytes to the provider for a single call, this value is returned to the user. However, if O_NONBLOCK is set or the function is interrupted by a signal, it is possible that only part of the data has actually been accepted by the communications provider. In this case, t_sndv() returns a value that is less than the value of nbytes. If t_sndv() is interrupted by a signal before it could transfer data to the communications provider, it returns –1 with t_errno set to TSYSERR and errno set to EINTR.

If the number of bytes of data in the *iov* array is zero and sending of zero octets is not supported by the underlying transport service, t_sndv() returns –1 with t_errno set to TBADDATA.

The size of each TSDU or ETSDU must not exceed the limits of the transport provider as specified by the current values in the TSDU or ETSDU fields in the *info* argument returned by t_getinfo(3NSL).

The error TLOOK is returned for asynchronous events. It is required only for an incoming disconnect event but may be returned for other events.

Return Values   On successful completion, t_sndv() 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 synchronous mode, if more than INT_MAX bytes of data are passed in the *iov* array, only the first INT_MAX bytes will be passed to the provider.

If the number of bytes accepted by the communications provider is less than the number of bytes requested, this may either indicate that O_NONBLOCK is set and the communications provider is blocked due to flow control, or that O_NONBLOCK is clear and the function was interrupted by a signal.

Valid States T_DATAXFER, T_INREL.

Errors On failure, t_errno is set to one of the following:

TBADDATA   Illegal amount of data:

TBADF   The specified file descriptor does not refer to a transport endpoint.

- 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.

- 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 WARNINGS, below.

- *iovcount* is greater than T_IOV_MAX.

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.

TLOOK   An asynchronous event has occurred on this transport endpoint.

TNOTSUPPORT   This function is not supported by the underlying transport provider.

TOUTSTATE   The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.

TPROTO   This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).

TSYSERR   A system error has occurred during execution of this function.

Tli Compatibility In the TLI interface definition, no counterpart of this routine was defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also   t_getinfo(3NSL), t_open(3NSL), t_rcvv(3NSL) t_rcv(3NSL), t_snd(3NSL), attributes(5)

Warnings   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_sndv() or t_snd(3NSL) calls, then the different data may be intermixed.

Multiple sends which exceed the maximum TSDU or ETSDU size may not be discovered by XTI. 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 XTI, t_sndv() fails with TBADDATA.

Name   t_sndvudata – send a data unit from one or more noncontiguous buffers

Synopsis   #include <xti.h>

int t_sndvudata(int *fd*, struct t_unitdata *unitdata*, struct t_iovec *iov*,
      unsigned int *iovcount*);

Description   This function is used in connectionless mode to send a data unit to another transport user.
The argument *fd* identifies the local transport endpoint through which data will be sent,
iovcount contains the number of non-contiguous *udata* buffers and is limited to an
implementation-defined value given by T_IOV_MAX which is at least 16, and *unitdata* points to
a t_unitdata structure containing the following members:

struct netbuf addr;
struct netbuf opt;
struct netbuf udata;

If the limit on *iovcount* is exceeded, the function fails with TBADDATA.

In unitdata, *addr* specifies the protocol address of the destination user, and *opt* identifies
options that the user wants associated with this request. The *udata* field is not used. 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.

The data to be sent is identified by *iov*[0] through *iov [iovcount-1]*.

Note that the limit on the total number of bytes available in all buffers passed:

iov(0).iov_len + . . + iov(iovcount-1).iov_len

may be constrained by implementation limits. If no other constraint applies, it will be limited
by INT_MAX. In practice, the availability of memory to an application is likely to impose a lower
limit on the amount of data that can be sent or received using scatter/gather functions.

By default, t_sndvudata() 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 O_NONBLOCK is set by means of t_open(3NSL) or fcntl(2),
t_sndvudata() executes in asynchronous mode and will fail under such conditions. The
process can arrange to be notified of the clearance of a flow control restriction by means of
either t_look(3NSL) or the EM interface.

If the amount of data specified in *iov0* through *iov [iovcount-1]* exceeds the TSDU size as
returned in the *tsdu* field of the *info* argument of t_open(3NSL) or t_getinfo(3NSL), or is
zero and sending of zero octets is not supported by the underlying transport service, a
TBADDATA error is generated. If t_sndvudata() is called before the destination user has
activated its transport endpoint (see t_bind(3NSL)), the data unit may be discarded.

If it is not possible for the transport provider to immediately detect the conditions that cause the errors TBADADDR and TBADOPT, these errors will alternatively be returned by t_rcvuderr(3NSL). An application must therefore be prepared to receive these errors in both of these ways.

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.

Valid States T_IDLE.

Errors 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* argument, or a send of a zero byte TSDU is not supported by the provider.

- *iovcount* is greater than T_IOV_MAX.

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 i 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 this transport endpoint.

TNOTSUPPORT   This function is not supported by the underlying transport provider.

TOUTSTATE   The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.

TPROTO   This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).

TSYSERR   A system error has occurred during execution of this function.

Tli Compatibility In the TLI interface definition, no counterpart of this routine was defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also fcntl(2), t_alloc(3NSL), t_open(3NSL), t_rcvudata(3NSL), t_rcvvudata(3NSL)
t_rcvuderr(3NSL), t_sndudata(3NSL), attributes(5)

Name   t_strerror – produce an error message string

Synopsis   #include <xti.h>

const char *t_strerror(int *errnum*);

Description   This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

The t_strerror() function maps the error number in *errnum* that corresponds to an XTI error to a language-dependent error message string and returns a pointer to the string. The string pointed to will not be modified by the program, but may be overwritten by a subsequent call to the *t_strerror* function. The string is not terminated by a newline character. The language for error message strings written by t_strerror() is that of the current locale. If it is English, the error message string describing the value in t_errno may be derived from the comments following the t_errno codes defined in <xti.h>. 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.

Valid States   ALL - apart from T_UNINIT.

Return Values   The function t_strerror() returns a pointer to the generated message string.

Tli Compatibility   The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header   The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

#include <tiuser.h>

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also t_errno(3NSL), t_error(3NSL), attributes(5)

Name t_sync – synchronize transport library

Synopsis #include <xti.h>

int t_sync(int *fd*);

Description This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

For the transport endpoint specified by *fd*, t_sync() synchronizes the data structures managed by the transport library with information from the underlying transport provider. In doing so, it can convert an uninitialized file descriptor (obtained by means of a open(2), dup(2) or as a result of a fork(2) and exec(2)) to an initialized transport endpoint, assuming that the file descriptor referenced a transport endpoint, by updating and allocating the necessary library data structures. This function 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(2), 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 function t_sync() 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() is issued.

If the transport endpoint is undergoing a state transition when t_sync() is called, the function will fail.

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).

Errors 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* 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 XTI and the transport provider for which there is no other suitable XTI error (t_errno).
TSTATECHNG	The transport endpoint is undergoing a state change.
TSYSERR	A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h. TLI interfaces should *not* use this header. They should use the header:

#include <tiuser.h>

Error Description Values The t_errno value that can be set by the XTI interface and cannot be set by the TLI interface is:

TPROTO

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also dup(2), exec(2), fork(2), open(2), attributes(5)

Name t_sysconf – get configurable XTI variables

Synopsis #include <xti.h>

int t_sysconf(int *name*);

Description The t_sysconf() function provides a method for the application to determine the current value of configurable and implementation-dependent XTI limits or options.

The *name* argument represents the XTI system variable to be queried. The following table lists the minimal set of XTI system variables from <xti.h> that can be returned by t_sysconf(), and the symbolic constants, defined in <xti.h> that are the corresponding values used for *name*.

Variable	Value of Name
T_IOV_MAX	_SC_T_IOV_MAX

Return Values If *name* is valid, t_sysconf() returns the value of the requested limit/option, which might be –1, and leaves t_errno unchanged. Otherwise, a value of –1 is returned and t_errno is set to indicate an error.

Valid States All.

Errors On failure, t_errno is set to the following:

TBADFLAG *name* has an invalid value.

Tli Compatibility In the TLI interface definition, no counterpart of this routine was defined.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also sysconf(3C), t_rcvv(3NSL), t_rcvvudata(3NSL), t_sndv(3NSL), t_sndvudata(3NSL), attributes(5)

Name t_unbind – disable a transport endpoint

Synopsis #include <xti.h>

int t_unbind(int *fd*);

Description The This routine is part of the XTI interfaces which evolved from the TLI interfaces. XTI represents the future evolution of these interfaces. However, TLI interfaces are supported for compatibility. When using a TLI routine that has the same name as an XTI routine, the tiuser.h header file must be used. Refer to the TLI COMPATIBILITY section for a description of differences between the two interfaces.

t_unbind() function disables the transport endpoint specified by *fd* which was previously bound by t_bind(3NSL). On completion of this call, no further data or events destined for this transport endpoint will be accepted by the transport provider. An endpoint which is disabled by using t_unbind() can be enabled by a subsequent call to t_bind(3NSL).

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.

Valid States T_IDLE.

Errors 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.
TOUTSTATE	The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).
TSYSERR	A system error has occurred during execution of this function.

Tli Compatibility The XTI and TLI interface definitions have common names but use different header files. This, and other semantic differences between the two interfaces are described in the subsections below.

Interface Header The XTI interfaces use the header file, xti.h . TLI interfaces should *not* use this header. They should use the header:

#include <tiuser.h>

Error Description Values   The t_errno value that can be set by the XTI interface and cannot be set by the TLI interface is:

TPROTO

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also   t_bind(3NSL), attributes(5)

Name   TXTRecordCreate, TXTRecordDeallocate, TXTRecordSetValue, TXTRecordRemoveValue,
TXTRecordGetLength, TXTRecordGetBytesPtr, TXTRecordContainsKey,
TXTRecordGetValuePtr, TXTRecordGetCount, TXTRecordGetItemAtIndex – DNS TXT
record manipulation functions

Synopsis   cc [*flag* ...] *file* ... -ldns_sd [*library* ...]
#include <dns_sd.h>

```
void TXTRecordCreate(TXTRecordRef *txtRecord, uint16_t bufferLen,
    void *buffer);

void TXTRecordDeallocate(TXTRecordRef*txtRecord);

DNSServiceErrorType txtRecord(TXTRecordRef *txtRecord,
    const char *key, uint8_t valueSize, const void *value);

DNSServiceErrorType TXTRecordRemoveValue(TXTRecordRef *txtRecord,
    const char *key);

uint16_t TXTRecordGetLength(const TXTRecordRef *txtRecord);

const void *TXTRecordGetBytesPtr(const TXTRecordRef *txtRecord);

int *TXTRecordContainsKey(uint16_t *txtLen,
    const void *txtRecord, const char *key);

const void *TXTRecordGetValuePtr(uint16_t *txtLen,
    const void *txtRecord, const char *key,
    uint8_t *valueLen);

uint16_t *TXTRecordGetCount(uint16_t *txtLen,
    const void *txtRecord);

DNSServiceErrorType TXTRecordGetItemAtIndex(uint16_t *txtLen,
    const void *txtRecord, uint16_t *index,
    uint16_t *keyBufLen, char *key,
    uint8_t *valueLen, const void **value);
```

Description   These functions in the libdns_sd library allow applications to create and to manipulate TXT
resource records. TXT resource records enable applications to include service specific
information, other than a host name and port number, as part of the service registration.

Attributes   See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also   attributes(5)

Name xdr – library routines for external data representation

Description XDR routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Data for remote procedure calls (RPC) are transmitted using these routines.

Index to Routines The following table lists XDR routines and the manual reference pages on which they are described:

XDR Routine	Manual Reference Page
xdr_array	xdr_complex(3NSL)
xdr_bool	xdr_simple(3NSL)
xdr_bytes	xdr_complex(3NSL)
xdr_char	xdr_simple(3NSL)
xdr_control	xdr_admin(3NSL)
xdr_destroy	xdr_create(3NSL)
xdr_double	xdr_simple(3NSL)
xdr_enum	xdr_simple(3NSL)
xdr_float	xdr_simple(3NSL)
xdr_free	xdr_simple(3NSL)
xdr_getpos	xdr_admin(3NSL)
xdr_hyper	xdr_simple(3NSL)
xdr_inline	xdr_admin(3NSL)
xdr_int	xdr_simple(3NSL)
xdr_long	xdr_simple(3NSL)
xdr_longlong_t	xdr_simple(3NSL)
xdr_opaque	xdr_complex(3NSL)
xdr_pointer	xdr_complex(3NSL)
xdr_quadruple	xdr_simple(3NSL)
xdr_reference	xdr_complex(3NSL)
xdr_setpos	xdr_admin(3NSL)
xdr_short	xdr_simple(3NSL)
xdr_sizeof	xdr_admin(3NSL)

xdr_string	xdr_complex(3NSL)
xdr_u_char	xdr_simple(3NSL)
xdr_u_hyper	xdr_simple(3NSL)
xdr_u_int	xdr_simple(3NSL)
xdr_u_long	xdr_simple(3NSL)
xdr_u_longlong_t	xdr_simple(3NSL)
xdr_u_short	xdr_simple(3NSL)
xdr_union	xdr_complex(3NSL)
xdr_vector	xdr_complex(3NSL)
xdr_void	xdr_simple(3NSL)
xdr_wrapstring	xdr_complex(3NSL)
xdrmem_create	xdr_create(3NSL)
xdrrec_create	xdr_create(3NSL)
xdrrec_endofrecord	xdr_admin(3NSL)
xdrrec_eof	xdr_admin(3NSL)
xdrrec_readbytes	xdr_admin(3NSL)
xdrrec_skiprecord	xdr_admin(3NSL)
xdrstdio_create	xdr_create(3NSL)

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also rpc(3NSL), xdr_admin(3NSL), xdr_complex(3NSL), xdr_create(3NSL), xdr_simple(3NSL), attributes(5)

Name xdr_admin, xdr_control, xdr_getpos, xdr_inline, xdrrec_endofrecord, xdrrec_eof, xdrrec_readbytes, xdrrec_skiprecord, xdr_setpos, xdr_sizeof – library routines for external data representation

Description XDR library routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data.

These routines deal specifically with the management of the XDR stream.

Routines See rpc(3NSL) for the definition of the XDR data structure. Note that any buffers passed to the XDR routines must be properly aligned. It is suggested either that malloc(3C) be used to allocate these buffers, or that the programmer insure that the buffer address is divisible evenly by four.

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

bool_t xdr_control(XDR **xdrs*, int *req*, void **info);
    A function macro to change or retrieve various information about an XDR stream. *req* indicates the type of operation and *info* is a pointer to the information. The supported values of *req* is XDR_GET_BYTES_AVAIL and its argument type is xdr_bytesrec *. They return the number of bytes left unconsumed in the stream and a flag indicating whether or not this is the last fragment.

uint_t xdr_getpos(const XDR **xdrs);
    A macro that invokes the get-position routine associated with the XDR stream, *xdrs*. The routine returns an unsigned integer, which indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this. Therefore, applications written for portability should not depend on this feature.

long *xdr_inline(XDR **xdrs, const int *len*);
    A macro that invokes the in-line routine associated with the XDR stream, *xdrs*. The routine returns a pointer to a contiguous piece of the stream's buffer; *len* is the byte length of the desired buffer. Note: pointer is cast to long *.

    Warning: xdr_inline() may return NULL (0) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency, and applications written for portability should not depend on this feature.

bool_t xdrrec_endofrecord(XDR *xdrs, int *sendnow*);
    This routine can be invoked only on streams created by xdrrec_create(). See xdr_create(3NSL). The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if *sendnow* is non-zero. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdrrec_eof(XDR *_xdrs_);
    This routine can be invoked only on streams created by xdrrec_create(). After
    consuming the rest of the current record in the stream, this routine returns TRUE if there is
    no more data in the stream's input buffer. It returns FALSE if there is additional data in the
    stream's input buffer.

int xdrrec_readbytes(XDR *_xdrs_, caddr_t _addr_, uint_t _nbytes_);
    This routine can be invoked only on streams created by xdrrec_create(). It attempts to
    read _nbytes_ bytes from the XDR stream into the buffer pointed to by _addr_. Upon success
    this routine returns the number of bytes read. Upon failure, it returns −1. A return value of
    0 indicates an end of record.

bool_t xdrrec_skiprecord(XDR *_xdrs_);
    This routine can be invoked only on streams created by xdrrec_create(). See
    xdr_create(3NSL). It tells the XDR implementation that the rest of the current record in
    the stream's input buffer should be discarded. This routine returns TRUE if it succeeds,
    FALSE otherwise.

bool_t xdr_setpos(XDR *_xdrs_, const uint_t _pos_);
    A macro that invokes the set position routine associated with the XDR stream _xdrs_. The
    parameter _pos_ is a position value obtained from xdr_getpos(). This routine returns TRUE if
    the XDR stream was repositioned, and FALSE otherwise.

    Warning: it is difficult to reposition some types of XDR streams, so this routine may fail
    with one type of stream and succeed with another. Therefore, applications written for
    portability should not depend on this feature.

unsigned long xdr_sizeof(xdrproc_t _func_, void *_data_);
    This routine returns the number of bytes required to encode _data_ using the XDR filter
    function _func_, excluding potential overhead such as RPC headers or record markers. 0 is
    returned on error. This information might be used to select between transport protocols, or
    to determine the buffer size for various lower levels of RPC client and server creation
    routines, or to allocate storage when XDR is used outside of the RPC subsystem.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   malloc(3C), rpc(3NSL), xdr_complex(3NSL), xdr_create(3NSL), xdr_simple(3NSL),
attributes(5)

Name xdr_complex, xdr_array, xdr_bytes, xdr_opaque, xdr_pointer, xdr_reference, xdr_string, xdr_union, xdr_vector, xdr_wrapstring – library routines for external data representation

Description XDR library routines allow C programmers to describe complex data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data. These routines are the XDR library routines for complex data structures. They require the creation of XDR streams. See xdr_create(3NSL).

Routines See rpc(3NSL) for the definition of the XDR data structure. Note that any buffers passed to the XDR routines must be properly aligned. It is suggested either that malloc() be used to allocate these buffers, or that the programmer insure that the buffer address is divisible evenly by four.

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

bool_t xdr_array(XDR *xdrs, caddr_t *arrp, uint_t *sizep, const uint_t maxsize, const uint_t elsize, const xdrproc_t elproc);
  xdr_array() translates between variable-length arrays and their corresponding external representations. The parameter *arrp* is the address of the pointer to the array, while *sizep* is the address of the element count of the array; this element count cannot exceed *maxsize*. The parameter *elsize* is the size of each of the array's elements, and *elproc* is an XDR routine that translates between the array elements' C form and their external representation. If **arrp* is NULL when decoding, xdr_array() allocates memory and **arrp* points to it. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_bytes(XDR *xdrs, char **sp, uint_t *sizep, const uint_t maxsize);
  xdr_bytes() translates between counted byte strings and their external representations. The parameter *sp* is the address of the string pointer. The length of the string is located at address *sizep*; strings cannot be longer than *maxsize*. If **sp* is NULL when decoding, xdr_bytes() allocates memory and **sp* points to it. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_opaque(XDR *xdrs, caddr_t cp, const uint_t cnt);
  xdr_opaque() translates between fixed size opaque data and its external representation. The parameter cp is the address of the opaque object, and *cnt* is its size in bytes. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_pointer(XDR *xdrs, char **objpp, uint_t objsize, const xdrproc_t xdrobj);
  Like xdr_reference() except that it serializes null pointers, whereas xdr_reference() does not. Thus, xdr_pointer() can represent recursive data structures, such as binary trees or linked lists. If **objpp* is NULL when decoding, xdr_pointer() allocates memory and **objpp* points to it.

bool_t xdr_reference(XDR *xdrs, caddr_t *pp, uint_t size, const xdrproc_t proc);
  xdr_reference() provides pointer chasing within structures. The parameter *pp* is the address of the pointer; size is the sizeof the structure that **pp* points to; and proc is an XDR procedure that translates the structure between its C form and its external representation. If **pp* is NULL when decoding, xdr_reference() allocates memory and **pp* points to it. This routine returns 1 if it succeeds, 0 otherwise.

Warning: this routine does not understand null pointers. Use xdr_pointer() instead.

bool_t xdr_string(XDR *xdrs, char **sp, const uint_t maxsize);

xdr_string() translates between C strings and their corresponding external representations. Strings cannot be longer than maxsize. Note: sp is the address of the string's pointer. If *sp is NULL when decoding, xdr_string() allocates memory and *sp points to it. This routine returns TRUE if it succeeds, FALSE otherwise. Note: xdr_string() can be used to send an empty string but not a null string.

bool_t xdr_union(XDR *xdrs, enum_t *dscmp, char *unp, const struct xdr_discrim *choices, const xdrproc_t (*defaultarm));

xdr_union() translates between a discriminated C union and its corresponding external representation. It first translates the discriminant of the union located at dscmp. This discriminant is always an enum_t. Next the union located at unp is translated. The parameter choices is a pointer to an array of xdr_discrim structures. Each structure contains an ordered pair of [value, proc]. If the union's discriminant is equal to the associated value, then the proc is called to translate the union. The end of the xdr_discrim structure array is denoted by a routine of value NULL. If the discriminant is not found in the choices array, then the defaultarm procedure is called (if it is not NULL). It returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_vector(XDR *xdrs, char *arrp, const uint_t size, const uint_t elsize, const xdrproc_t elproc);

xdr_vector() translates between fixed-length arrays and their corresponding external representations. The parameter arrp is the address of the pointer to the array, while size is the element count of the array. The parameter elsize is the sizeof each of the array's elements, and elproc is an XDR routine that translates between the array elements' C form and their external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_wrapstring(XDR *xdrs, char **sp);

A routine that calls xdr_string(xdrs, sp, maxuint); where maxuint is the maximum value of an unsigned integer.

Many routines, such as xdr_array(), xdr_pointer(), and xdr_vector() take a function pointer of type xdrproc_t(), which takes two arguments. xdr_string(), one of the most frequently used routines, requires three arguments, while xdr_wrapstring() only requires two. For these routines, xdr_wrapstring() is desirable. This routine returns TRUE if it succeeds, FALSE otherwise.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also malloc(3C), rpc(3NSL), xdr_admin(3NSL), xdr_create(3NSL), xdr_simple(3NSL), attributes(5)

Name xdr_create, xdr_destroy, xdrmem_create, xdrrec_create, xdrstdio_create – library routines for external data representation stream creation

Synopsis #include <rpc/xdr.h>

```
void xdr_destroy(XDR *xdrs);

void xdrmem_create(XDR *xdrs, const caddr_t addr, const uint_t size,
    const enum xdr_op op);

void xdrrec_create(XDR *xdrs, const uint_t sendsz, const uint_t recvsz,
    const caddr_t handle, const int (*readit)const void *read_handle,
    char *buf, const int len, const int (*writeit)
    const void *write_handle, const char *buf, const int len);

void xdrstdio_create(XDR *xdrs, FILE *
    file, const enum xdr_op op);
```

Description The XDR library routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data.

These routines deal with the creation of XDR streams, which must be created before any data can be translated into XDR format.

Routines See rpc(3NSL) for the definition of the XDR CLIENT and SVCXPRT data structures. Any buffers passed to the XDR routines must be properly aligned. Use malloc(3C) to allocate these buffers or be sure that the buffer address is divisible evenly by four.

xdr_destroy()	A macro that invokes the destroy routine associated with the XDR stream, *xdrs*. Private data structures associated with the stream are freed. Using *xdrs* after xdr_destroy() is invoked is undefined.
xdrmem_create()	This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to or read from a chunk of memory at location *addr* whose length is no less than *size* bytes long. The *op* determines the direction of the XDR stream. The value of *op* can be either XDR_ENCODE, XDR_DECODE, or XDR_FREE.
xdrrec_create()	This routine initializes the read-oriented XDR stream object pointed to by *xdrs*. The stream's data is written to a buffer of size *sendsz*. A value of 0 indicates the system should use a suitable default. The stream's data is read from a buffer of size *recvsz*. It too can be set to a suitable default by passing a 0 value. When a stream's output buffer is full, *writeit* is called. Similarly, when a stream's input buffer is empty, xdrrec_create() calls *readit*. The behavior of these two routines is similar to the system calls read() and write(), except that an appropriate handle, *read_handle* or *write_handle*, is passed to the

former routines as the first parameter instead of a file descriptor. See read(2) and write(2), respectively. The XDR stream's *op* field must be set by the caller.

This XDR stream implements an intermediate record stream. Therefore, additional bytes in the stream are provided for record boundary information.

xdrstdio_create()   This routine initializes the XDR stream object pointed to by *xdrs*. The XDR stream data is written to or read from the standard I/O stream file. The parameter *op* determines the direction of the XDR stream. The value of *op* can be either XDR_ENCODE, XDR_DECODE, or XDR_FREE.

The destroy routine associated with XDR streams calls fflush() on the *file* stream, but never fclose(). See fclose(3C).

A failure of any of these functions can be detected by first initializing the *x_ops* field in the XDR structure (*xdrs*-> *x_ops*) to NULL before calling the xdr*_create() function. If the *x_ops* field is still NULL, after the return from the xdr*_create() function, the call has failed. If the *x_ops* field contains some other value, assume that the call has succeeded.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also read(2), write(2), fclose(3C), malloc(3C), rpc(3NSL), xdr_admin(3NSL), xdr_complex(3NSL), xdr_simple(3NSL), attributes(5)

Name xdr_simple, xdr_bool, xdr_char, xdr_double, xdr_enum, xdr_float, xdr_free, xdr_hyper,
xdr_int, xdr_long, xdr_longlong_t, xdr_quadruple, xdr_short, xdr_u_char, xdr_u_hyper,
xdr_u_int, xdr_u_long, xdr_u_longlong_t, xdr_u_short, xdr_void – library routines for
external data representation

Synopsis #include<rpc/xdr.h>

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

bool_t xdr_char(XDR **xdrs*, char *cp*);

bool_t xdr_double(XDR **xdrs*, double *dp*);

bool_t xdr_enum(XDR **xdrs*, enum_t *ep*);

bool_t xdr_float(XDR **xdrs*, float *fp*);

void xdr_free(xdrproc_t *proc*, char *objp*);

bool_t xdr_hyper(XDR **xdrs*, longlong_t *llp*);

bool_t xdr_int(XDR **xdrs*, int *ip*);

bool_t xdr_long(XDR **xdrs*, longt *lp*);

bool_t xdr_longlong_t(XDR **xdrs*, longlong_t *llp*);

bool_t xdr_quadruple(XDR **xdrs*, long double *pq*);

bool_t xdr_short(XDR **xdrs*, short *sp*);

bool_t xdr_u_char(XDR **xdrs*, unsigned char *ucp*);

bool_t xdr_u_hyper(XDR **xdrs*, u_longlong_t *ullp*);

bool_t xdr_u_int(XDR **xdrs*, unsigned *up*);

bool_t xdr_u_long(XDR **xdrs*, unsigned long *ulp*);

bool_t xdr_u_longlong_t(XDR **xdrs*, u_longlong_t *ullp*);

bool_t xdr_u_short(XDR *xdrs*, unsigned short *usp*);

bool_t xdr_void(void)

Description The XDR library routines allow C programmers to describe simple data structures in a
machine-independent fashion. Protocols such as remote procedure calls (RPC) use these
routines to describe the format of the data.

These routines require the creation of XDR streams (see xdr_create(3NSL)).

Routines See rpc(3NSL) for the definition of the XDR data structure. Note that any buffers passed to the
XDR routines must be properly aligned. It is suggested that malloc(3C) be used to allocate
these buffers or that the programmer insure that the buffer address is divisible evenly by four.

xdr_bool()   xdr_bool() translates between booleans (C integers) and their
                      external representations. When encoding data, this filter produces
                      values of either 1 or 0. This routine returns TRUE if it succeeds,
                      FALSE otherwise.

xdr_char()   xdr_char() translates between C characters and their external
                      representations. This routine returns TRUE if it succeeds, FALSE
                      otherwise. Note: encoded characters are not packed, and occupy 4
                      bytes each. For arrays of characters, it is worthwhile to consider
                      xdr_bytes(), xdr_opaque(), or xdr_string() (see
                      xdr_complex(3NSL)).

xdr_double()   xdr_double() translates between C double precision numbers and
                      their external representations. This routine returns TRUE if it
                      succeeds, FALSE otherwise.

xdr_enum()   xdr_enum() translates between C enums (actually integers) and their
                      external representations. This routine returns TRUE if it succeeds,
                      FALSE otherwise.

xdr_float()   xdr_float() translates between C floats and their external
                      representations. This routine returns TRUE if it succeeds, FALSE
                      otherwise.

xdr_free()   Generic freeing routine. The first argument is the XDR routine for
                      the object being freed. The second argument is a pointer to the
                      object itself. Note: the pointer passed to this routine is not freed, but
                      what it points to is freed (recursively, depending on the XDR
                      routine).

xdr_hyper()   xdr_hyper() translates between ANSI C long long integers and
                      their external representations. This routine returns TRUE if it
                      succeeds, FALSE otherwise.

xdr_int()   xdr_int() translates between C integers and their external
                      representations. This routine returns TRUE if it succeeds, FALSE
                      otherwise.

xdr_long()   xdr_long() translates between C long integers and their external
                      representations. This routine returns TRUE if it succeeds, FALSE
                      otherwise.

                      In a 64-bit environment, this routine returns an error if the value of
                      lp is outside the range [INT32_MIN, INT32_MAX]. The xdr_int()
                      routine is recommended in place of this routine.

xdr_longlong_t()	xdr_longlong_t() translates between ANSI C long long integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise. This routine is identical to xdr_hyper().
xdr_quadruple()	xdr_quadruple() translates between IEEE quadruple precision floating point numbers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.
xdr_short()	xdr_short() translates between C short integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.
xdr_u_char()	xdr_u_char() translates between unsigned C characters and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.
xdr_u_hyper()	xdr_u_hyper() translates between unsigned ANSI C long long integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.
xdr_u_int()	A filter primitive that translates between a C unsigned integer and its external representation. This routine returns TRUE if it succeeds, FALSE otherwise.
xdr_u_long()	xdr_u_long() translates between C unsigned long integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise. In a 64-bit environment, this routine returns an error if the value of *ulp* is outside the range [0, UINT32_MAX]. The xdr_u_int() routine is recommended in place of this routine.
xdr_u_longlong_t()	xdr_u_longlong_t() translates between unsigned ANSI C long long integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise. This routine is identical to xdr_u_hyper().
xdr_u_short()	xdr_u_short() translates between C unsigned short integers and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise.
xdr_void()	This routine always returns TRUE. It may be passed to RPC routines that require a function parameter, where nothing is to be done.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also   malloc(3C), rpc(3NSL), xdr_admin(3NSL), xdr_complex(3NSL), xdr_create(3NSL),
attributes(5)

Name ypclnt, yp_get_default_domain, yp_bind, yp_unbind, yp_match, yp_first, yp_next, yp_all, yp_order, yp_master, yperr_string, ypprot_err – NIS Version 2 client interface

Synopsis cc [-flag...] *file*... -lnsl [*library*...]
```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>

int yp_bind(char *indomain);

void yp_unbind (char *indomain);

int yp_get_default_domain(char **outdomain);

int yp_match(char *indomain, char *inmap, char *inkey, int inkeylen,
     char *char **outval, int *outvallen);

int yp_first(char *indomain, char *inmap, char **outkey, int *outkeylen,
     char **outval, int *outvallen);

int yp_next(char *indomain, char *inmap, char *inkey, int *inkeylen,
     char **outkey, int *outkeylen, char **outval,
     int *outvallen);

int yp_all(char *indomain, char *inmap, struct ypall_callback *incallback);

int yp_order(char *indomain, char *inmap, unsigned long *outorder);

int yp_master(char *indomain, char *inmap, char **outname);

char *yperr_string(int incode);

int ypprot_err(char *domain);
```

Description This package of functions provides an interface to NIS, Network Information Service Version 2, formerly referred to as YP. In this version of SunOS, NIS version 2 is supported only for compatibility with previous versions. The current SunOS supports the client interface to NIS version 2. This client interface will be served by an existing ypserv process running on another machine on the network. For commands used to access NIS from a client machine, see ypbind(1M), ypwhich(1), ypmatch(1), and ypcat(1). The package can be loaded from the standard library, /usr/lib/libnsl.so.1.

All input parameter names begin with *in*. Output parameters begin with *out*. Output parameters of type char ** should be addresses of uninitialized character pointers. Memory is allocated by the NIS client package using malloc(3C) and can be freed by the user code if it has no continuing need for it. For each *outkey* and *outval*, two extra bytes of memory are allocated at the end that contain NEWLINE and null, respectively, but these two bytes are not reflected in *outkeylen* or *outvallen*. The *indomain* and *inmap* strings must be non-null and null-terminated. String parameters that are accompanied by a count parameter may not be null, but they may point to null strings, with the count parameter indicating this. Counted strings need not be null-terminated.

All functions in this package of type *int* return 0 if they succeed. Otherwise, they return a failure code (YPERR_*xxxx*). Failure codes are described in the ERRORS section.

Routines	yp_bind()	To use the NIS name services, the client process must be "bound" to an NIS server that serves the appropriate domain using yp_bind(). Binding need not be done explicitly by user code. Binding is done automatically whenever an NIS lookup function is called. The yp_bind() function can be called directly for processes that make use of a backup strategy , for example, a local file in cases when NIS services are not available. A process should call yp_unbind() when it is finished using NIS in order to free up resources.

yp_unbind()

Each binding allocates or uses up one client process socket descriptor. Each bound domain costs one socket descriptor. However, multiple requests to the same domain use that same descriptor. The yp_unbind() function is available at the client interface for processes that explicitly manage their socket descriptors while accessing multiple domains. The call to yp_unbind() makes the domain *unbound*, and frees all per-process and per-node resources used to bind it.

If an RPC failure results upon use of a binding, that domain will be unbound automatically. At that point, the ypclnt() layer will retry a few more times or until the operation succeeds, provided that rpcbind(1M) and ypbind(1M) are running, and either:

- The client process cannot bind a server for the proper domain; or
- RPC requests to the server fail.

Under the following circumstances, the ypclnt layer will return control to the user code, with either an error or success code and the results:

- If an error is not RPC-related.
- If rpcbind is not running.
- If ypbind is not running.
- If a bound ypserv process returns any answer (success or failure).

yp_get_default_domain()

NIS lookup calls require a map name and a domain name, at minimum. The client process should know the name of the map of interest. Client processes fetch the node's default domain by calling yp_get_default_domain() and use the returned *outdomain* as the *indomain* parameter to successive NIS name service calls. The domain returned is the same as

that returned using the SI_SRPC_DOMAIN command to the sysinfo(2) system call. The value returned in *outdomain* should not be freed.

yp_match()
: The yp_match() function returns the value associated with a passed key. This key must be exact because no pattern matching is available. yp_match() requires a full YP map name, such as hosts.byname, instead of the nickname hosts.

yp_first()
: The yp_first() function returns the first key-value pair from the named map in the named domain.

yp_next()
: The yp_next() function returns the next key-value pair in a named map. The *inkey* parameter must be the *outkey* returned from an initial call to yp_first() (to get the second key-value pair) or the one returned from the *n*th call to yp_next() (to get the *n*th + second key-value pair). Similarly, the *inkeylen* parameter must be the *outkeylen* returned from the earlier yp_first() or yp_next() call.

The concept of first and next is particular to the structure of the NIS map being processed. Retrieval order is not related to either the lexical order within any original (non-NIS name service) data base, or to any obvious numerical sorting order on the keys, values, or key-value pairs. The only ordering guarantee is that if the yp_first() function is called on a particular map, and then the yp_next() function is repeatedly called on the same map at the same server until the call fails with a reason of YPERR_NOMORE, every entry in the data base is seen exactly once. Further, if the same sequence of operations is performed on the same map at the same server, the entries are seen in the same order.

Under conditions of heavy server load or server failure, the domain can become unbound, then bound once again (perhaps to a different server) while a client is running. This binding can cause a break in one of the enumeration rules. Specific entries may be seen twice by the client, or not at all. This approach protects the client from error messages that would otherwise be returned in the midst of the enumeration. For a better solution to enumerating all entries in a map, see yp_all().

yp_all()
: The yp_all() function provides a way to transfer an entire map from server to client in a single request using TCP (rather than UDP as with other functions in this package).

The entire transaction takes place as a single RPC request and response. The yp_all() function can be used just like any other NIS name service procedure to identify the map in the normal manner and to supply the name of a function that will be called to process each key-value pair within the map. The call to yp_all() returns only when the transaction is completed (successfully or unsuccessfully), or the foreach() function decides that it does not want to see any more key-value pairs.

The third parameter to yp_all() is:

```
struct ypall_callback *incallback {
    int (*foreach)( );
    char *data;
};
```

The function foreach() is called:

```
foreach(int instatus, char *inkey,
int inkeylen, char *inval,
int invallen, char *indata);
```

The *instatus* parameter holds one of the return status values defined in <rpcsvc/yp_prot.h>, either YP_TRUE or an error code. See ypprot_err(), for a function that converts an NIS name service protocol error code to a ypclnt layer error code.

The key and value parameters are somewhat different than defined in the synopsis section above. First, the memory pointed to by the *inkey* and *inval* parameters is private to the yp_all() function, and is overwritten with the arrival of each new key-value pair. The foreach() function must do something useful with the contents of that memory, but it does not own the memory itself. Key and value objects presented to the foreach() function look exactly as they do in the server's map. If they were not NEWLINE-terminated or null-terminated in the map, they would not be here either.

The *indata* parameter is the contents of the *incallback->data* element passed to yp_all(). The data element of the callback structure can be used to share state information between the foreach() function and the mainline code. Its use is optional, and no part of the NIS client package inspects its contents; cast it to something useful, or ignore it. The foreach() function is Boolean. It should return 0 to indicate

that it wants to be called again for further received key-value pairs, or non-zero to stop the flow of key-value pairs. If foreach() returns a non-zero value, it is not called again. The functional value of yp_all() is then 0.

yp_order()	The yp_order() function returns the order number for a map.
yp_master()	The yp_master() function returns the machine name of the master NIS server for a map.
yperr_string()	The yperr_string() function returns a pointer to an error message string that is null-terminated but contains no period or NEWLINE.
ypprot_err()	The ypprot_err() function takes an NIS name service protocol error code as input, and returns a ypclnt() layer error code, which can be used as an input to yperr_string().

Return Values All integer functions return 0 if the requested operation is successful, or one of the following errors if the operation fails:

YPERR_ACCESS	Access violation.
YPERR_BADARGS	The arguments to the function are bad.
YPERR_BADDB	The YP database is bad.
YPERR_BUSY	The database is busy.
YPERR_DOMAIN	Cannot bind to server on this domain.
YPERR_KEY	No such key in map.
YPERR_MAP	No such map in server's domain.
YPERR_NODOM	Local domain name not set.
YPERR_NOMORE	No more records in map database.
YPERR_PMAP	Cannot communicate with rpcbind.
YPERR_RESRC	Resource allocation failure.
YPERR_RPC	RPC failure; domain has been unbound.
YPERR_YPBIND	Cannot communicate with ypbind.
YPERR_YPERR	Internal YP server or client error.
YPERR_YPSERV	Cannot communicate with ypserv.
YPERR_VERS	YP version mismatch.

Files /usr/lib/libnsl.so.1

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also ypcat(1), ypmatch(1), ypwhich(1), rpcbind(1M), ypbind(1M), ypserv(1M), sysinfo(2), malloc(3C), ypfiles(4), attributes(5)

Name   yp_update – change NIS information

Synopsis   #include <rpcsvc/ypclnt.h>

int yp_update(char *_domain_, char *_map_, unsigned _ypop_, char *_key_,
    char *int _keylen_, char *_data_, int _datalen_);

Description   yp_update() is used to make changes to the NIS database. The syntax is the same as that of
yp_match() except for the extra parameter _ypop_ which may take on one of four values. If it is
POP_CHANGE then the data associated with the key will be changed to the new value. If the key is
not found in the database, then yp_update() will return YPERR_KEY. If _ypop_ has the value
YPOP_INSERT then the key-value pair will be inserted into the database. The error YPERR_KEY is
returned if the key already exists in the database. To store an item into the database without
concern for whether it exists already or not, pass _ypop_ as YPOP_STORE and no error will be
returned if the key already or does not exist. To delete an entry, the value of _ypop_ should be
YPOP_DELETE.

This routine depends upon secure RPC, and will not work unless the network is running
secure RPC.

Return Values   If the value of _ypop_ is POP_CHANGE, yp_update() returns the error YPERR_KEY if the key is not
found in the database.

If the value of _ypop_ is POP_INSERT, yp_update() returns the error YPERR_KEY if the key
already exists in the database.

Attributes   See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also   secure_rpc(3NSL), ypclnt(3NSL), attributes(5)

Notes   This interface is unsafe in multithreaded applications. Unsafe interfaces should be called only
from the main thread.