# man pages section 3: Networking Library Functions

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 806-0628-10
February 2000

# 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 6 contains available games and demos.
- 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 9 provides reference information needed to write device drivers in the kernel environment. It describes two device driver interface specifications: the Device Driver Interface (DDI) and the Driver/Kernel Interface (DKI).
- Section 9E describes the DDI/DKI, 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 secton 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:<br>Commands<br>Modifiers<br>Variables<br>Expressions<br>Input Grammar |
| EXAMPLES | This section provides examples of usage or of how to use a command or function. Wherever possible a complete 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. |

CHAPTER

# Introduction to Library Functions

**NAME**

accept – accept a connection on a socket

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lsocket –lnsl` [ *library* ... ]
#include <sys / types.h>
#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. |
| `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`(3SOCKET), `netconfig`(4), `attributes`(5), `socket`(3HEAD)

## 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 **address*, socklen_t **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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`bind`(3XNET), `connect`(3XNET), `listen`(3XNET), `socket`(3XNET), `attributes`(5)

**NAME**

ber_decode, ber_alloc_t, ber_free, ber_bvdup, ber_init, ber_flatten, ber_get_next, ber_skiptag, 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**(struct berval **bv*);

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

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

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

**ber_get_next**(Sockbuf **sb*, unsigned long **len*, char **bv_val*);

**ber_skip_tag**(BerElement ***ber*, unsigned long ***len*);

**ber_peek_tag**(BerElement ***ber*, unsigned long ***len*);

**ber_get_int**(BerElement ***ber*, long ***num*);

**ber_get_stringb**(BerElement ***ber*, char ***buf*, unsigned long ***len*);

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

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

**ber_get_null**(BerElement ***ber*);

**ber_get_boolean**(BerElement ***ber*, int ***bool*);

**ber_get_bitstringa**(BerElement ***ber*, char ****buf*, unsigned long ***blen*);

**ber_first_element**(BerElement ***ber*, unsigned long ***len*, char ****cookie*);

**ber_next_element**(BerElement ***ber*, unsigned long ***len*, char ***cookie*);

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

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

**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_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 preallocated 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 *cookie* a magic cookie parameter that should be passed to subsequent calls to `ber_next_element( )` , which returns similar information.

`ber_alloc_t()` 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. `ber_init` returns the null pointer on error.

`ber_free()` 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.

`ber_bvdup()` 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` routine 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 routine 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),
         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 file. 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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

## SEE ALSO

`ber_encode`(3LDAP)

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. |
| `–i` | Integer. An integer parameter should be supplied. An integer 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. |
| `–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. |
| `–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 *'s 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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`attributes`(5) , `ber_decode`(3LDAP)

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 the `<lber.h>` header file.

## NAME

bind – bind a name to a socket

## SYNOPSIS

```
cc [ flag ... ] file ... –lsocket –lnsl [ library ... ]
#include <sys/types.h>
#include <sys/socket.h>

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

## DESCRIPTION

`bind( )` 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. `bind( )` requests that the name pointed to by *name* be assigned to the socket.

## RETURN VALUES

If the bind is successful, `0` is returned. A return value of `–1` indicates an error, which is further specified in the global `errno`.

## ERRORS

The `bind( )` call will fail if:

| | |
|---|---|
| `EACCES` | The requested address is protected and the current user has inadequate permission to access it. |
| `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. |
| `EINVAL` | The socket is already bound to an address. |
| `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), `socket`(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 (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 the current user does not have permission to bind to it. |
| `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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`connect`(3XNET), `getsockname`(3XNET), `listen`(3XNET), `socket`(3XNET), `attributes`(5)

**NAME**

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

**SYNOPSIS**

```
#include <sys/types.h>
#include <netinet/in.h>
#include <inttypes.h>
uint32_t htonl(unint32_t hostlong);

uint16_t htons(uint16_t hostshort);

uint32_t ntohl(uint32_t netlong);

uint16_t ntohs(uint16_t netshort);
```

**DESCRIPTION**

These routines convert 16 and 32 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, if their host byte order is different from network byte order, these routines are functional.

These routines 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) , `attributes`(5) , `inet`(3HEAD)

**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);
```

**PARAMETERS**

| | |
|---|---|
| `ld` | The LDAP pointer returned by a previous call to `cldap_open`(3LDAP). |

**DESCRIPTION**

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

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Availability | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`ldap`(3LDAP) `cldap_search_s`(3LDAP), `cldap_setretryinfo`(3LDAP), `cldap_close`(3LDAP), `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 5: | Repeat steps 2 and 3 for each LDAP server address. |
| Step 6: | Set the current timeout to twice its previous value and repeat Steps 2 through 6 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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`ldap`(3LDAP), `ldap_error`(3LDAP), `ldap_search_s`(3LDAP), `cldap_open`(3LDAP), `cldap_setretryinfo`(3LDAP), `cldap_close`(3LDAP), `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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`ldap`(3LDAP), `cldap_open`(3LDAP), `cldap_search_s`(3LDAP), `cldap_close`(3LDAP)

## NAME

connect – initiate a connection on a socket

## SYNOPSIS

```
cc [ flag ... ] file ... –lsocket –lnsl [ library ... ]
#include <sys/types.h>
#include <sys/socket.h>

int connect(int s, const struct sockaddr *name, int 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; it 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. The other socket is specified by *name*. *name* 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 it will be bound to an address selected by the underlying transport provider. Generally, stream sockets may successfully `connect( )` only once; datagram sockets may use `connect( )` multiple times to change their association. Datagram sockets may dissolve the association by connecting to a null address.

## RETURN VALUES

If the connection or binding succeeds, `0` is returned. Otherwise, `–1` is returned and sets `errno` to indicate the error.

## 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 on the remote machine. |
| `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. The calling program should `close`(2) the socket descriptor, and issue another `socket`(3SOCKET) call to obtain a new descriptor before attempting another `connect( )` call. |
| `EINPROGRESS` | The socket is non-blocking and the connection cannot be completed immediately. It is possible |

| | |
|---|---|
| | to `select`(3C) for completion by selecting the socket for writing. However, this is only possible if the socket STREAMS module is the topmost module on the protocol stack with a write service procedure. This will be the normal case. |
| `EINTR` | The connection attempt was interrupted before any data arrived by the delivery of a signal. |
| `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. |
| `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 may 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 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), `socket`(3SOCKET), `attributes`(5), `socket`(3HEAD)

## 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 |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

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

**NAME**

dial – establish an outgoing terminal line connection

**SYNOPSIS**

cc [ *flag* ... ] *file* ... –lnsl [ *library* ... ]
#include <dial.h>

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

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

**DESCRIPTION**

dial( ) 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**

doconfig – execute a configuration script

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lnsl` [ *library* ... ]
# include <sac.h>

int **doconfig**(int *fildes*, char **script*, long *rflag*);

**DESCRIPTION**

`doconfig( )` is a Service Access Facility library function that interprets the configuration scripts contained in the files `</etc/saf/`*pmtag*`/_config>`, `</etc/saf/_sysconfig>`, and `</etc/saf/`*pmtag*`/`*svctag*`>`, where *pmtag* specifies the tag associated with the port monitor, and *svctag* specifies the service tag associated with a given service. See `pmadm`(1M) and `sacadm`(1M).

`script` is the name of the configuration script; *fildes* is a file descriptor that designates the stream to which stream manipulation operations are to be applied; *rflag* is a bitmask that indicates the mode in which `script` is to be interpreted. If *rflag* is zero, all commands in the configuration script are eligible to be interpreted. If *rflag* has the `NOASSIGN` bit set, the `assign` command is considered illegal and will generate an error return. If *rflag* has the `NORUN` bit set, the `run` and `runwait` commands are considered illegal and will generate error returns.

The configuration language in which `script` is written consists of a sequence of commands, each of which is interpreted separately. The following reserved keywords are defined: `assign`, `push`, `pop`, `runwait`, and `run`. The comment character is #; when a # occurs on a line, everything from that point to the end of the line is ignored. Blank lines are not significant. No line in a command script may exceed 1024 characters.

`assign` *variable*=*value*
: Used to define environment variables. *variable* is the name of the environment variable and *value* is the value to be assigned to it. The value assigned must be a string constant; no form of parameter substitution is available. *value* may be quoted. The quoting rules are those used by the shell for defining environment variables. `assign` will fail if space cannot be allocated for the new variable or if any part of the specification is invalid.

`push` *module1*[, *module2*, *module3*, . . .]
: Used to push STREAMS modules onto the stream designated by *fildes*. *module1* is the name of the first module to be pushed, *module2* is the name of the second module to be pushed, etc. The command will fail if any of the named modules cannot be pushed. If a module cannot be pushed, the subsequent modules on the same command line will be ignored and modules that have already been pushed will be popped.

`pop` [*module*]

Used to pop STREAMS modules off the designated stream. If `pop` is invoked with no arguments, the top module on the stream is popped. If an argument is given, modules will be popped one at a time until the named module is at the top of the stream. If the named module is not on the designated stream, the stream is left as it was and the command fails. If *module* is the special keyword ALL, then all modules on the stream will be popped. Note that only modules above the topmost driver are affected.

`runwait` `command`
The `runwait` command runs a command and waits for it to complete. `command` is the pathname of the command to be run. The command is run with `/usr/bin/sh –c` prepended to it; shell scripts may thus be executed from configuration scripts. The `runwait` command will fail if `command` cannot be found or cannot be executed, or if `command` exits with a non-zero status.

`run` `command`
The `run` command is identical to `runwait` except that it does not wait for `command` to complete. `command` is the pathname of the command to be run. `run` will not fail unless it is unable to create a child process to execute the command.

Although they are syntactically indistinguishable, some of the commands available to `run` and `runwait` are interpreter built-in commands. Interpreter built-ins are used when it is necessary to alter the state of a process within the context of that process. The `doconfig()` interpreter built-in commands are similar to the shell special commands and, like these, they do not spawn another process for execution. See `sh`(1). The built-in commands are:

```
cd
ulimit
umask
```

**RETURN VALUES**

`doconfig()` returns `0` if the script was interpreted successfully. If a command in the script fails, the interpretation of the script ceases at that point and a positive number is returned; this number indicates which line in the script failed. If a system error occurs, a value of `–1` is returned. When a script fails, the process whose environment was being established should *not* be started.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Unsafe |

**SEE ALSO**

`sh`(1), `pmadm`(1M), `sacadm`(1M), `attributes`(5)

**NOTES**

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

## 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, size_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 |
|---|---|
| MT-Level | Unsafe |

**SEE ALSO**

`endservent`(3XNET), `htonl`(3XNET), `inet_addr`(3XNET), `attributes`(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 |
|---|---|
| MT-Level | Unsafe |

**SEE ALSO**

`attributes`(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 |
|---|---|
| MT-Level | Unsafe |

**SEE ALSO**

`attributes`(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 |
|---|---|
| MT-Level | Unsafe |

**SEE ALSO**

`endhostent`(3XNET), `endprotoent`(3XNET), `htonl`(3XNET), `inet_addr`(3XNET), `attributes`(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/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
char *ether_ntoa(struct ether_addr *e);

struct ether_addr *ether_aton(char *s);

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

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

int ether_line(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, the NIS maps “ethers.byname” and “ethers.byaddr” and the NIS+ table “ethers”. 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`

`/etc/nsswitch.conf`

**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)

**BUGS**

Programs that call `ether_hostton( )` or `ether_ntohost( )` routines cannot be linked statically since the implementation of these routines requires dynamic linker functionality to access shared objects at run time.

## NAME

fn_attr_bind – bind a reference to a name and associate attributes with named object

## SYNOPSIS

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

int fn_attr_bind(FN_ctx_t *ctx, const FN_composite_name_t *name, const FN_ref_t *ref,
const FN_attrset_t *attrs, unsigned int exclusive, FN_status_t *status);
```

## DESCRIPTION

This operation binds the supplied reference *ref* to the supplied composite name *name* relative to *ctx*, and associates the attributes specified in *attrs* with the named object. The binding is made in the target context, that is, that context named by all but the terminal atomic part of *name*. The operation binds the terminal atomic name to the supplied reference in the target context. The target context must already exist.

The value of *exclusive* determines what happens if the terminal atomic part of the name is already bound in the target context. If *exclusive* is nonzero and *name* is already bound, the operation fails. If *exclusive* is `0`, the new binding replaces any existing binding, and, if *attrs* is not `NULL`, *attrs* replaces any existing attributes associated with the named object. If *attrs* is `NULL` and *exclusive* is `0`, any existing attributes associated with the named object are left unchanged.

## RETURN VALUES

`fn_attr_bind( )` returns `1` upon success, `0` upon failure.

## ERRORS

`fn_attr_bind( )` sets *status* as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN). Of special relevance for this operation is the following status code:

| | |
|---|---|
| `FN_E_NAME_IN_USE` | The supplied name is already in use. |

## USAGE

The value of *ref* cannot be `NULL`. If the intent is to reserve a name using `fn_attr_bind( )`, a reference containing no address should be supplied. This reference may be name service-specific or it may be the conventional `NULL` reference.

If multiple sources are updating a reference or attributes associated with a named object, they must synchronize amongst each other when adding, modifying, or removing from the address list of a bound reference, or manipulating attributes associated with the named object.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`FN_composite_name_t`(3XFN), `FN_ctx_t`(3XFN), `FN_ref_t`(3XFN), `FN_status_t`(3XFN), `fn_ctx_bind`(3XFN), fn_ctx_lookup(3XFN), fn_ctx_unbind(3XFN), xfn_attributes(3XFN), xfn_status_codes(3XFN), attributes(5)

## NAME

fn_attr_create_subcontext – create a subcontext in a context and associate attributes with newly created context

## SYNOPSIS

#include <xfn/xfn.h>

FN_ref_t ***fn_attr_create_subcontext**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, const FN_attrset_t **attrs*, FN_status_t **status*);

## DESCRIPTION

This operation creates a new XFN context of the same type as the target context, that is, that context named by all but the terminal atomic component of *name*, and binds it to the supplied composite name. In addition, attributes given in *attrs* are associated with the newly created context.

The target context must already exist. The new context is created and bound in the target context using the terminal atomic name in *name*. The operation returns a reference to the newly created context.

## RETURN VALUES

`fn_attr_create_subcontext()` returns a reference to the newly created context; if the operation fails, it returns a `NULL` pointer.

## ERRORS

`fn_attr_create_subcontext()` sets *status* as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN). Of special relevance for this operation is the following status code:

| | |
|---|---|
| `FN_E_NAME_IN_USE` | The terminal atomic name already exists in the target context. |

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`FN_composite_name_t`(3XFN), `FN_ctx_t`(3XFN), `FN_ref_t`(3XFN), `FN_status_t`(3XFN), `fn_attr_bind`(3XFN), `fn_ctx_bind`(3XFN), `fn_ctx_create_subcontext`(3XFN), `fn_ctx_destroy_subcontext`(3XFN), `fn_ctx_lookup`(3XFN), `xfn_attributes`(3XFN), `xfn_status_codes`(3XFN), `attributes`(5)

## NAME

fn_attr_ext_search, FN_ext_searchlist_t, fn_ext_searchlist_next, fn_ext_searchlist_destroy – search for names in the specified context(s) whose attributes satisfy the filter

## SYNOPSIS

#include <xfn/xfn.h>
FN_ext_searchlist_t ***fn_attr_ext_search**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, const FN_search_control_t **control*, const FN_search_filter_t **filter*, FN_status_t **status*);

FN_composite_name_t ***fn_ext_searchlist_next**(FN_ext_searchlist_t **esl*, FN_ref_t ***returned_ref*, FN_attrset_t ***returned_attrs*, FN_status_t **status*);

void **fn_ext_searchlist_destroy**(FN_ext_searchlist_t **esl*);

## DESCRIPTION

This set of operations is used to list names of objects whose attributes satisfy the filter expression. The references to which these names are bound and specified attributes and their values may also be returned.

*control* encapsulates the option settings for the search. These options are:

- the scope of the search
- whether XFN links are followed
- a limit on the number of names returned
- whether references and specific attributes associated with the named objects that satisfy the filter are returned

The scope of the search is one of:

- the object named *name* relative to the context *ctx*
- the context named *name* relative to the context *ctx*
- the context named *name* relative to the context *ctx* ,
  and its subcontexts
  or
- the context named *name* relative to the context *ctx* , and a context implementation-defined set of subcontexts

If the value of *control* is `0` , default control option settings are used. The default settings are:

- scope is search named context
- links are not followed
- all names of objects that satisfy the filter are returned
- references and attributes are not returned

The `FN_search_control_t` type is described in `FN_search_control_t`(3XFN) .

The filter expression *filter* in `fn_attr_ext_search()` is evaluated against the attributes of the objects bound in the scope of the search. The filter evaluates to either `TRUE` or `FALSE` . The names and, optionally, the references and attributes of objects whose attributes satisfy the filter are enumerated. If the value of *filter* is `0` , all names within the search scope are enumerated. The `FN_search_filter_t` type is described in `FN_search_filter_t`(3XFN) .

The call to `fn_attr_ext_search()` initiates the search process. It returns a handle to an `FN_ext_searchlist_t` object that is used to enumerate the names of the objects that satisfy the filter.

The operation `fn_ext_searchlist_next()` returns the next name in the enumeration identified by *esl* ; it also updates *esl* to indicate the state of the enumeration. If the reference to which the name is bound was requested, it is returned in *returned_ref* . Requested attributes associated with the name are returned in *returned_attrs* ; each attribute consists of an attribute identifier, syntax, and value(s). Successive calls to `fn_ext_searchlist_next()` using *esl* return successive names and, optionally, their references and attributes, in the enumeration; these calls further update the state of the enumeration.

The names that are returned are composite names, to be resolved relative to the starting context for the search. This starting context is the context named *name* relative to *ctx* unless the scope of the search is only the named object. If the scope of the search is only the named object, the terminal atomic name in *name* is returned.

`fn_ext_searchlist_destroy()` releases resources used during the enumeration. This may be invoked at any time to terminate the enumeration.

**RETURN VALUES**

`fn_attr_ext_search()` returns a pointer to an `FN_ext_searchlist_t` object if the search is successfully initiated; it returns a `NULL` pointer if the search cannot be initiated or if no named object with attributes whose values satisfy the filter expression is found.

`fn_ext_searchlist_next()` returns a pointer to an `FN_composite_name_t` object (see `FN_composite_name_t`(3XFN) ) that is the next name in the enumeration; it returns a `NULL` pointer if no more names can be returned. If *returned_attrs* is a `NULL` pointer, no attributes are returned; otherwise, *returned_attrs* contains the attributes associated with the named object, as specified in the control parameter to `fn_attr_ext_search()` . If *returned_ref* is a `NULL` pointer, no reference is returned; otherwise, if *control* specified the return of the reference of the named object, that reference is returned in *returned_ref* .

In the case of a failure, these operations return in the *status* argument a code indicating the nature of the failure.

## ERRORS

If successful, `fn_attr_ext_search()` returns a pointer to an `FN_ext_searchlist_t` object and sets *status* to `FN_SUCCESS.`

`fn_attr_ext_search()` returns a `NULL` pointer when no more names can be returned. *status* is set in the following way:

| | |
|---|---|
| `FN_SUCCESS` | A named object could not be found whose attributes satisfied the filter expression. |
| `FN_E_NOT_A_CONTEXT` | The object named for the start of the search was not a context and the search scope was the given context or the given context and its subcontexts. |
| `FN_E_SEARCH_INVALID_FILTER` | The filter could not be evaluated `TRUE` or `FALSE,` or there was some other problem with the filter. |
| `FN_E_SEARCH_INVALID_OPTION` | A supplied search control option could not be supported. |
| `FN_E_SEARCH_INVALID_OP` | An operator in the filter expression is not supported or, if the operator is an extended operator, the number of types of arguments supplied does not match the signature of the operation. |
| `FN_E_ATTR_NO_PERMISSION` | The caller did not have permission to read one or more of the attributes specified in the filter. |
| `FN_E_INVALID_ATTR_VALUE` | A value type in the filter did not match the syntax of the attribute against which it was being evaluated. |

Other status codes are possible as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN) .

Each successful call to `fn_ext_searchlist_next()` returns a name and, optionally, its reference in *returned_ref* and requested attributes in *returned_attrs* . *status* is set in the following way:

| | |
|---|---|
| `FN_SUCCESS` | All requested attributes were returned successfully with the name. |
| `FN_E_ATTR_NO_PERMISSION` | The caller did not have permission to read one or more of the requested attributes. |

| | |
|---|---|
| `FN_E_INVALID_ATTR_IDENTIFIER` | A requested attribute identifier was not in a format acceptable to the naming system, or its contents were not valid for the format specified. |
| `FN_E_NO_SUCH_ATTRIBUTE` | The named object did not have one of the requested attributes. |
| `FN_E_INSUFFICIENT_RESOURCES` | Insufficient resources are available to return all the requested attributes and their values. |
| `FN_E_ATTR_NO_PERMISSION` | |
| `FN_E_INVALID_ATTR_IDENTIFIER` | |
| `FN_E_NO_SUCH_ATTRIBUTE` | |
| `FN_E_INSUFFICIENT_RESOURCES` | These indicate that some of the requested attributes may have been returned in *returned_attrs* but one or more of them could not be returned. Use `fn_attr_get`(3XFN) or `fn_attr_multi_get`(3XFN) to discover why these attributes could not be returned. |

If `fn_ext_searchlist_next( )` returns a name, it can be called again to get the next name in the enumeration.

`fn_ext_searchlist_next( )` returns a `NULL` pointer if no more names can be returned. *status* is set in the following way:

| | |
|---|---|
| `FN_SUCCESS` | The search has completed successfully. |
| `FN_E_PARTIAL_RESULT` | The enumeration is not yet complete but cannot be continued. |
| `FN_E_ATTR_NO_PERMISSION` | The caller did not have permission to read one or more of the attributes specified in the filter. |
| `FN_E_INVALID_ENUM_HANDLE` | The supplied enumeration handle was not valid. Possible reasons could be that the handle was from another enumeration, or the context being enumerated no longer accepts the handle (due to such events as handle expiration or updates to the context). |

Other status codes are possible as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN) .

**USAGE**

The search performed by `fn_attr_ext_search( )` is not ordered in any way, including the traversal of subcontexts. The names enumerated using `fn_ext_searchlist_next( )` are not ordered in any way. Furthermore, there is no guarantee that any two series of enumerations with the same arguments to `fn_attr_ext_search( )` will return the names in the same order.

XFN links encountered during the resolution of *name* are followed, regardless of the follow links control setting, and the search starts at the final named object or context.

If *control* specifies that the search should follow links, XFN link names encountered during the search are followed and the terminal named object is searched. If the terminal named object is bound to a context and the scope of the search includes subcontexts, that context and its subcontexts are also searched. For example, if *aname* is bound to an XFN link, *lname* , in a context within the scope of the search, and *aname* is returned by `fn_ext_searchlist_next( )` , this means that the object identified by *lname* satisfied the filter expression. *aname* is returned instead of *lname* because *aname* can always be named relative to the starting context for the search.

If *control* specifies that the search should not follow links, the attributes associated with the names of XFN links are searched. For example, if *aname* is bound to an XFN link, *lname* , in a context within the scope of the search, and *aname* is returned by `fn_ext_searchlist_next( )` , this means that the object identified by *aname* satisfied the filter expression.

When following XFN links, `fn_attr_ext_search( )` may search contexts outside of *scope* . In addition, if the link name's terminal atomic name is bound in a context within *scope* , the operation may return the same object more than once.

XFN does not specify how *control* affects the following of native naming system links during the search.

**EXAMPLES**

**EXAMPLE 1** A sample program of displaying how the `fn_attr_ext_search( )` operation may be used.

The following code fragment illustrates how the `fn_attr_ext_search( )` operation may be used. The code consists of three parts: preparing the arguments for the search, performing the search, and cleaning up.

The first part involves getting the name of the context to start the search and constructing the search filter that named objects in the context must satisfy. This is done in the declarations part of the code and by the routine `get_search_query` . See `FN_search_filter_t`(3XFN) for the description of *sfilter* and the filter creation operation.

The next part involves doing the search and enumerating the results of the search. This is done by first getting a context handle to the Initial Context, and then passing that handle along with the name of the target context and search filter to `fn_attr_ext_search( )` . This particular call to `fn_attr_ext_search( )` uses the default search control options (by passing in `0` as the *control* argument). This means that the search will be performed in the context named by *target_name* and that no reference or attributes will be returned. In addition, any XFN links encountered will not be followed and all named objects that satisfy the search filter will be returned (that is, no limit). If successful, `fn_attr_ext_search( )` returns *esl* , a handle for enumerating the results of the search. The results of the search are enumerated using calls to `fn_ext_searchlist_next( )` , which returns the name of the object. (The arguments *returned_ref* and *returned_attrs* to `fn_ext_searchlist_next( )` are `0` because the default search control used i `fn_attr_ext_search( )` did not request them to be returned.)

The last part of the code involves cleaning up the resources used during the search and enumeration. The call to `fn_ext_searchlist_destroy( )` releases resources reserved for this enumeration. The other calls release the context handle, name, filter, and status objects created earlier.

```
/* Declarations */
FN_ctx_t *ctx;
FN_ext_searchlist_t *esl;
FN_composite_name_t *name;
FN_status_t *status = fn_status_create();
FN_composite_name_t *target_name = get_name_from_user_input();
FN_search_filter_t *sfilter = get_search_query();
/* Get context handle to Initial Context */
ctx = fn_ctx_handle_from_initial(status);
/* error checking on 'status' */
/* Initiate search */
if ((esl=fn_attr_ext_search(ctx, target_name,
 /* default controls */ 0, sfilter, status)) == 0) {
 /* report 'status', cleanup, and exit */
}
/* Enumerate names requested */
while (name=fn_ext_searchlist_next(esl, 0, 0, status)) {
 /* do something with 'name' */
 fn_composite_destroy(name);
}
/* check 'status' for reason for end of enumeration */
/* Clean up */
fn_ext_searchlist_destroy(esl);
fn_search_filter_destroy(sfilter);
fn_ctx_handle_destroy(ctx);
fn_composite_name_destroy(target_name);
fn_status_destroy(status);
/*
* Procedure for constructing the filter object for search:
*  "age" attribute is greater than or equal to 17 AND
*  less than or equal to 25
* AND the "student" attribute is present.
```

```
*/
FN_search_filter_t *
get_search_query()
{
 extern FN_attribute_t *attr_age;
 extern FN_attribute_t *attr_student;
 FN_search_filter_t *sfilter;
 unsigned int filter_status;
 sfilter = fn_search_filter_create(
  &filter_status,
  "(%a >= 17) and (%a <= 25) and %a",
  attr_age, attr_age, attr_student);
 /* error checking on ’filter_status’ */
 return (sfilter);
}
```

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_attrset_t`(3XFN) , `FN_composite_name_t`(3XFN) , `FN_ctx_t`(3XFN) , `FN_ref_t`(3XFN) , `FN_search_control_t`(3XFN) , `FN_search_filter_t`(3XFN) , `FN_status_t`(3XFN) , `fn_attr_get`(3XFN) , `fn_attr_multi_get`(3XFN) , `xfn_status_codes`(3XFN) , `attributes`(5)

## NAME

fn_attr_get – return specified attribute associated with name

## SYNOPSIS

`cc [ flag ... ] file ... -lxfn [ library ... ]`
`#include <xfn/xfn.h>`

FN_attribute_t ***fn_attr_get**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, const FN_identifier_t **attribute_id*, unsigned int *follow_link*, FN_status_t **status*);

## DESCRIPTION

This operation returns the identifier, syntax and values of a specified attribute for the object named *name* relative to *ctx*. If *name* is empty, the attribute associated with *ctx* is returned.

The value of *follow_link* determines what happens when the terminal atomic part of *name* is bound to an XFN link. If *follow_link* is non-zero, such a link is followed, and the values of the attribute associated with the final named object are returned; if *follow_link* is zero, such a link is not followed. Any XFN links encountered before the terminal atomic name are always followed.

## RETURN VALUES

`fn_attr_get` returns a pointer to an `FN_attribute_t` object if the operation succeeds; it returns a `NULL` pointer (`0`) if the operation fails.

## ERRORS

`fn_attr_get()` sets *status* as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN).

## USAGE

`fn_attr_get_values()` and its related operations are used for getting individual values of an attribute. They should be used if the combined size of all the values are expected to be too large to be returned in a single invocation of `fn_attr_get()`.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`FN_attribute_t`(3XFN), `FN_composite_name_t`(3XFN), `FN_ctx_t`(3XFN), `FN_identifier_t`(3XFN), `FN_status_t`(3XFN), `fn_attr_get_values`(3XFN), `xfn`(3XFN), `xfn_attributes`(3XFN), `xfn_status_codes`(3XFN), `attributes`(5)

## NOTES

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

## NAME

fn_attr_get_ids – get a list of the identifiers of all attributes associated with named object

## SYNOPSIS

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>

FN_attrset_t ***fn_attr_get_ids**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, unsigned int *follow_link*, FN_status_t **status*);

## DESCRIPTION

This operation returns a list of the attribute identifiers of all attributes associated with the object named by *name* relative to the context *ctx*. If *name* is empty, the attribute identifiers associated with *ctx* are returned.

The value of *follow_link* determines what happens when the terminal atomic part of *name* is bound to an XFN link. If *follow_link* is non-zero, such a link is followed, and the values of the attribute associated with the final named object are returned; if *follow_link* is zero, such a link is not followed. Any XFN links encountered before the terminal atomic name are always followed.

## RETURN VALUES

This operation returns a pointer to an object of type `FN_attrset_t`; if the operation fails, a `NULL` pointer (`0`) is returned.

## ERRORS

This operation sets *status* as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN).

## USAGE

The attributes in the returned set do not contain the syntax or values of the attributes, only their identifiers.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`FN_attribute_t`(3XFN), `FN_attrset_t`(3XFN), `FN_composite_name_t`(3XFN), `FN_ctx_t`(3XFN), `FN_status_t`(3XFN), `fn_attr_get`(3XFN), `fn_attr_multi_get`(3XFN) `xfn`(3XFN), `xfn_attributes`(3XFN), `xfn_status_codes`(3XFN), `attributes`(5)

## NOTES

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

fn_attr_get_values, FN_valuelist_t, fn_valuelist_next, fn_valuelist_destroy – return values of an attribute

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>
FN_valuelist_t ***fn_attr_get_values**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, const FN_identifier_t **attribute_id*, unsigned int *follow_link*, FN_status_t **status*);

FN_attrvalue_t ***fn_valuelist_next**(FN_valuelist_t **vl*, FN_identifier_t ***attr_syntax*, FN_status_t **status*);

void **fn_valuelist_destroy**(FN_valuelist_t **vl*, FN_status_t **status*);

**DESCRIPTION**

This set of operations is used to obtain the values of a single attribute, identified by *attribute_id* , associated with the object named *name* , resolved in the context *ctx* . If *name* is empty, the attribute values associated with *ctx* are obtained.

The value of *follow_link* determines what happens when the terminal atomic part of *name* is bound to an XFN link. If *follow_link* is non-zero, such a link is followed, and the values of the attribute associated with the final named object are returned; if *follow_link* is zero, such a link is not followed. Any XFN links encountered before the terminal atomic name are always followed.

The operation `fn_attr_get_values( )` initiates the enumeration process. It returns a handle to an `FN_valuelist_t` object that can be used to enumerate the values of the specified attribute.

The operation `fn_valuelist_next( )` returns a new `FN_attrvalue_t` object containing the next value in the attribute and may be called multiple times until all values are retrieved. The syntax of the attribute is returned in *attr_syntax* .

The operation `fn_valuelist_destroy( )` is used to release the resources used during the enumeration. This may be invoked before the enumeration has completed to terminate the enumeration.

These operations work in a fashion similar to the `fn_ctx_list_names( )` operations.

**RETURN VALUES**

`fn_attr_get_values( )` returns a pointer to an `FN_valuelist_t` object if the enumeration process is successfully initiated; it returns a `NULL` pointer if the process failed.

`fn_valuelist_next( )` returns a `NULL` pointer if no more attribute values can be returned.

In the case of a failure, these operations set *status* to indicate the nature of the failure.

**ERRORS**

Each successful call to `fn_valuelist_next( )` returns an attribute value. *status* is set to `FN_SUCCESS` .

When `fn_valuelist_next()` returns a `NULL` pointer, it indicates that no more values can be returned. *status* is set in the following way:

| | |
|---|---|
| `FN_SUCCESS` | The enumeration has completed successfully. |
| `FN_E_INVALID_ENUM_HANDLE` | The given enumeration handle is not valid. Possible reasons could be that the handle was from another enumeration, or the context being enumerated no longer accepts the handle (due to such events as handle expiration or updates to the context). |
| `FN_E_PARTIAL_RESULT` | The enumeration is not yet complete but cannot be continued. |

In addition to these status codes, other status codes are also possible in calls to these operations. In such cases, *status* is set as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN) .

**USAGE**

This interface should be used instead of `fn_attr_get()` if the combined size of all the values is expected to be too large to be returned by `fn_attr_get()` .

There may be a relationship between the *ctx* argument supplied to `fn_attr_get_values()` and the `FN_valuelist_t` object it returns. For example, some implementations may store the context handle *ctx* within the `FN_valuelist_t` object for subsequent `fn_valuelist_next()` calls. In general, an `fn_ctx_handle_destroy`(3XFN) should not be invoked on *ctx* until the enumeration has terminated.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_attribute_t`(3XFN) , `FN_attrvalue_t`(3XFN) , `FN_composite_name_t`(3XFN) , `FN_ctx_t`(3XFN) , `FN_identifier_t`(3XFN) , `FN_status_t`(3XFN) , `fn_attr_get`(3XFN) , `fn_ctx_handle_destroy`(3XFN) , `fn_ctx_list_names`(3XFN) , `xfn`(3XFN) , `xfn_attributes`(3XFN) , `xfn_status_codes`(3XFN) , `attributes`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

## NAME

FN_attribute_t, fn_attribute_create, fn_attribute_destroy, fn_attribute_copy, fn_attribute_assign, fn_attribute_identifier, fn_attribute_syntax, fn_attribute_valuecount, fn_attribute_first, fn_attribute_next, fn_attribute_add, fn_attribute_remove – an XFN attribute

## SYNOPSIS

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>
FN_attribute_t ***fn_attribute_create**(constFN_identifier_t **attribute_id*, const FN_identifier_t **attribute_syntax*);

void **fn_attribute_destroy**(FN_attribute_t **attr*);

FN_attribute_t ***fn_attribute_copy**(constFN_attribute_t **attr*);

FN_attribute_t ***fn_attribute_assign**(FN_attribute_t **dst*, const FN_attribute_t **src*);

const FN_identifier_t ***fn_attribute_identifier**(constFN_attribute_t **attr*);

const FN_identifier_t ***fn_attribute_syntax**(constFN_attribute_t **attr*);

unsigned int **fn_attribute_valuecount**(constFN_attribute_t **attr*);

const FN_attrvalue_t ***fn_attribute_first**(constFN_attribute_t **attr*, void ***iter_pos*);

const FN_attrvalue_t ***fn_attribute_next**(constFN_attribute_t **attr*, void ***iter_pos*);

int **fn_attribute_add**(FN_attribute_t **attr*, const FN_attrvalue_t **attribute_value*, unsigned int *exclusive*);

int **fn_attribute_remove**(FN_attribute_t **attr*, const FN_attrvalue_t **attribute_value*);

## DESCRIPTION

An attribute has an attribute identifier, a syntax, and a set of distinct values. Each value is a sequence of octets. The operations associated with objects of type `FN_attribute_t` allow the construction, destruction, and manipulation of an attribute and its value set.

The attribute identifier and its syntax are specified using an `FN_identifier_t` . `fn_attribute_create( )` creates a new attribute object with the given identifier and syntax, and an empty set of values. `fn_attribute_destroy( )` releases the storage associated with *attr* . `fn_attribute_copy( )` returns a copy of the object pointed to by *attr* . `fn_attribute_assign( )` makes a copy of the attribute object pointed to by *src* and assigns it to *dst* , releasing any old contents of *dst* . A pointer to the same object as *dst* is returned.

`fn_attribute_identifier( )` returns the attribute identifier of *attr* . `fn_attribute_syntax( )` returns the attribute syntax of *attr* . `fn_attribute_valuecount( )` returns the number of attribute values in *attr* .

`fn_attribute_first( )` and `fn_attribute_next( )` are used to enumerate the values of an attribute. Enumeration of the values of an attribute may return the values in any order. `fn_attribute_first( )` returns an attribute value from *attr* and sets the iteration marker *iter_pos* . Subsequent calls to `fn_attribute_next( )` returns the next attribute value identified by *iter_pos* and advances *iter_pos* . Adding or removing values from an attribute invalidates any iteration markers that the caller holds.

`fn_attribute_add( )` adds a new value *attribute_value* to *attr* . The operation succeeds (but no change is made) if *attribute_value* is already in *attr* and *exclusive* is `0` ; the operation fails if *attribute_value* is already in *attr* and *exclusive* is non-zero.

`fn_attribute_remove( )` removes *attribute_value* from *attr* . The operation succeeds even if *attribute_value* is not amongst *attr* 's values.

**RETURN VALUES**

`fn_attribute_first( )` returns `0` if the attribute contains no values. `fn_attribute_next( )` returns `0` if there are no more values to be returned in the attribute (as identified by the iteration marker) or if the iteration marker is invalid.

`fn_attribute_add( )` and `fn_attribute_remove( )` return `1` if the operation succeeds, `0` if it fails.

**USAGE**

Manipulation of attributes using the operations described in this manual page does not affect their representation in the underlying naming system. Changes to attributes in the underlying naming system can only be effected through the use of the interfaces described in `xfn_attributes`(3XFN) .

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_attrset_t`(3XFN) , `FN_attrvalue_t`(3XFN) , `FN_identifier_t`(3XFN) , `fn_attr_get`(3XFN) , `fn_attr_modify`(3XFN) , `xfn`(3XFN) , `xfn_attributes`(3XFN) , `attributes`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

fn_attr_modify – modify specified attribute associated with name

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>

int **`fn_attr_modify`**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, unsigned int *mod_op*, const FN_attribute_t **attr*, unsigned int *follow_link*, FN_status_t **status*);

**DESCRIPTION**

This operation modifies according to *mod_op* the attribute *attr* associated with the object named *name* relative to *ctx*. If *name* is empty, the attribute associated with *ctx* is modified.

The value of *follow_link* determines what happens when the terminal atomic part of *name* is bound to an XFN link. If *follow_link* is non-zero, such a link is followed, and the values of the attribute associated with the final named object are returned; if *follow_link* is zero, such a link is not followed. Any XFN links encountered before the terminal atomic name are always followed.

The modification is made on the attribute identified by the attribute identifier of *attr*. The syntax and values of *attr* are used according to the modification operation.

The modification operations are as follows:

| | |
|---|---|
| `FN_ATTR_OP_ADD` | Add an attribute with given attribute identifier and set of values. If an attribute with this identifier already exists, replace the set of values with those in the given set. The set of values may be empty if the target naming system permits. |
| `FN_ATTR_OP_ADD_EXCLUSIVE` | Add an attribute with the given attribute identifier and set of values. The operation fails if an attribute with this identifier already exists. The set of values may be empty if the target naming system permits. |
| `FN_ATTR_OP_REMOVE` | Remove the attribute with the given attribute identifier and all of its values. The operation succeeds even if the attribute does not exist. The values of the attribute supplied with this operation are ignored. |
| `FN_ATTR_OP_ADD_VALUES` | Add the given values to those of the given attribute (resulting in the |

| | |
|---|---|
| | attribute having the union of its prior value set with the set given). Create the attribute if it does not exist already. The set of values may be empty if the target naming system permits. |
| `FN_ATTR_OP_REMOVE_VALUES` | Remove the given values from those of the given attribute (resulting in the attribute having the set difference of its prior value set and the set given). This succeeds even if some of the given values are not in the set of values that the attribute has. In naming systems that require an attribute to have at least one value, removing the last value will remove the attribute as well. |

**RETURN VALUES**

| | |
|---|---|
| `1` | Successful operation. |
| `0` | Operation failed. |

**ERRORS**

`fn_attr_modify()` sets *status* as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN).

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_attribute_t`(3XFN), `FN_composite_name_t`(3XFN), `FN_ctx_t`(3XFN), `FN_status_t`(3XFN), `fn_attr_multi_modify`(3XFN), `xfn`(3XFN), `xfn_attributes`(3XFN), `xfn_status_codes`(3XFN), `attributes`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

## NAME

FN_attrmodlist_t, fn_attrmodlist_create, fn_attrmodlist_destroy, fn_attrmodlist_copy, fn_attrmodlist_assign, fn_attrmodlist_count, fn_attrmodlist_first, fn_attrmodlist_next, fn_attrmodlist_add – a list of attribute modifications

## SYNOPSIS

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>
FN_attrmodlist_t ***fn_attrmodlist_create**(void);

void **fn_attrmodlist_destroy**(FN_attrmodlist_t **modlist*);

FN_attrmodlist_t ***fn_attrmodlist_copy**(const FN_attrmodlist_t **modlist*);

FN_attrmodlist_t ***fn_attrmodlist_assign**(FN_attrmodlist_t **dst*, const FN_attrmodlist_t **src*);

unsigned int **fn_attrmodlist_count**(const FN_attrmodlist_t **modlist*);

const FN_attribute_t ***fn_attrmodlist_first**(const FN_attrmodlist_t **modlist*, void ***iter_pos*, unsigned int **first_mod_op*);

const FN_attribute_t ***fn_attrmodlist_next**(const FN_attrmodlist_t **modlist*, void ***iter_pos*, unsigned int **mod_op*);

int **fn_attrmodlist_add**(FN_attrmodlist_t **modlist*, unsigned int *mod_op*, const FN_attribute_t **attr*);

## DESCRIPTION

An attribute modification list allows for multiple modification operations to be made on the attributes associated with a single named object. It is used in the `fn_attr_multi_modify`(3XFN) operation.

An attribute modification list is a list of attribute modification specifiers. An attribute modification specifier consists of an attribute object and an operation specifier. The attribute's identifier indicates the attribute that is to be operated upon. The attribute's values are used in a manner depending on the operation. The operation specifier is an `unsigned int` that must have one of the values:

`FN_ATTR_OP_ADD`

`FN_ATTR_OP_ADD_EXCLUSIVE`

`FN_ATTR_OP_REMOVE`

`FN_ATTR_OP_ADD_VALUES`

or

`FN_ATTR_OP_REMOVE_VALUES`

(See `fn_attr_modify`(3XFN) for detailed descriptions of these specifiers.) The operations are to be performed in the order in which they appear in the modification list.

`fn_attrmodlist_create( )` creates an empty attribute modification list. `fn_attrmodlist_destroy( )` releases the storage associated with *modlist* . `fn_attrmodlist_copy( )` returns a copy of the attribute modification list *modlist* . `fn_attrmodlist_assign( )` makes a copy of *src* and assigns it to *dst* , releasing any old contents of *dst* . It returns a pointer to the same object as *dst* .

`fn_attrmodlist_count( )` returns the number attribute modification items in the attribute modification list.

The iterators `fn_attrmodlist_first( )` and `fn_attrmodlist_next( )` return a handle to the attribute part of the modification and return the operation specifier part through an `unsigned int *` parameter. `fn_attrmodlist_first( )` returns the attribute of the first modification item from *modlist* and sets *mod_op* to be the code of the modification operation of that item; *iter_pos* is set after the first modification item.

`fn_attrmodlist_next( )` returns the attribute of the next modification item from *modlist* after *iter_pos* and advances *iter_pos* ; *mod_op* is set to the code of the modification operation of that item. The order of the items returned during an enumeration is the same as the order by which the items were added to the modification list.

`fn_attrmodlist_add( )` adds a new item consisting of the given modification operation code *mod_op* and attribute *attr* to the end of the modification list *modlist* . *attr* 's identifier indicates the attribute that is to be operated upon. *attr* 's values are used in a manner depending on the operation.

## RETURN VALUES

`fn_attrmodlist_first( )` returns `0` if the modification list is empty. `fn_attrmodlist_next( )` returns `0` if there are no more items on the modification list to be enumerated or if the iteration marker is invalid.

`fn_attrmodlist_add( )` returns `1` if the operation succeeds, `0` if the operation fails.

## USAGE

Manipulation of attributes using the operations described in this manual page does not affect their representation in the underlying naming system. Changes to attributes in the underlying naming system can only be effected through the use of the interfaces described in `xfn_attributes`(3XFN) .

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`FN_attribute_t`(3XFN) , `FN_attrset_t`(3XFN) , `FN_identifier_t`(3XFN) , `fn_attr_modify`(3XFN) , `fn_attr_multi_modify`(3XFN) , `xfn`(3XFN) , `xfn_attributes`(3XFN) , `attributes`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

## NAME

fn_attr_multi_get, FN_multigetlist_t, fn_multigetlist_next, fn_multigetlist_destroy – return multiple attributes associated with named object

## SYNOPSIS

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>

FN_multigetlist_t ***fn_attr_multi_get**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, const FN_attrset_t **attr_ids*, unsigned int *follow_link*, FN_status_t **status*);

FN_attribute_t ***fn_multigetlist_next**(FN_multigetlist_t **ml*, FN_status_t **status*);

void **fn_multigetlist_destroy**(FN_multigetlist_t **ml*, FN_status_t **status*);

## DESCRIPTION

This set of operations returns one or more attributes associated with the object named by *name* relative to the context *ctx* . If *name* is empty, the attributes associated with *ctx* are returned.

The value of *follow_link* determines what happens when the terminal atomic part of *name* is bound to an XFN link. If *follow_link* is non-zero, such a link is followed, and the values of the attribute associated with the final named object are returned; if *follow_link* is zero, such a link is not followed. Any XFN links encountered before the terminal atomic name are always followed.

The attributes returned are those specified in *attr_ids* . If the value of *attr_ids* is `0` , all attributes associated with the named object are returned. Any attribute values in *attr_ids* provided by the caller are ignored; only the attribute identifiers are relevant for this operation. Each attribute (identifier, syntax, values) is returned one at a time using an enumeration scheme similar to that for listing a context.

`fn_attr_multi_get( )` initiates the enumeration process. It returns a handle to an `FN_multigetlist_t` object that can be used for the enumeration.

The operation `fn_multigetlist_next( )` returns a new `FN_attribute_t` object containing the next attribute (identifiers, syntaxes, and values) requested and updates *ml* to indicate the state of the enumeration.

The operation `fn_multigetlist_destroy( )` releases the resources used during the enumeration. It may be invoked before the enumeration has completed to terminate the enumeration.

## RETURN VALUES

`fn_attr_multi_get( )` returns a pointer to an `FN_multigetlist_t` object if the enumeration has been initiated successfully; a `NULL` pointer (`0` ) is returned if it failed.

`fn_multigetlist_next( )` returns a pointer to an `FN_attribute_t` object if an attribute was returned, a `NULL` pointer (`0` ) if no attribute was returned.

In the case of a failure, these operations set *status* to indicate the nature of the failure.

**ERRORS**

Each call to `fn_multigetlist_next()` sets status as follows:

| | |
|---|---|
| `FN_SUCCESS` | If an attribute was returned, there are more attributes to be enumerated. If no attribute was returned, the enumeration has completed successfully. |
| `FN_E_ATTR_NO_PERMISSION` | The caller did not have permission to read this attribute. |
| `FN_E_INSUFFICIENT_RESOURCES` | Insufficient resources are available to return the attribute's values. |
| `FN_E_INVALID_ATTR_IDENTIFIER` | This attribute identifier was not in a format acceptable to the naming system, or its contents was not valid for the format specified for the identifier. |
| `FN_E_INVALID_ENUM_HANDLE` | (No attribute should be returned with this status code). The given enumeration handle is not valid. Possible reasons could be that the handle was from another enumeration, or the object being processed no longer accepts the handle (due to such events as handle expiration or updates to the object's attribute set). |
| `FN_E_NO_SUCH_ATTRIBUTE` | The object did not have an attribute with the given identifier. |
| `FN_E_PARTIAL_RESULT` | (No attribute should be returned with this status code). The enumeration is not yet complete but cannot be continued. |

For `FN_E_ATTR_NO_PERMISSION`, `FN_E_INVALID_ATTR_IDENTIFIER`, `FN_E_INSUFFICIENT_RESOURCES`, or `FN_E_NO_SUCH_ATTRIBUTE`, the returned attribute contains only the attribute identifier (no value or syntax). For these four status codes and `FN_SUCCESS` (when an attribute was returned), `fn_multigetlist_next()` can be called again to return another attribute. All other status codes indicate that no more attributes can be returned by `fn_multigetlist_next()`.

Other status codes, such as `FN_E_COMMUNICATION_FAILURE,` are also possible, in which case, no attribute is returned. In such cases, *status* is set as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN) .

**USAGE**

Implementations are not required to return all attributes requested by *attr_ids* . Some may choose to return only the attributes found successfully, followed by a status of `FN_E_PARTIAL_RESULT;` such implementations may not necessarily return attributes identifying those that could not be read. Implementations are not required to return the attributes in any order.

There may be a relationship between the *ctx* argument supplied to `fn_attr_multi_get()` and the `FN_multigetlist_t` object it returns. For example, some implementations may store the context handle *ctx* within the `FN_multigetlist_t` object for subsequent `fn_multigetlist_next()` calls. In general, a `fn_ctx_handle_destroy()` should not be invoked on *ctx* until the enumeration has terminated.

**EXAMPLES**

**EXAMPLE 1** A sample program displaying how to use `fn_attr_multi_get()` function.

The following code fragment illustrates to obtain all attributes associated with a given name using the `fn_attr_multi_get()` operations.

```
/* list all attributes associated with given name */
extern FN_string_t *input_string;
FN_ctx_t  *ctx;
FN_composite_name_t *target_name = fn_composite_name_from_string(input_string);
FN_multigetlist_t *ml;
FN_status_t  *status = fn_status_create();
FN_attribute_t *attr;
int done = 0;
ctx = fn_ctx_handle_from_initial(status);
/* error checking on ’status’ */
/* attr_ids == 0 indicates all attributes are to be returned */
if ((ml=fn_attr_multi_get(ctx, target_name, 0, status)) == 0) {
 /* report ’status’ and exit */
}
while ((attr=fn_multigetlist_next(ml, status)) && !done) {
 switch (fn_status_code(status)) {
 case FN_SUCCESS:
  /* do something with ’attr’ */
  break;
 case FN_E_ATTR_NO_PERMISSION:
 case FN_E_ATTR_INVALID_ATTR_IDENTIFIER:
 case FN_E_NO_SUCH_ATTRIBUTE:
  /* report error using identifier in ’attr’ */
  break;
 default:
  /* other error handling */
  done = 1;
 }
 if (attr)
  fn_attribute_destroy(attr);
}
/* check 'status' for reason for end of enumeration and report if necessary */
/* clean up */
fn_multigetlist_destroy(ml, status);
/* report 'status' */
```

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_attribute_t`(3XFN) , `FN_attrset_t`(3XFN) , `FN_composite_name_t`(3XFN) , `FN_ctx_t`(3XFN) , `FN_identifier_t`(3XFN) , `FN_status_t`(3XFN) , `fn_attr_get`(3XFN) , `fn_ctx_handle_destroy`(3XFN) , `fn_ctx_list_names`(3XFN) , `xfn`(3XFN) , `xfn_attributes`(3XFN) , `xfn_status_codes`(3XFN) , `attributes`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

fn_attr_multi_modify – modify multiple attributes associated with named object

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>

int **`fn_attr_multi_modify`**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, const FN_attrmodlist_t **mods*, unsigned int *follow_link*, FN_attrmodlist_t ***unexecuted_mods*, FN_status_t **status*);

**DESCRIPTION**

This operation modifies the attributes associated with the object named *name* relative to *ctx*. If *name* is empty, the attributes associated with *ctx* are modified.

The value of *follow_link* determines what happens when the terminal atomic part of *name* is bound to an XFN link. If *follow_link* is non-zero, such a link is followed, and the values of the attribute associated with the final named object are returned; if *follow_link* is zero, such a link is not followed. Any XFN links encountered before the terminal

In the *mods* parameter, the caller specifies a sequence of modifications that are to be done in order on the attributes. Each modification in the sequence specifies a modification operation code (see `fn_attr_modify`(3XFN)) and an attribute on which to operate.

The `FN_attrmodlist_t` type is described in `FN_attrmodlist_t`(3XFN).

**RETURN VALUES**

`fn_attr_multi_modify( )` returns `1` if all the modification operations were performed successfully. The function returns `0` if it any error occurs. If the operation fails, *status* and *unexecuted_mods* are set as described below.

**ERRORS**

If an error is encountered while performing the list of modifications, *status* indicates the type of error and *unexecuted_mods* is set to a list of unexecuted modifications. The contents of *unexecuted_mods* do not share any state with *mods*; items in *unexecuted_mods* are copies of items in *mods* and appear in the same order in which they were originally supplied in *mods*. The first operation in *unexecuted_mods* is the first one that failed and the code in *status* applies to this modification operation in particular. If *status* indicates failure and a `NULL` pointer (`0`) is returned in *unexecuted_mods*, that indicates no modifications were executed.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_attrmodlist_t`(3XFN), `FN_composite_name_t`(3XFN), `FN_ctx_t`(3XFN), `FN_status_t`(3XFN), `fn_attr_modify`(3XFN), `xfn`(3XFN), `xfn_attributes`(3XFN), `xfn_status_codes`(3XFN), `attributes`(5)

## NOTES

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

fn_attr_search, FN_searchlist_t, fn_searchlist_next, fn_searchlist_destroy – search for the atomic name of objects with the specified attributes in a single context

**SYNOPSIS**

#include <xfn/xfn.h>
FN_searchlist_t ***fn_attr_search**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, const FN_attrset_t **match_attrs*, unsigned int *return_ref*, const FN_attrset_t **return_attr_ids*, FN_status_t **status*);

FN_string_t ***fn_searchlist_next**(FN_searchlist_t **sl*, FN_ref_t ***returned_ref*, FN_attrset_t ***returned_attrs*, FN_status_t **status*);

void **fn_searchlist_destroy**(FN_searchlist_t **sl*);

**DESCRIPTION**

This set of operations is used to enumerate names of objects bound in the target context named *name* relative to the context *ctx* with attributes whose values match all those specified by *match_attrs* .

The attributes specified by *match_attrs* form a conjunctive AND expression against which the attributes of each named object in the target context are evaluated. For multi-valued attributes, the list order of values is ignored and attribute values not specified in *match_attrs* are ignored. If no value is specified for an attribute in *match_attrs* , the presence of the attribute is tested. If the value of *match_attrs* is 0 , all names in the target context are enumerated.

If a non-zero value of *return_ref* is passed to fn_attr_search( ) , the reference bound to the name is returned in the *returned_ref* argument to fn_searchlist_next( ) .

Attribute identifiers and values associated with named objects that satisfy *match_attrs* may be returned by fn_searchlist_next( ) . The attributes returned are those listed in the *return_attr_ids* argument to fn_attr_search( ) . If the value of *return_attr_ids* is 0 , all attributes are returned. If *return_attr_ids* is an empty FN_attrset_t(3XFN) object, no attributes are returned. Any attribute values in *return_attr_ids* are ignored; only the attribute identifiers are relevant for *return_attr_ids* .

The call to fn_attr_search( ) initiates the enumeration process. It returns a handle to an FN_searchlist_t object that is used to enumerate the names of the objects whose attributes match the attributes specified by *match_attrs* .

The operation fn_searchlist_next( ) returns the next name in the enumeration identified by the *sl* . The reference of the name is returned in *returned_ref* if *return_ref* was set in the call to fn_attr_search( ) . The attributes specified by *return_attr_ids* are returned in *returned_attrs* . fn_searchlist_next( ) also updates *sl* to indicate the state of the enumeration. Successive calls to fn_searchlist_next( ) using *sl* return successive names, and optionally, references and attributes, in the enumeration; these calls further update the state of the enumeration.

`fn_searchlist_destroy()` releases resources used during the enumeration. This can be invoked at any time to terminate the enumeration.

`fn_attr_search()` does not follow XFN links that are bound in the target context.

**RETURN VALUES**

`fn_attr_search()` returns a pointer to an `FN_searchlist_t` object if the enumeration is successfully initiated; it returns a `NULL` pointer if the enumeration cannot be initiated or if no named object with attributes whose values match those specified in *match_attrs* is found.

`fn_searchlist_next()` returns a pointer to an `FN_string_t`(3XFN) object; it returns a `NULL` pointer if no more names can be returned in the enumeration. If *returned_ref* is a `NULL` pointer, or if the *return_ref* parameter to *fn_attr_search* was `0` , no reference is returned; otherwise, *returned_ref* contains the reference bound to the name. If *returned_attrs* is a `NULL` pointer, no attributes are returned; otherwise, *returned_attrs* contains the attributes associated with the named object, as specified by the *return_attr_ids* parameter to `fn_attr_search()` .

In the case of a failure, these operations return in the *status* argument a code indicating the nature of the failure.

**ERRORS**

`fn_attr_search()` returns a `NULL` pointer if the enumeration could not be initiated. The *status* argument is set in the following way:

| | |
|---|---|
| `FN_SUCCESS` | A named object could not be found whose attributes satisfied the implied filter of equality and conjunction. |
| `FN_E_ATTR_NO_PERMISSION` | The caller did not have permission to read one or more of the specified attributes. |
| `FN_E_INVALID_ATTR_VALUE` | A value type in the specified attributes did not match the syntax of the attribute against which it was being evaluated. |

Other status codes are possible as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN) .

Each successful call to `fn_searchlist_next()` returns a name and, optionally, the reference and requested attributes. *status* is set in the following way:

| | |
|---|---|
| `FN_SUCCESS` | All requested attributes were returned successfully with the name. |

| | |
|---|---|
| `FN_E_ATTR_NO_PERMISSION` | The caller did not have permission to read one or more of the requested attributes. |
| `FN_E_INVALID_ATTR_IDENTIFIER` | A requested attribute identifier was not in a format acceptable to the naming system, or its contents was not valid for the format specified. |
| `FN_E_NO_SUCH_ATTRIBUTE` | The named object did not have one of the requested attributes. |
| `FN_E_INSUFFICIENT_RESOURCES` | Insufficient resources are available to return all the requested attributes and their values. |

`FN_E_ATTR_NO_PERMISSION`
`FN_E_INVALID_ATTR_IDENTIFIER`
`FN_E_NO_SUCH_ATTRIBUTE`
`FN_E_INSUFFICIENT_RESOURCES` These indicate that some of the requested attributes may have been returned in *returned_attrs* but one or more of them could not be returned. Use `fn_attr_get`(3XFN) or `fn_attr_multi_get`(3XFN) to discover why these attributes could not be returned.

`fn_searchlist_next()` returns a `NULL` pointer if no more names can be returned. The status argument is set in the following way:

| | |
|---|---|
| `FN_SUCCESS` | The search has completed successfully. |
| `FN_E_PARTIAL_RESULT` | The enumeration is not yet complete but cannot be continued. |
| `FN_E_ATTR_NO_PERMISSION` | The caller did not have permission to read one or more of the specified attributes. |
| `FN_E_INVALID_ENUM_HANDLE` | The supplied enumeration handle was not valid. Possible reasons could be that the handle was from another enumeration, or the context being enumerated no longer accepts the handle (due to such events as handle expiration or updates to the context). |

Other status codes are possible as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN) .

**USAGE**

The names enumerated using `fn_searchlist_next( )` are not ordered in any way. Furthermore, there is no guarantee that any two series of enumerations on the same context with identical *match_attrs* will return the names in the same order.

**EXAMPLES**

**EXAMPLE 1** A sample program of displaying how to use `fn_attr_search( )` function.

The following code fragment illustrates how the `fn_attr_search( )` operation may be used. The code consists of three parts: preparing the arguments for the search, performing the search, and cleaning up.

The first part involves getting the name of the context to start the search and constructing the set of attributes that named objects in the context must satisfy. This is done in the declarations part of the code and by the routine `get_search_query` .

The next part involves doing the search and enumerating the results of the search. This is done by first getting a context handle to the Initial Context, and then passing that handle along with the name of the target context and matching attributes to `fn_attr_search( )` . This particular call to `fn_attr_search( )` is requesting that no reference be returned (by passing in `0` for *return_ref* ), and that all attributes associated with the named object be returned (by passing in `0` as the *return_attr_ids* argument). If successful, `fn_attr_search( )` returns *sl* , a handle for enumerating the results of the search. The results of the search are enumerated using calls to `fn_searchlist_next( )` , which returns the name of the object and the attributes associated with the named object in *returned_attrs* .

The last part of the code involves cleaning up the resources used during the search and enumeration. The call to `fn_searchlist_destroy( )` releases resources reserved for this enumeration. The other calls release the context handle, name, attribute set, and status objects created earlier.

```
/* Declarations */
FN_ctx_t *ctx;
FN_searchlist_t *sl;
FN_string_t *name;
FN_attrset_t *returned_attrs;
FN_status_t *status = fn_status_create();
FN_composite_name_t *target_name = get_name_from_user_input();
FN_attrset_t *match_attrs = get_search_query();
/* Get context handle to Initial Context */
ctx = fn_ctx_handle_from_initial(status);
/* error checking on 'status' */
/* Initiate search */
if ((sl=fn_attr_search(ctx, target_name, match_attrs,
 /* no reference */ 0, /* return all attrs */ 0, status)) == 0) {
 /* report 'status', cleanup, and exit */
}
/* Enumerate names and attributes requested */
while (name=fn_searchlist_next(sl, 0, &returned_attrs, status)) {
 /* do something with ’name’ and ’returned_attrs’*/
 fn_string_destroy(name);
 fn_attrset_destroy(returned_attrs);
}
/* check ’status’ for reason for end of enumeration */
/* Clean up */
fn_searchlist_destroy(sl); /* Free resources of ’sl’ */
fn_status_destroy(status);
fn_attrset_destroy(match_attrs);
fn_ctx_handle_destroy(ctx);
fn_composite_name_destroy(target_name);
/*
 * Procedure for constructing attribute set containing
 * attributes to be matched:
 *  "zip_code" attribute value is "02158"
 *  AND "employed" attribute is present.
*/
FN_attrset_t *
get_search_query()
{
 /* Zip code and employed attribute identifier, syntax */
 extern FN_attribute_t   *attr_zip_code;
 extern FN_attribute_t   *attr_employed;
 FN_attribute_t *zip_code = fn_attribute_copy(attr_zip_code);
 FN_attr_value_t zc_value = {5, "02158"};
 FN_attrset_t *match_attrs = fn_attrset_create();
 fn_attribute_add(zip_code, &zc_value, 0);
 fn_attrset_add(match_attrs, zip_code, 0);
 fn_attrset_add(match_attrs, attr_employed, 0);
 return (match_attrs);
}
```

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`FN_attribute_t`(3XFN) , `FN_attrset_t`(3XFN) , `FN_attrvalue_t`(3XFN) , `FN_composite_name_t`(3XFN) , `FN_ctx_t`(3XFN) , `FN_status_t`(3XFN) , `FN_string_t`(3XFN) , `fn_attr_ext_search`(3XFN) , `fn_attr_get`(3XFN) , `fn_attr_multi_get`(3XFN) , `fn_ctx_list_names`(3XFN) , `xfn_status_codes`(3XFN) , `attributes`(5)

**NAME**

FN_attrset_t, fn_attrset_create, fn_attrset_destroy, fn_attrset_copy, fn_attrset_assign, fn_attrset_get, fn_attrset_count, fn_attrset_first, fn_attrset_next, fn_attrset_add, fn_attrset_remove – a set of XFN attributes

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]

#include <xfn/xfn.h>

FN_attrset_t ***fn_attrset_create**(void);

void **fn_attrset_destroy**(FN_attrset_t **aset*);

FN_attrset_t ***fn_attrset_copy**(constFN_attrset_t **aset*);

FN_attrset_t ***fn_attrset_assign**(FN_attrset_t **dst*, const FN_attrset_t **src*);

const FN_attribute_t ***fn_attrset_get**(constconst FN_attrset_t **aset*, const FN_identifier_t **attr_id*);

unsigned int **fn_attrset_count**(constFN_attrset_t **aset*);

const FN_attribute_t ***fn_attrset_first**(constFN_attrset_t **aset*, void ***iter_pos*);

const FN_attribute_t ***fn_attrset_next**(constFN_attrset_t **aset*, void ***iter_pos*);

int **fn_attrset_add**(FN_attrset_t **aset*, const FN_attribute_t **attr*, unsigned int *exclusive*);

int **fn_attrset_remove**(FN_attrset_t **aset*, const FN_identifier_t **attr_id*);

**DESCRIPTION**

An attribute set is a set of attribute objects with distinct identifiers. The `fn_attr_multi_get`(3XFN) operation takes an attribute set as parameter and returns an attribute set. The `fn_attr_get_ids`(3XFN) operation returns an attribute set containing the identifiers of the attributes.

Attribute sets are represented by the type `FN_attrset_t` . The following operations are defined for manipulating attribute sets.

`fn_attrset_create( )` creates an empty attribute set. `fn_attrset_destroy( )` releases the storage associated with the attribute set `aset` . `fn_attrset_copy( )` returns a copy of the attribute set `aset` . `fn_attrset_assign( )` makes a copy of the attribute set *src* and assigns it to *dst* , releasing any old contents of *dst* . A pointer to the same object as *dst* is returned.

`fn_attrset_get( )` returns the attribute with the given identifier *attr_id* from `aset` . `fn_attrset_count( )` returns the number attributes found in the attribute set `aset` .

`fn_attrset_first( )` and `fn_attrset_next( )` are functions that can be used to return an enumeration of all the attributes in an attribute set. The attributes are not ordered in any way. There is no guaranteed relation between the order in which items are added to an attribute set and the order of the enumeration. The specification does guarantee that any two enumerations will return the members in the same order, provided that no `fn_attrset_add()` or `fn_attrset_remove()` operation was performed on the object in between or during the two enumerations. `fn_attrset_first()` returns the first attribute from the set and sets *iter_pos* after the first attribute. `fn_attrset_next ()` returns the attribute following *iter_pos* and advances *iter_pos* .

`fn_attrset_add()` adds the attribute *attr* to the attribute set `aset` , replacing the attribute's values if the identifier of *attr* is not distinct in `aset` and *exclusive* is `0` . If *exclusive* is non-zero and the identifier of *attr* is not distinct in `aset` , the operation fails.

`fn_attrset_remove()` removes the attribute with the identifier *attr_id* from `aset` . The operation succeeds even if no such attribute occurs in `aset` .

**RETURN VALUES**

`fn_attrset_first()` returns `0` if the attribute set is empty. `fn_attrset_next()` returns `0` if there are no more attributes in the set.

`fn_attrset_add()` and `fn_attrset_remove()` return `1` if the operation succeeds, and `0` if the operation fails.

**USAGE**

Manipulation of attributes using the operations described in this manual page does not affect their representation in the underlying naming system. Changes to attributes in the underlying naming system can only be effected through the use of the interfaces described in `xfn_attributes`(3XFN) .

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_attribute_t`(3XFN) , `FN_attrvalue_t`(3XFN) , `FN_identifier_t`(3XFN) , `fn_attr_get_ids`(3XFN) , `fn_attr_multi_get`(3XFN) , `xfn`(3XFN) , `xfn_attributes`(3XFN) , `attributes`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

FN_attrvalue_t – an XFN attribute value

**SYNOPSIS**

**cc** [ *flag* ... ] *file* ... –lxfn [ *library* ... ]

#include <xfn/xfn.h>

**DESCRIPTION**

The type FN_attrvalue_t is used to represent the contents of a single attribute value, within an attribute of type FN_attribute_t.

The representation of this structure is defined by XFN as follows:

```
typedef struct { size_t length;
void *contents; } FN_attrvalue_t;
```

**SEE ALSO**

FN_attribute_t(3XFN), fn_attr_get_values(3XFN), xfn(3XFN)

**NAME**

FN_composite_name_t, fn_composite_name_create, fn_composite_name_destroy, fn_composite_name_from_str, fn_composite_name_from_string, fn_string_from_composite_name, fn_composite_name_copy, fn_composite_name_assign, fn_composite_name_is_empty, fn_composite_name_count, fn_composite_name_first, fn_composite_name_next, fn_composite_name_prev, fn_composite_name_last, fn_composite_name_prefix, fn_composite_name_suffix, fn_composite_name_is_equal, fn_composite_name_is_prefix, fn_composite_name_is_suffix, fn_composite_name_prepend_comp, fn_composite_name_append_comp, fn_composite_name_insert_comp, fn_composite_name_delete_comp, fn_composite_name_prepend_name, fn_composite_name_append_name, fn_composite_name_insert_name – a sequence of component names spanning multiple naming systems

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>
FN_composite_name_t ***fn_composite_name_create**(void);

void **fn_composite_name_destroy**(FN_composite_name_t **name*);

FN_composite_name_t ***fn_composite_name_from_str**(const unsigned char **cstr*);

FN_composite_name_t ***fn_composite_name_from_string**(const FN_string_t **str*);

FN_string_t ***fn_string_from_composite_name**(const FN_composite_name_t **name*, unsigned int **status*);

FN_composite_name_t ***fn_composite_name_copy**(const FN_composite_name_t **name*);

FN_composite_name_t ***fn_composite_name_assign**(FN_composite_name_t **dst*, const FN_composite_name_t **src*);

int **fn_composite_name_is_empty**(const FN_composite_name_t **name*);

unsigned int **fn_composite_name_count**(const FN_composite_name_t **name*);

const FN_string_t ***fn_composite_name_first**(const FN_composite_name_t **name*, void ***iter_pos*);

const FN_string_t ***fn_composite_name_next**(const FN_composite_name_t **name*, void ***iter_pos*);

const FN_string_t ***fn_composite_name_prev**(const FN_composite_name_t **name*, void ***iter_pos*);

const FN_string_t ***fn_composite_name_last**(const FN_composite_name_t **name*, void ***iter_pos*);

FN_composite_name_t ***fn_composite_name_prefix**(const FN_composite_name_t **name*, const void **iter_pos*);

FN_composite_name_t ***fn_composite_name_suffix**(const FN_composite_name_t **name*, const void **iter_pos*);

int **fn_composite_name_is_equal**(const FN_composite_name_t **name*, const FN_composite_name_t **name2*, unsigned int **status*);

int **fn_composite_name_is_prefix**(const FN_composite_name_t **name*, const FN_composite_name_t **prefix*, void ***iter_pos*, unsigned int **status*);

int **fn_composite_name_is_suffix**(const FN_composite_name_t **name*, const FN_composite_name_t **suffix*, void ***iter_pos*, unsigned int **status*);

int **fn_composite_name_prepend_comp**(FN_composite_name_t **name*, const FN_string_t **newcomp*);

int **fn_composite_name_append_comp**(FN_composite_name_t **name*, const FN_string_t **newcomp*);

int **fn_composite_name_insert_comp**(FN_composite_name_t **name*, void ***iter_pos*, const FN_string_t **newcomp*);

int **fn_composite_name_delete_comp**(FN_composite_name_t **name*, void ***iter_pos*);

int **fn_composite_name_prepend_name**(FN_composite_name_t **name*, const FN_composite_name_t **newcomps*);

int **fn_composite_name_append_name**(FN_composite_name_t **name*, const FN_composite_name_t **newcomps*);

int **fn_composite_name_insert_name**(FN_composite_name_t **name*, void ***iter_pos*, const FN_composite_name_t **newcomps*);

## DESCRIPTION

A composite name is represented by an object of type `FN_composite_name_t` . Each component is a string name, of type `FN_string_t` , from the namespace of a single naming system. It may be an atomic name or a compound name in that namespace.

`fn_composite_name_create` creates an `FN_composite_name_t` object with zero components. Components may be subsequently added to the composite name using the modify operations described below. `fn_composite_name_destroy` releases any storage associated with the given `FN_composite_name_t` handle.

`fn_composite_name_from_str( )` creates an `FN_composite_name_t` from the given null-terminated string based on the code set of the current locale setting, using the XFN composite name syntax. `fn_composite_name_from_string( )` creates an `FN_composite_name_t` from the string *str* using the XFN composite name syntax. `fn_string_from_composite_name( )` returns the standard string form of the given composite name, by concatenating the components of the composite name in a left to right order, each separated by the XFN component separator.

`fn_composite_name_copy( )` returns a copy of the given composite name object. `fn_composite_name_assign( )` makes a copy of the composite name object pointed to by *src* and assigns it to *dst* , releasing any old contents of *dst* . A pointer to the same object as *dst* is returned.

`fn_composite_name_is_empty( )` returns `1` if the given composite name is an empty composite name (that is, it consists of a single, empty component name); otherwise, it returns `0` . `fn_composite_name_count( )` returns the number of components in the given composite name.

The iteration scheme is based on the exchange of an opaque `void *` argument, *iter_pos* , that serves to record the position of the iteration in the sequence. Conceptually, *iter_pos* records a position between two successive components (or at one of the extreme ends of the sequence).

The function `fn_composite_name_first( )` returns a handle to the `FN_string_t` that is the first component in the name, and sets *iter_pos* to indicate the position immediately following the first component. It returns `0` if the name has no components. Thereafter, successive calls of the `fn_composite_name_next( )` function return pointers to the component following the iteration marker, and advance the iteration marker. If the iteration marker is at the end of the sequence, `fn_composite_name_next( )` returns `0` . Similarly, `fn_composite_name_prev( )` returns the component preceding the iteration pointer and moves the marker back one component. If the marker is already at the beginning of the sequence, `fn_composite_name_prev( )` returns `0` . The function `fn_composite_name_last( )` returns a pointer to the last component of the name and sets the iteration marker immediately preceding this component (so that subsequent calls to `fn_composite_name_prev( )` can be used to step through leading components of the name).

The `fn_composite_name_suffix( )` function returns a composite name consisting of a copy of those components following the supplied iteration marker. The method `fn_composite_name_prefix( )` returns a composite name consisting of those components that precede the iteration marker. Using these functions with an iteration marker that was not initialized using `fn_composite_name_first( )` , `fn_composite_name_last( )` , `fn_composite_name_is_prefix( )` , or `fn_composite_name_is_suffix( )` yields undefined and generally undesirable behavior.

The functions `fn_composite_name_is_equal( )` , `fn_composite_name_is_prefix( )` , and `fn_composite_name_is_suffix( )` test for equality between composite names or between parts of composite names. For these functions, equality is defined as exact string equality, not name equivalence. A name's syntactic property, such as case-insensitivity, is not taken into account by these functions.

The function `fn_composite_name_is_prefix( )` tests if one composite name is a prefix of another. If so, it returns `1` and sets the iteration marker immediately following the prefix. (For example, a subsequent call to `fn_composite_name_suffix( )` will return the remainder of the name.) Otherwise, it returns `0` and the value of the iteration marker is undefined. The function `fn_composite_name_is_suffix( )` is similar. It tests if one composite name is a suffix of another. If so, it returns `1` and sets the iteration marker immediately preceding the suffix.

The functions `fn_composite_name_prepend_comp( )` and `fn_composite_name_append_comp( )` prepend and append a single component to the given composite name, respectively. These operations invalidate any iteration marker the client holds for that object. `fn_composite_name_insert_comp( )` inserts a single component before *iter_pos* to the given composite name and sets *iter_pos* to be immediately after the component just inserted. `fn_composite_name_delete_comp( )` deletes the component located before *iter_pos* from the given composite name and sets *iter_pos* back one component.

The functions `fn_composite_name_prepend_name( )` , `fn_composite_name_append_name( )` , and `fn_composite_name_insert_name( )` perform the same update functions as their *_comp* counterparts, respectively, except that multiple components are being added, rather than single components. For example, `fn_composite_name_insert_name( )` sets *iter_pos* to be immediately after the name just added.

## RETURN VALUES

The functions `fn_composite_name_is_empty( )` , `fn_composite_name_is_equal( )` , `fn_composite_name_is_suffix( )` , and `fn_composite_name_is_prefix( )` return `1` if the test indicated is true; `0` otherwise.

The update functions `fn_composite_name_prepend_comp( )` , `fn_composite_name_append_comp( )` , `fn_composite_name_insert_comp( )` , `fn_composite_name_delete_comp( )` , and their *_name* counterparts return `1` if the update was successful; `0` otherwise.

If a function is expected to return a pointer to an object, a `NULL` pointer (`0` ) is returned if the function fails.

**ERRORS**

Code set mismatches that occur during the composition of the string form or during comparisons of composite names are resolved in an implementation-dependent way. `fn_string_from_composite_name()`, `fn_composite_name_is_equal()`, `fn_composite_name_is_suffix()`, and `fn_composite_name_is_prefix()` set *status* to `FN_E_INCOMPATIBLE_CODE_SETS` for composite names whose components have code sets that are determined by the implementation to be incompatible.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_string_t`(3XFN), `xfn`(3XFN), `attributes`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

FN_compound_name_t, fn_compound_name_from_syntax_attrs, fn_compound_name_get_syntax_attrs, fn_compound_name_destroy, fn_string_from_compound_name, fn_compound_name_copy, fn_compound_name_assign, fn_compound_name_count, fn_compound_name_first, fn_compound_name_next, fn_compound_name_prev, fn_compound_name_last, fn_compound_name_prefix, fn_compound_name_suffix, fn_compound_name_is_empty, fn_compound_name_is_equal, fn_compound_name_is_prefix, fn_compound_name_is_suffix, fn_compound_name_prepend_comp, fn_compound_name_append_comp, fn_compound_name_insert_comp, fn_compound_name_delete_comp, fn_compound_name_delete_all – an XFN compound name

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>
FN_compound_name_t ***fn_compound_name_from_syntax_attrs**(const FN_attrset_t **aset*, const FN_string_t **name*, FN_status_t **status*);

FN_attrset_t ***fn_compound_name_get_syntax_attrs**(const FN_compound_name_t **name*);

void **fn_compound_name_destroy**(FN_compound_name_t **name*);

FN_string_t ***fn_string_from_compound_name**(const FN_compound_name_t **name*);

FN_compound_name_t ***fn_compound_name_copy**(const FN_compound_name_t **name*);

FN_compound_name_t ***fn_compound_name_assign**(FN_compound_name_t **dst*, const FN_compound_name_t **src*);

unsigned int **fn_compound_name_count**(const FN_compound_name_t **name*);

const FN_string_t ***fn_compound_name_first**(const FN_compound_name_t **name*, void ***iter_pos*);

const FN_string_t ***fn_compound_name_next**(const FN_compound_name_t **name*, void ***iter_pos*);

const FN_string_t ***fn_compound_name_prev**(const FN_compound_name_t **name*, void ***iter_pos*);

const FN_string_t ***fn_compound_name_last**(const FN_compound_name_t **name*, void ***iter_pos*);

FN_compound_name_t ***fn_compound_name_prefix**(const FN_compound_name_t **name*, const void **iter_pos*);

FN_compound_name_t ***fn_compound_name_suffix**(const FN_compound_name_t **name*, const void **iter_pos*);

int **fn_compound_name_is_empty**(const FN_compound_name_t **name*);

int **fn_compound_name_is_equal**(const FN_compound_name_t **name1*, const FN_compound_name_t **name2*, unsigned int **status*);

int **fn_compound_name_is_prefix**(const FN_compound_name_t **name*, const FN_compound_name_t **pre*, void ***iter_pos*, unsigned int **status*);

int **fn_compound_name_is_suffix**(const FN_compound_name_t **name*, const FN_compound_name_t **suffix*, void ***iter_pos*, unsigned int **status*);

int **fn_compound_name_prepend_comp**(FN_compound_name_t **name*, const FN_string_t **atomic_comp*, unsigned int **status*);

int **fn_compound_name_append_comp**(FN_compound_name_t **name*, const FN_string_t **atomic_comp*, unsigned int **status*);

int **fn_compound_name_insert_comp**(FN_compound_name_t **name*, void ***iter_pos*, const FN_string_t **atomic_comp*, unsigned int **status*);

int **fn_compound_name_delete_comp**(FN_compound_name_t **name*, void ***iter_pos*);

int **fn_compound_name_delete_all**(FN_compound_name_t **name*);

## DESCRIPTION

Most applications treat names as opaque data. Hence, the majority of clients of the XFN interface will not need to parse names. Some applications, however, such as browsers, need to parse names. For these applications, XFN provides support in the form of the FN_compound_name_t object.

Each naming system in an XFN federation potentially has its own naming conventions. The FN_compound_name_t object has associated operations for applications to process compound names that conform to the XFN model of expressing compound name syntax. The XFN syntax model for compound names covers a large number of specific name syntaxes and is expressed in terms of syntax properties of the naming convention. See xfn_compound_names(3XFN) .

An FN_compound_name_t object is constructed by the operation fn_compound_name_from_syntax_attrs , using a string name and an attribute set containing the "fn_syntax_type" (with identifier format FN_ID_STRING ) attribute identifying the namespace syntax of the string name. The value "standard" (with identifier format FN_ID_STRING ) in the "fn_syntax_type" specifies a syntax model that is by default supported by the FN_compound_name_t object. An implementation may support other syntax types instead of the XFN standard syntax model, in which case the value of the "fn_syntax_type" attribute would be set to an implementation-specific string. fn_compound_name_get_syntax_attrs( ) returns an attribute set containing the syntax attributes that describes the given compound name. fn_compound_name_destroy( ) releases the storage associated with the given compound name. `fn_string_from_compound_name( )` returns the string form of the given compound name. `fn_compound_name_copy( )` returns a copy of the given compound name. `fn_compound_name_assign( )` makes a copy of the compound name *src* and assigns it to *dst* , releasing any old contents of *dst* . A pointer to the object pointed to by *dst* is returned. `fn_compound_name_count( )` returns the number of atomic components in the given compound name.

The function `fn_compound_name_first( )` returns a handle to the `FN_string_t` that is the first atomic component in the compound name, and sets *iter_pos* to indicate the position immediately following the first component. It returns `0` if the name has no components. Thereafter, successive calls of the `fn_compound_name_next( )` function return pointers to the component following the iteration marker, and advance the iteration marker. If the iteration marker is at the end of the sequence, `fn_compound_name_next( )` returns `0` . Similarly, `fn_compound_name_prev( )` returns the component preceding the iteration pointer and moves the marker back one component. If the marker is already at the beginning of the sequence, `fn_compound_name_prev( )` returns `0` . The function `fn_compound_name_last( )` returns a pointer to the last component of the name and sets the iteration marker immediately preceding this component (so that subsequent calls to `fn_compound_name_prev( )` can be used to step through trailing components of the name).

The `fn_compound_name_suffix( )` function returns a compound name consisting of a copy of those components following the supplied iteration marker. The function `fn_compound_name_prefix( )` returns a compound name consisting of those components that precede the iteration marker. Using these functions with an iteration marker that was not initialized with the use of `fn_compound_name_first( )` , `fn_compound_name_last( )` , `fn_compound_name_is_prefix( )` , or `fn_compound_name_is_suffix( )` yields undefined and generally undesirable behavior.

The functions `fn_compound_name_is_equal( )` , `fn_compound_name_is_prefix( )` , and `fn_compound_name_is_suffix( )` test for equality between compound names or between parts of compound names. For these functions, equality is defined as name equivalence. A name's syntactic property, such as case-insensitivity, is taken into account by these functions.

The function `fn_compound_name_is_prefix( )` tests if one compound name is a prefix of another. If so, it returns `1` and sets the iteration marker immediately following the prefix. (For example, a subsequent call to `fn_compound_name_suffix( )` will return the remainder of the name.) Otherwise, it returns `0` and value of the iteration marker is undefined. The function `fn_compound_name_is_suffix( )` is similar. It tests if one compound name is a suffix of another. If so, it returns `1` and sets the iteration marker immediately preceding the suffix.

The functions `fn_compound_name_prepend_comp()` and `fn_compound_name_append_comp()` prepend and append a single atomic component to the given compound name, respectively. These operations invalidate any iteration marker the client holds for that object. `fn_compound_name_insert_comp()` inserts an atomic component before *iter_pos* to the given compound name and sets *iter_pos* to be immediately after the component just inserted. `fn_compound_name_delete_comp()` deletes the atomic component located before *iter_pos* from the given compound name and sets *iter_pos* back one component. `fn_compound_name_delete_all ()` deletes all the atomic components from *name* .

**RETURN VALUES**

The following test functions return `1` if the test indicated is true; otherwise, they return `0` :

`fn_compound_name_is_empty()`

`fn_compound_name_is_equal()`

`fn_compound_name_is_suffix()`

`fn_compound_name_is_prefix()`

The following update functions return `1` if the update was successful; otherwise, they return `0` :

`fn_compound_name_prepend_comp()`

`fn_compound_name_append_comp()`

`fn_compound_name_insert_comp()`

`fn_compound_name_delete_comp()`

`fn_compound_name_delete_all()`

If a function is expected to return a pointer to an object, a `NULL` pointer (`0` ) is returned if the function fails.

**ERRORS**

When the function `fn_compound_name_from_syntax_attrs()` fails, it returns a status code in *status* . The possible status codes are:

| | |
|---|---|
| `FN_E_ILLEGAL_NAME` | The name supplied to the operation was not a well- formed XFN compound name, or one of the component names was not well-formed according to the syntax of the naming system(s) involved in its resolution. |

| | |
|---|---|
| `FN_E_INCOMPATIBLE_CODE_SETS` | The code set of the given string is incompatible with that supported by the compound name. |
| `FN_E_INVALID_SYNTAX_ATTRS` | The syntax attributes supplied are invalid or insufficient to fully specify the syntax. |
| `FN_E_SYNTAX_NOT_SUPPORTED` | The syntax type specified is not supported. |

The following functions may return in *status* the status code `FN_E_INCOMPATIBLE_CODE_SETS` when the code set of the given string is incompatible with that of the compound name:

`fn_compound_name_is_equal( )`

`fn_compound_name_is_suffix( )`

`fn_compound_name_is_prefix( )`

`fn_compound_name_prepend_comp( )`

`fn_compound_name_append_comp( )`

`fn_compound_name_insert_comp( )`

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_attribute_t`(3XFN) , `FN_attrset_t`(3XFN) , `FN_composite_name_t`(3XFN) , `FN_status_t`(3XFN) , `FN_string_t`(3XFN) , `fn_ctx_get_syntax_attrs` (3XFN) , `xfn`(3XFN) , `xfn_compound_names`(3XFN) , `attributes`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

fn_ctx_bind – bind a reference to a name

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>

int **`fn_ctx_bind`**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, const FN_ref_t **ref*, unsigned int *exclusive*, FN_status_t **status*);

**DESCRIPTION**

This operation binds the supplied reference *ref* to the supplied composite name *name* relative to *ctx*. The binding is made in the target context, that is, the context named by all but the terminal atomic part of *name*. The operation binds the terminal atomic name to the supplied reference in the target context. The target context must already exist.

The value of *exclusive* determines what happens if the terminal atomic part of the name is already bound in the target context. If *exclusive* is nonzero and *name* is already bound, the operation fails. If *exclusive* is `0`, the new binding replaces any existing binding.

**RETURN VALUES**

When the bind operation is successful it returns `1`; on error it returns `0`.

**ERRORS**

`fn_ctx_bind` sets *status* as described in `FN_status_t`(3XFN) and `xfn_status_codes`. Of special relevance for this operation is the status code `FN_E_NAME_IN_USE,` which indicates that the supplied name is already in use.

**USAGE**

The value of *ref* cannot be `NULL`. If the intent is to reserve a name using `fn_ctx_bind( )`, a reference containing no address should be supplied. This reference may be name service-specific or it may be the conventional `NULL` reference defined in the X/Open registry (see `fns_references`(5)).

If multiple sources are updating a reference, they must synchronize amongst each other when adding, modifying, or removing from the address list of a bound reference.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_composite_name_t`(3XFN), `FN_ctx_t`(3XFN), `FN_ref_t`(3XFN), `FN_status_t`(3XFN), `fn_ctx_lookup`(3XFN), `fn_ctx_unbind`(3XFN), `xfn`(3XFN), `xfn_status_codes`(3XFN), `attributes`(5), `fns_references`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

fn_ctx_create_subcontext – create a subcontext in a context

**SYNOPSIS**

```
cc [ flag ... ] file ... –lxfn [ library ... ]
#include <xfn/xfn.h>

FN_ref_t *fn_ctx_create_subcontext(FN_ctx_t *ctx, const FN_composite_name_t
*name, FN_status_t *status);
```

**DESCRIPTION**

This operation creates a new XFN context of the same type as the target context — that named by all but the terminal atomic component of *name* — and binds it to the supplied composite name.

As with `fn_ctx_bind( ),` the target context must already exist. The new context is created and bound in the target context using the terminal atomic name in *name*. The operation returns a reference to the newly created context.

**RETURN VALUE**

`fn_ctx_create_subcontext( )` returns a reference to the newly created context; if the operation fails, it returns a `NULL` pointer (`0`).

**ERRORS**

`fn_ctx_create_subcontext( )` sets *status* as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN). Of special relevance for this operation is the following status code:

| | |
|---|---|
| `FN_E_NAME_IN_USE` | The terminal atomic name already exists in the target context. |

**APPLICATION USAGE**

The new subcontext is an XFN context and is created in the same naming system as the target context. The new subcontext also inherits the same syntax attributes as the target context. XFN does not specify any further properties of the new subcontext. The target context and its naming system determine these.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe. |

**SEE ALSO**

`FN_composite_name_t`(3XFN), `FN_ctx_t`(3XFN), `FN_ref_t`(3XFN), `FN_status_t`(3XFN), `fn_ctx_bind`(3XFN), `fn_ctx_lookup`(3XFN), `fn_ctx_destroy_subcontext`(3XFN), `xfn_status_codes`(3XFN), `xfn`(3XFN), `attributes`(5)

**NAME**

fn_ctx_destroy_subcontext – destroy the named context and remove its binding from the parent context

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>

int **`fn_ctx_destroy_subcontext`**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, FN_status_t **status*);

**DESCRIPTION**

This operation destroys the subcontext named by *name* relative to *ctx*, and unbinds the name.

As with `fn_ctx_unbind( ),` this operation succeeds even if the terminal atomic name is not bound in the target context — the context named by all but the terminal atomic name in *name*.

**RETURN VALUE**

`fn_ctx_destroy_subcontext( )` returns `1` on success and `0` on failure.

**ERRORS**

`fn_ctx_destroy_subcontext( )` sets *status* as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN). Of special relevance for `fn_ctx_destroy_subcontext( )` are the following status codes:

| | |
|---|---|
| `FN_E_CTX_NOT_A_CONTEXT` | *name* does not name a context. |
| `FN_E_CTX_NOT_EMPTY` | The naming system being asked to do the destroy does not support removal of a context that still contains bindings. |

**APPLICATION USAGE**

Some aspects of this operation are not specified by XFN, but are determined by the target context and its naming system. For example, XFN does not specify what happens if the named subcontext is non-empty when the operation is invoked.

In naming systems that support attributes, and store the attributes along with names or contexts, this operation removes the name, the context, and its associated attributes.

Normal resolution always follows links. In a `fn_ctx_destroy_subcontext( )` operation, resolution of *name* continues to the target context; the terminal atomic name is not resolved. If the terminal atomic name is bound to a link, the link is not followed and the operation fails with `FN_E_CTX_NOT_A_CONTEXT` because the name is not bound to a context.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe. |

**SEE ALSO**

`FN_ctx_t`(3XFN), `FN_composite_name_t`(3XFN), `FN_status_t`(3XFN), `fn_ctx_create_subcontext`(3XFN), `fn_ctx_unbind`(3XFN), `xfn`(3XFN), `xfn_status_codes`(3XFN), `attributes`(5)

**NAME**

fn_ctx_equivalent_name – construct an equivalent name in same context

**SYNOPSIS**

#include <xfn/xfn.h>

FN_composite_name_t ***fn_ctx_equivalent_name**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, const FN_string_t **leading_name*, FN_status_t * *status*);

**DESCRIPTION**

Given the name of an object *name* relative to the context *ctx*, this operation returns an equivalent name for that object, relative to the same context *ctx*, that has *leading_name* as its initial atomic name. Two names are said to be equivalent if they have prefixes that resolve to the same context, and the parts of the names immediately following the prefixes are identical.

The existence of a binding for *leading_name* in *ctx* does not guarantee that a name equivalent to *name* can be constructed. The failure may be because such equivalence is not meaningful, or due to the inability of the system to construct a name with the equivalence. For example, supplying `_thishost` as *leading_name* when *name* starts with `_myself` to `fn_ctx_equivalent_name()` in the Initial Context would not be meaningful; this results in the return of the error code `FN_E_NO_EQUIVALENT_NAME.`

**RETURN VALUES**

If an equivalent name cannot be constructed, the value `0` is returned and *status* is set appropriately.

**ERRORS**

`fn_ctx_equivalent_name()` sets *status* as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN). The following status code is especially relevant for this operation:

| | |
|---|---|
| `FN_E_NO_EQUIVALENT_NAME` | No equivalent name can be constructed, either because there is no meaningful equivalence between *name* and *leading_name*, or the system does not support constructing the requested equivalent name, for implementation-specific reasons. |

**EXAMPLES**

**EXAMPLE 1** Naming Files

In the Initial Context supporting XFN enterprise policies, a user `jsmith` is able to name one of her files relative to this context in several ways.

```
_myself/_fs/map.ps
_user/jsmith/_fs/map.ps
_orgunit/finance/_user/jsmith/_fs/map.ps
```

The first of these may be appealing to the user `jsmith` in her day-to-day operations. This name is not, however, appropriate for her to use when referring the file in an electronic mail message sent to a colleague. The second of these names would be appropriate if the colleague were in the same organizational unit, and the third appropriate for anyone in the same enterprise.

When the following sequence of instructions is executed by the user `jsmith` in the organizational unit `finance`, `enterprise_wide_name` would contain the composite name `_orgunit/finance/_user/jsmith/_fs/map.ps`:

```
FN_string_t* namestr =
    fn_string_from_str((const unsigned char*)"_myself/_fs/map.ps");
FN_composite_name_t* name = fn_composite_name_from_string(namestr);
FN_string_t* org_lead =
    fn_string_from_str((const unsigned char*)"_orgunit");
FN_status_t* status = fn_status_create();
FN_composite_name_t* enterprise_wide_name;
FN_ctx_t* init_ctx = fn_ctx_handle_from_initial(status);
/* check status of from_initial( ) */
enterprise_wide_name = fn_ctx_equivalent_name(init_ctx, name, org_lead,
status);
```

When the following sequence of instructions is executed by the user `jsmith` in the organizational unit `finance`, `shortest_name` would contain the composite name `_myself/_fs/map.ps`:

```
FN_string_t* namestr =
    fn_string_from_str((const unsigned char*)
        "_orgunit/finance/_user_jsmith/_fs/map.ps");
FN_composite_name_t* name = fn_composite_name_from_string(namestr);
FN_string_t* mylead = fn_string_from_str((const unsigned char*)"_myself");
FN_status_t* status = fn_status_create();
FN_composite_name_t* shortest_name;
FN_ctx_t* init_ctx = fn_ctx_handle_from_initial(status);
/* check status of from_initial( ) */
shortest_name = fn_ctx_equivalent_name(init_ctx, name, mylead, status);
```

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`FN_composite_name_t`(3XFN), `FN_ctx_t`(3XFN), `FN_status_t`(3XFN), `FN_string_t`(3XFN), `xfn_status_codes`(3XFN), `attributes`(5)

**NAME**

fn_ctx_get_ref – return a context's reference

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>

FN_ref_t ***`fn_ctx_get_ref`**(const FN_ctx_t **ctx*, FN_status_t **status*);

**DESCRIPTION**

This operation returns a reference to the supplied context object.

**RETURN VALUE**

`fn_ctx_get_ref( )` returns a pointer to an `FN_ref_t` object if the operation succeeds, it returns `0` if the operation fails.

**ERRORS**

`fn_ctx_get_ref( )` sets *status* as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN). The following status code is of particular relevance to this operation:

| | |
|---|---|
| `FN_E_OPERATION_NOT_SUPPORTED` | Using the `fn_ctx_get_ref( )` operation on the Initial Context returns this status code. |

**APPLICATION USAGE**

`fn_ctx_get_ref( )` cannot be used on the Initial Context. `fn_ctx_get_ref( )` can be used on contexts bound in the Initial Context (in other words, the bindings in the Initial Context have references).

If the context handle was created earlier using the `fn_ctx_handle_from_ref( )` operation, the reference returned by the `fn_ctx_get_ref( )` operation may not necessarily be exactly the same in content as that originally supplied. For example, `fn_ctx_handle_from_ref( )` may construct the context handle from one address from the list of addresses. The context implementation may return with a call to `fn_ctx_get_ref( )` only that address, or a more complete list of addresses than what was supplied in `fn_ctx_handle_from_ref( )`.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe. |

**SEE ALSO**

`FN_ctx_t`(3XFN), `FN_ref_t`(3XFN), `FN_status_t`(3XFN), `fn_ctx_handle_from_initial`(3XFN), `fn_ctx_handle_from_ref`(3XFN), `xfn_status_codes` (3XFN), `xfn`(3XFN), `attributes`(5)

**NAME**

fn_ctx_get_syntax_attrs – return syntax attributes associated with named context

**SYNOPSIS**

cc [ *flag* ... ] *file* ... −lxfn [ *library* ... ]
#include <xfn/xfn.h>

FN_attrset_t ***fn_ctx_get_syntax_attrs**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, FN_status_t **status*);

**DESCRIPTION**

Each context has an associated set of syntax-related attributes. This operation returns the syntax attributes associated with the context named by *name* relative to the context *ctx*.

The attributes must contain the attribute `fn_syntax_type` ( `FN_ID_STRING` format). If the context supports a syntax that conforms to the XFN standard syntax model, `fn_syntax_type` is set to "standard" (ASCII attribute syntax) and the attribute set contains the rest of the relevant syntax attributes described in `xfn_compound_names`(3XFN).

This operation is different from other XFN attribute operations in that these syntax attributes could be obtained directly from the context. Attributes obtained through other XFN attribute operations may not necessarily be associated with the context; they may be associated with the reference of context, rather than the context itself (see `xfn_attributes`(3XFN)).

**RETURN VALUE**

`fn_ctx_get_syntax_attrs( )` returns an attribute set if successful; it returns a `NULL` pointer (`0`) if the operation fails.

**ERRORS**

`fn_ctx_get_syntax_attrs( )` sets *status* as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN).

**APPLICATION USAGE**

Implementations may choose to support other syntax types in addition to, or in place of, the XFN standard syntax model, in which case, the value of the `fn_syntax_type` attribute would be set to an implementation-specific string, and different or additional syntax attributes will be in the set.

Syntax attributes of a context may be generated automatically by a context, in response to `fn_ctx_get_syntax_attrs( ),` or they may be created and updated using the base attribute operations. This is implementation-dependent.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe. |

**SEE ALSO**

`FN_attrset_t`(3XFN), `FN_composite_name_t`(3XFN), `FN_compound_name_t`(3XFN), `FN_ctx_t`(3XFN), `FN_status_t`(3XFN), `fn_attr_get`(3XFN), `fn_attr_multi_get`(3XFN),

`xfn_compound_names`(3XFN), `xfn_attributes`(3XFN), `xfn_status_codes`(3XFN), `xfn`(3XFN), `attributes`(5)

**NAME**

fn_ctx_handle_destroy – release storage associated with context handle

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>

void **`fn_ctx_handle_destroy`**(FN_ctx_t **ctx*);

**DESCRIPTION**

This operation destroys the context handle *ctx* and allows the implementation to free resources associated with the context handle. This operation does not affect the state of the context itself.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe. |

**SEE ALSO**

`FN_ctx_t`(3XFN), `fn_ctx_handle_from_initial`(3XFN), `fn_ctx_handle_from_ref`(3XFN), `xfn`(3XFN), `attributes`(5)

**NAME**

fn_ctx_handle_from_initial – return a handle to the Initial Context

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>

FN_ctx_t ***`fn_ctx_handle_from_initial`**(unsigned int *authoritative*, FN_status_t **status*);

**DESCRIPTION**

This operation returns a handle to the caller's Initial Context. On successful return, the handle points to a context which meets the specification of the XFN Initial Context (see `fns_initial_context`(5)).

*authoritative* specifies whether the handle to the context returned should be authoritative with respect to information the context obtains from the naming service. When the flag is non-zero, subsequent operations on the context will access the most authoritative information. When *authoritative* is `0`, the handle to the context returned need not be authoritative.

**RETURN VALUES**

`fn_ctx_handle_from_initial()` returns a pointer to an `FN_ctx_t` object if the operation succeeds; it returns a `NULL` pointer (`0`) otherwise.

**ERRORS**

`fn_ctx_handle_from_initial()` sets only the status code portion of the status object *status*.

**USAGE**

Authoritativeness is determined by specific naming services. For example, in a naming service that supports replication using a master / slave model, the source of authoritative information would come from the master server. In some naming systems, bypassing the naming service cache may reach servers which provide the most authoritative information. The availability of an authoritative context might be lower due to the lower number of servers offering this service. For the same reason, it might also provide poorer performance than contexts that need not be authoritative.

Applications set *authoritative* to `0` for typical day-to-day operations. Applications only set *authoritative* to a non-zero value when they require access to the most authoritative information, possibly at the expense of lower availability and / or poorer performance.

It is implementation-dependent whether authoritativeness is transferred from one context to the next as composite name resolution proceeds. Getting an authoritative context handle to the Initial Context means that operations on bindings in the Initial Context are processed using the most authoritative information. Contexts referenced implicitly through an authoritative Initial Context (for example, through the use of composite names) may not necessarily themselves be authoritative.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_ctx_t`(3XFN), `FN_status_t`(3XFN), `fn_ctx_get_ref`(3XFN), `fn_ctx_handle_from_ref`(3XFN), `xfn`(3XFN), `xfn_status_codes`(3XFN), `attributes`(5), `fns_initial_context`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

fn_ctx_handle_from_ref – construct a handle to a context object using the given reference

**SYNOPSIS**

`cc` [ *flag* … ] *file* … `–lxfn` [ *library* … ]
#include <xfn/xfn.h>

FN_ctx_t ***`fn_ctx_handle_from_ref`**(const FN_ref_t **ref*, unsigned int *authoritative*, FN_status_t **status*);

**DESCRIPTION**

This operation creates a handle to an `FN_ctx_t` object using an `FN_ref_t` object for that context.

*authoritative* specifies whether the handle to the context returned should be authoritative with respect to information the context obtains from the naming service. When the flag is non-zero, subsequent operations on the context will access the most authoritative information. When *authoritative* is `0`, the handle to the context returned need not be authoritative.

**RETURN VALUES**

This operation returns a pointer to an `FN_ctx_t` object if the operation succeeds; otherwise, it returns a `NULL` pointer (`0`).

**ERRORS**

`fn_ctx_handle_from_ref( )` sets *status* as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN). The following status code is of particular relevance to this operation:

| | |
|---|---|
| `FN_E_NO_SUPPORTED_ADDRESS` | A context object could not be constructed from a particular reference. The reference contained no address type over which the context interface was supported. |

**USAGE**

Authoritativeness is determined by specific naming services. For example, in a naming service that supports replication using a master / slave model, the source of authoritative information would come from the master server. In some naming systems, bypassing the naming service cache may reach servers which provide the most authoritative information. The availability of an authoritative context might be lower due to the lower number of servers offering this service. For the same reason, it might also provide poorer performance than contexts that need not be authoritative.

Applications set *authoritative* to `0` for typical day-to-day operations. Applications only set *authoritative* to a non-zero value when they require access to the most authoritative information, possibly at the expense of lower availability and / or poorer performance.

To control the authoritativeness of the target context, the application first resolves explicitly to the target context using `fn_ctx_lookup`(3XFN). It then uses `fn_ctx_handle_from_ref( )` with the appropriate authoritative argument to obtain a handle to the context. This returns a handle to a context with the specified authoritativeness. The application then uses the XFN operations, such as lookup and list, with this context handle.

It is implementation-dependent whether authoritativeness is transferred from one context to the next as composite name resolution proceeds. The application should use the approach recommended above to achieve the desired level of authoritativeness on a per context basis.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`FN_ctx_t`(3XFN), `FN_ref_t`(3XFN), `FN_status_t`(3XFN), `fn_ctx_get_ref`(3XFN), `fn_ctx_handle_destroy`(3XFN), `fn_ctx_lookup`(3XFN), `xfn`(3XFN), `xfn_status_codes`(3XFN), `attributes`(5), `fns_references`(5)

## NOTES

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

fn_ctx_list_bindings, FN_bindinglist_t, fn_bindinglist_next, fn_bindinglist_destroy – list the atomic names and references bound in a context

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>
FN_bindinglist_t ***fn_ctx_list_bindings**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, FN_status_t **status*);

FN_string_t ***fn_bindinglist_next**(FN_bindinglist_t **bl*, FN_ref_t ***ref*, FN_status_t **status*);

void **fn_bindinglist_destroy**(FN_bindinglist_t **bl*, FN_status_t **status*);

**DESCRIPTION**

This set of operations is used to list the names and bindings in the context named by *name* relative to the context *ctx* . Note that *name* must name a context. If the intent is to list the contents of *ctx* , *name* should be an empty composite name.

The semantics of these operations are similar to those for listing names (see `fn_ctx_list_names`(3XFN) ). In addition to a name string being returned, `fn_bindinglist_next()` also returns the reference of the binding for each member of the enumeration.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_composite_name_t`(3XFN) , `FN_ctx_t`(3XFN) , `FN_ref_t`(3XFN) , `FN_status_t`(3XFN) , `FN_string_t`(3XFN) , `fn_ctx_list_names`(3XFN) , `xfn`(3XFN) , `xfn_status_codes`(3XFN) , `attributes`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

## NAME

fn_ctx_list_names, FN_namelist_t, fn_namelist_next, fn_namelist_destroy – list the atomic names bound in a context

## SYNOPSIS

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>
FN_namelist_t ***fn_ctx_list_names**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, FN_status_t **status*);

FN_string_t ***fn_namelist_next**(FN_namelist_t **nl*, FN_status_t **status*);

void **fn_namelist_destroy**(FN_namelist_t **nl*, FN_status_t **status*);

## DESCRIPTION

This set of operations is used to list the names bound in the target context named *name* relative to the context *ctx* . Note that *name* must name a context. If the intent is to list the contents of *ctx* , *name* should be an empty composite name.

The call to `fn_ctx_list_names( )` initiates the enumeration process. It returns a handle to an `FN_namelist_t` object that can be used to enumerate the names in the target context.

The operation `fn_namelist_next( )` returns the next name in the enumeration identified by `nl` and updates `nl` to indicate the state of the enumeration. Successive calls to `fn_namelist_next( )` using `nl` return successive names in the enumeration and further update the state of the enumeration. `fn_namelist_next( )` returns a `NULL` pointer (`0` ) when the enumeration has been completed.

`fn_namelist_destroy( )` is used to release resources used during the enumeration. This may be invoked at any time to terminate the enumeration.

## RETURN VALUES

`fn_ctx_list_names( )` returns a pointer to an `FN_namelist_t` object if the enumeration is successfully initiated; otherwise it returns a `NULL` pointer (`0` ).

`fn_namelist_next( )` returns a `NULL` pointer (`0` ) if no more names can be returned in the enumeration.

In the case of a failure, these operations return in *status* a code indicating the nature of the failure.

## ERRORS

Each successful call to `fn_namelist_next( )` returns a name and sets *status* to `FN_SUCCESS.`

When `fn_namelist_next( )` returns a `NULL` pointer (`0` ), it indicates that no more names can be returned. *status* is set in the following way:

| | |
|---|---|
| `FN_SUCCESS` | The enumeration has completed successfully. |
| `FN_E_INVALID_ENUM_HANDLE` | The supplied enumeration handle is not valid. Possible reasons could be that the handle was from another |

| | |
|---|---|
| | enumeration, or the context being enumerated no longer accepts the handle (due to such events as handle expiration or updates to the context). |
| `FN_E_PARTIAL_RESULT` | The enumeration is not yet complete but cannot be continued. |

Other status codes, such as `FN_E_COMMUNICATION_FAILURE,` are also possible in calls to `fn_ctx_list_names()` , `fn_namelist_next()` , and `fn_namelist_destroy()` . These functions set *status* for these other status codes as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN) .

**USAGE**

The names enumerated using `fn_namelist_next()` are not ordered in any way. There is no guaranteed relation between the order in which names are added to a context and the order of names obtained by enumeration. The specification does *not* guarantee that any two series of enumerations will return the names in the same order.

When a name is added to or removed from a context, this may or may not invalidate the enumeration handle that the client holds for that context. If the enumeration handle becomes invalid, the status code `FN_E_INVALID_ENUM_HANDLE` is returned in *status* . If the enumeration handle remains valid, the update may or may not be visible to the client.

In addition, there may be a relationship between the *ctx* argument supplied to `fn_ctx_list_names()` and the `FN_namelist_t` object it returns. For example, some implementations may store the context handle *ctx* within the `FN_namelist_t` object for subsequent `fn_namelist_next()` calls. In general, a `fn_ctx_handle_destroy`(3XFN) should not be invoked on *ctx* until the enumeration has terminated.

**EXAMPLES**

**EXAMPLE 1** A sample program.

The following code fragment illustrates how the list names operations may be used:

```
extern FN_string_t *user_input;
FN_ctx_t *ctx;
FN_composite_name_t *target_name = fn_composite_name_from_string(user_input);
FN_status_t *status = fn_status_create();
FN_string_t *name;
FN_namelist_t *nl;
ctx = fn_ctx_handle_from_initial(status);
/* error checking on ’status’ */
if ((nl=fn_ctx_list_names(ctx, target_name, status)) == 0) {
 /* report ’status’ and exit */
}
while (name=fn_namelist_next(nl, status)) {
 /* do something with ’name’ */
 fn_string_destroy(name);
}
/* check 'status' for reason for end of enumeration and report if necessary */
/* clean up */
fn_namelist_destroy(nl, status);
/* report 'status' */
```

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`FN_composite_name_t`(3XFN) , `FN_ctx_t`(3XFN) , `FN_status_t`(3XFN) , `FN_string_t`(3XFN) , `fn_ctx_handle_destroy`(3XFN) , `xfn`(3XFN) , `xfn_status_codes`(3XFN) , `attributes`(5)

## NOTES

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

fn_ctx_lookup – look up name in context

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `-lxfn` [ *library* ... ]
#include <xfn/xfn.h>

FN_ref_t ***fn_ctx_lookup**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, FN_status_t **status*);

**DESCRIPTION**

This operation returns the reference bound to *name* relative to the context *ctx*.

**RETURN VALUE**

If the operation succeeds, the `fn_ctx_lookup()` function returns a handle to the reference bound to *name*. Otherwise, `0` is returned and *status* is set appropriately.

**ERRORS**

`fn_ctx_lookup()` sets *status* as described `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN).

**APPLICATION USAGE**

Some naming services may not always have reference information for all names in their contexts; for such names, such naming services may return a special reference whose type indicates that the name is not bound to any address. This reference may be name service specific or it may be the conventional NULL reference defined in the X/Open registry. See `fns_references`(5).

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe. |

**SEE ALSO**

`FN_composite_name_t`(3XFN), `FN_ctx_t`(3XFN), `FN_ref_t`(3XFN), `FN_status_t`(3XFN), `fns_references`(5), `xfn_status_codes` (3XFN), `xfn`(3XFN), `attributes`(5)

**NAME**

fn_ctx_lookup_link – look up the link reference bound to a name

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>

FN_ref_t ***fn_ctx_lookup_link**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, FN_status_t **status*);

**DESCRIPTION**

This operation returns the XFN link bound to *name*. The terminal atomic part of *name* must be bound to an XFN link.

The normal `fn_ctx_lookup`(3XFN) operation follows all links encountered, including any bound to the terminal atomic part of *name*. This operation differs from the normal lookup in that when the terminal atomic part of *name* is an XFN link, this link is not followed, and the operation returns the link.

**RETURN VALUES**

If `fn_ctx_lookup_link( )` fails, a `NULL` pointer (`0`) is returned.

**ERRORS**

`fn_ctx_lookup_link( )` sets *status* as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN). Of special relevance for `fn_ctx_lookup_link( )` is the following status code:

| | |
|---|---|
| `FN_E_MALFORMED_LINK` | *name* resolved to a reference that was not a link. |

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_composite_name_t`(3XFN), `FN_ctx_t`(3XFN), `FN_ref_t`(3XFN), `FN_status_t`(3XFN), `fn_ctx_lookup`(3XFN), `xfn`(3XFN), `xfn_links`(3XFN), `xfn_status_codes`(3XFN), `attributes`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

fn_ctx_rename – rename the name of a binding

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>

int **`fn_ctx_rename`**(FN_ctx_t **ctx*, const FN_composite_name_t **oldname*, const FN_composite_name_t **newname*, unsigned int *exclusive*, FN_status_t **status*);

**DESCRIPTION**

The `fn_ctx_rename( )` operation binds the reference currently bound to *oldname* relative to *ctx*, to the name *newname*, and unbinds *oldname*. *newname* is resolved relative to the target context (that named by all but the terminal atomic part of *oldname*).

If *exclusive* is `0`, the operation overwrites any old binding of *newname*. If *exclusive* is nonzero, the operation fails if *newname* is already bound.

**RETURN VALUES**

`fn_ctx_rename( )` returns `1` if the operation is successful, `0` otherwise.

**ERRORS**

`fn_ctx_rename( )` sets *status* as described `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN).

**USAGE**

The only restriction that XFN places on *newname* is that it be resolved relative to the target context. XFN does not specify further restrictions on *newname*. For example, in some implementations, *newname* might be restricted to be a name in the same naming system as the terminal component of *oldname*. In another implementation, *newname* might be restricted to be an atomic name.

Normal resolution always follows links. In an `fn_ctx_rename( )` operation, resolution of *oldname* continues to the target context; the terminal atomic name is not resolved. If the terminal atomic name is bound to a link, the link is not followed and the operation binds *newname* to the link and unbinds the terminal atomic name of *oldname*.

In naming systems that support attributes and store the attributes along with the names, the unbind of the terminal atomic name of *oldname* also removes its associated attributes. It is implementation-dependent whether these attributes become associated with *newname*.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_composite_name_t`(3XFN), `FN_ctx_t`(3XFN), `FN_ref_t`(3XFN), `FN_status_t`(3XFN), `fn_ctx_bind`(3XFN) `fn_ctx_unbind`(3XFN), `xfn`(3XFN), `xfn_status_codes`(3XFN), `attributes`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

FN_ctx_t – an XFN context

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `-lxfn` [ *library* ... ]
#include <xfn/xfn.h>

FN_ctx_t ***fn_ctx_handle_from_initial**(unsigned int *authoritative*, FN_status_t **status*);

FN_ctx_t ***fn_ctx_handle_from_ref**(const FN_ref_t **ref*, unsigned int *authoritative*, FN_status_t **status*);

FN_ref_t ***fn_ctx_get_ref**(const FN_ctx_t **ctx*, FN_status_t **status*);

void **fn_ctx_handle_destroy**(FN_ctx_t **ctx*);

FN_ref_t ***fn_ctx_lookup**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, FN_status_t **status*);

FN_namelist_t ***fn_ctx_list_names**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, FN_status_t **status*);

FN_string_t ***fn_namelist_next**(FN_namelist_t **nl*, FN_status_t **status*);

void **fn_namelist_destroy**(FN_namelist_t **nl*, FN_status_t **status*);

FN_bindinglist_t ***fn_ctx_list_bindings**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, FN_status_t **status*);

FN_string_t ***fn_bindinglist_next**(FN_bindinglist_t **iter*, FN_ref_t ***ref*, FN_status_t **status*);

void **fn_bindinglist_destroy**(FN_bindinglist_t **iter_pos*, FN_status_t **status*);

int **fn_ctx_bind**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, const FN_ref_t **ref*, unsigned int *exclusive*, FN_status_t **status*);

int **fn_ctx_unbind**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, FN_status_t **status*);

int **fn_ctx_rename**(FN_ctx_t **ctx*, const FN_composite_name_t **oldname*, const FN_composite_name_t **newname*, unsigned int *exclusive*, FN_status_t **status*);

FN_ref_t ***fn_ctx_create_subcontext**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, FN_status_t **status*);

int **fn_ctx_destroy_subcontext**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, FN_status_t **status*);

FN_ref_t ***fn_ctx_lookup_link**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, FN_status_t **status*);

FN_attrset_t ***fn_ctx_get_syntax_attrs**(FN_ctx_t **ctx*, const FN_composite_name_t **name*, FN_status_t **status*);

## DESCRIPTION

An XFN context consists of a set of name to reference bindings. An XFN context is represented by the type `FN_ctx_t` in the client interface. The operations for manipulating an `FN_ctx_t` object are described in detail in separate reference manual pages.

The following contains a brief summary of these operations:

`fn_ctx_handle_from_initial()` returns a pointer to an Initial Context that provides a starting point for resolution of composite names. `fn_ctx_handle_from_ref()` returns a handle to an `FN_ctx_t` object using the given reference *ref*. `fn_ctx_get_ref()` returns the reference of the context *ctx*. `fn_ctx_handle_destroy()` releases the resources associated with the `FN_ctx_t` object *ctx*; it does not affect the state of the context itself.

`fn_ctx_lookup()` returns the reference bound to *name* resolved relative to *ctx*. `fn_ctx_list_names()` is used to enumerate the atomic names bound in the context named by *name* resolved relative to *ctx*. `fn_ctx_list_bindings()` is used to enumerate the atomic names and their references in the context named by *name* resolved relative to *ctx*.

`fn_ctx_bind()` binds the composite name *name* to a reference *ref* resolved relative to *ctx*. `fn_ctx_unbind()` unbinds *name* resolved relative to *ctx*. `fn_ctx_rename()` binds *newname* to the reference bound to *oldname* and unbinds *oldname*. *oldname* is resolved relative to *ctx*; *newname* is resolved relative to the target context.

`fn_ctx_create_subcontext()` creates a new context with the given composite name *name* resolved relative to *ctx*. `fn_ctx_destroy_subcontext()` destroys the context named by *name* resolved relative to *ctx*.

Normal resolution always follows links. `fn_ctx_lookup_link()` looks up *name* relative to *ctx*, following links except for the last atomic part of *name*, which must be bound to an XFN link.

`fn_ctx_get_syntax_attrs()` returns an attribute set containing attributes that describe a context's syntax. *name* must name a context.

## ERRORS

In each context operation, the caller supplies an `FN_status_t` object as a parameter. The called function sets this status object as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN).

## USAGE

In most of the operations of the base context interface, the caller supplies a context and a composite name. The supplied name is always interpreted relative to the supplied context.

The operation may eventually be effected on a different context called the operation's *target context*. Each operation has an initial resolution phase that conveys the operation to its target context, and the operation is then applied. The effect (but not necessarily the implementation) is that of doing a lookup on that portion of the name that represents the target context, and then invoking the operation on the target context. The contexts involved only in the resolution phase are called *intermediate contexts*.

Normal resolution of names in context operations always follows XFN links.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`FN_attrset_t`(3XFN), `FN_composite_name_t`(3XFN), `FN_ref_t`(3XFN), `FN_status_t`(3XFN), `fn_ctx_bind`(3XFN), `fn_ctx_create_subcontext`(3XFN), `fn_ctx_destroy_subcontext`(3XFN), `fn_ctx_get_ref`(3XFN), `fn_ctx_get_syntax_attrs`(3XFN), `fn_ctx_handle_destroy`(3XFN), `fn_ctx_handle_from_initial`(3XFN), `fn_ctx_handle_from_ref`(3XFN), `fn_ctx_list_bindings`(3XFN), `fn_ctx_list_names`(3XFN), `fn_ctx_lookup`(3XFN), `fn_ctx_lookup_link`(3XFN), `fn_ctx_rename`(3XFN), `fn_ctx_unbind`(3XFN), `xfn`(3XFN), `xfn_links`(3XFN), `xfn_status_codes`(3XFN), `attributes`(5)

## NOTES

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

fn_ctx_unbind – unbind a name from a context

**SYNOPSIS**

```
cc [ flag ... ] file ... -lxfn [ library ... ]
#include <xfn/xfn.h>


int fn_ctx_unbind(FN_ctx_t *ctx, const FN_composite_name_t *name, FN_status_t
*status);
```

**DESCRIPTION**

This operation removes the terminal atomic name in *name* from the the target context — that named by all but the terminal atomic part of *name*.

This operation is successful even if the terminal atomic name was not bound in target context, but fails if any of the intermediate names are not bound. `fn_ctx_unbind( )` is idempotent.

**RETURN VALUE**

The operation returns `1` if successful, and `0` otherwise.

**ERRORS**

`fn_ctx_unbind( )` sets *status* as described in `FN_status_t` and `xfn_status_codes` (3XFN).

Certain naming systems may disallow unbinding a name if the name is bound to an existing context in order to avoid orphan contexts that cannot be reached via any name. In such situations, the status code `FN_E_OPERATION_NOT_SUPPORTED` is returned.

**APPLICATION USAGE**

In naming systems that support attributes, and store the attributes along with the names, the unbind operation removes the name and its associated attributes.

Normal resolution always follows links. In an `fn_ctx_unbind( )` operation, resolution of *name* continues to the target context; the terminal atomic name is not resolved. If the terminal atomic name is bound to a link, the link is not followed and the link itself is unbound from the terminal atomic name.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe. |

**SEE ALSO**

`FN_composite_name_t`(3XFN), `FN_ctx_t`(3XFN), `FN_ref_t`(3XFN), `FN_status_t`(3XFN), `fn_ctx_bind`(3XFN), `fn_ctx_lookup`(3XFN), `xfn_status_codes`(3XFN), `xfn`(3XFN), `attributes`(5)

**NAME**

FN_identifier_t – an XFN identifier

**DESCRIPTION**

Identifiers are used to identify reference types and address types in an XFN reference, and to identify attributes and their syntax in the attribute operations.

An XFN identifier consists of an `unsigned int`, which determines the format of identifier, and the actual identifier, which is expressed as a sequence of octets.

The representation of this structure is defined by XFN as follows:

```
typedef struct {
unsigned int format;
size_t length;
void *contents;
} FN_identifier_t;
```

XFN defines a small number of standard forms for identifiers:

| | |
|---|---|
| `FN_ID_STRING` | The identifier is an ASCII string (ISO 646). |
| `FN_ID_DCE_UUID` | The identifier is an OSF DCE UUID in string representation. (See the X/Open DCE RPC.) |
| `FN_ID_ISO_OID_STRING` | The identifier is an ISO OID in ASN.1 dot-separated integer list string format. (See the ISO ASN.1.) |
| `FN_ID_ISO_OID_BER` | The identifier is an ISO OID in ASN.1 Basic Encoding Rules (BER) format. (See the ISO BER.) |

**FILES**

`#include <xfn/xfn.h>`

**SEE ALSO**

`FN_attribute_t`(3XFN), `FN_ref_addr_t`(3XFN), `FN_ref_t`(3XFN), `xfn`(3XFN)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

## NAME

FN_ref_addr_t, fn_ref_addr_create, fn_ref_addr_destroy, fn_ref_addr_copy, fn_ref_addr_assign, fn_ref_addr_type, fn_ref_addr_length, fn_ref_addr_data, fn_ref_addr_description – an address in an XFN reference

## SYNOPSIS

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>
FN_ref_addr_t ***fn_ref_addr_create**(constFN_identifier_t **type*, size_t *length*, const void **data*);

void **fn_ref_addr_destroy**(FN_ref_addr_t **addr*);

FN_ref_addr_t ***fn_ref_addr_copy**(constFN_ref_addr_t **addr*);

FN_ref_addr_t ***fn_ref_addr_assign**(FN_ref_addr_t **dst*, const FN_ref_addr_t **src*);

const FN_identifier_t ***fn_ref_addr_type**(constFN_ref_addr_t **addr*);

size_t **fn_ref_addr_length**(const FN_ref_addr_t **addr*);

const void* **fn_ref_addr_data**(const FN_ref_addr_t **addr*);

FN_string_t ***fn_ref_addr_description**(constFN_ref_addr_t **addr*, unsigned int *detail*, unsigned int **more_detail*);

## DESCRIPTION

An XFN reference is represented by the type `FN_ref_t` . An object of this type contains a reference type and a list of addresses. Each address in the list is represented by an object of type `FN_ref_addr_t` . An address consists of an opaque data buffer and a type field, of type `FN_identifier_t` .

`fn_ref_addr_create( )` creates and returns an address with the given type and data. *length* indicates the size of the data. `fn_ref_addr_destroy( )` releases the storage associated with the given address. `fn_ref_addr_copy( )` returns a copy of the given address object. `fn_ref_addr_assign( )` makes a copy of the address pointed to by *src* and assigns it to *dst* , releasing any old contents of *dst* . A pointer to the same object as *dst* is returned.

`fn_ref_addr_type( )` returns the type of the given address. `fn_ref_addr_length( )` returns the size of the address in bytes. `fn_ref_addr_data( )` returns the contents of the address.

`fn_ref_addr_description( )` returns the implementation-defined textual description of the address. It takes as arguments a number, *detail* , and a pointer to a number, *more_detail* . *detail* specifies the level of detail for which the description should be generated; the higher the number, the more detail is to be provided. If *more_detail* is `0` , it is ignored. If *more_detail* is non-zero, it is set by the description operation to indicate the next level of detail available, beyond that specified by *detail* . If no higher level of detail is available, *more_detail* is set to *detail* .

**USAGE**

The address type of an `FN_ref_addr_t` object is intended to identify the mechanism that should be used to reach the object using that address. The client must interpret the contents of the opaque data buffer of the address based on the type of the address, and on the type of the reference that the address is in. However, this interpretation is intended to occur below the application layer. Most applications developers should not have to manipulate the contents of either address or reference objects themselves. These interfaces would generally be used within service libraries.

Multiple addresses in a single reference are intended to identify multiple communication endpoints for the same conceptual object. Multiple addresses may arise for various reasons, such as the object offering interfaces over more than one communication mechanism.

Manipulation of addresses using the operations described in this manual page does not affect their representation in the underlying naming system. Changes to addresses in the underlying naming system can only be effected through the use of the interfaces described in `FN_ctx_t`(3XFN) .

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_ctx_t`(3XFN) , `FN_identifier_t`(3XFN) , `FN_ref_t`(3XFN) , `FN_string_t`(3XFN) , `xfn`(3XFN) , `attributes`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

FN_ref_t, fn_ref_create, fn_ref_destroy, fn_ref_copy, fn_ref_assign, fn_ref_type, fn_ref_addrcount, fn_ref_first, fn_ref_next, fn_ref_prepend_addr, fn_ref_append_addr, fn_ref_insert_addr, fn_ref_delete_addr, fn_ref_delete_all, fn_ref_create_link, fn_ref_is_link, fn_ref_link_name, fn_ref_description – an XFN reference

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `−lxfn` [ *library* ... ]
#include <xfn/xfn.h>
FN_ref_t ***fn_ref_create**(const FN_identifier_t **ref_type*);

void **fn_ref_destroy**(FN_ref_t **ref*);

FN_ref_t ***fn_ref_copy**(const FN_ref_t **ref*);

FN_ref_t ***fn_ref_assign**(FN_ref_t **dst*, const FN_ref_t **src*);

const FN_identifier_t ***fn_ref_type**(const FN_ref_t **ref*);

unsigned int **fn_ref_addrcount**(const FN_ref_t **ref*);

const FN_ref_addr_t ***fn_ref_first**(const FN_ref_t **ref*, void ***iter_pos*);

const FN_ref_addr_t ***fn_ref_next**(const FN_ref_t **ref*, void ***iter_pos*);

int **fn_ref_prepend_addr**(FN_ref_t **ref*, const FN_ref_addr_t **addr*);

int **fn_ref_append_addr**(FN_ref_t **ref*, const FN_ref_addr_t **addr*);

int **fn_ref_insert_addr**(FN_ref_t **ref*, void ***iter_pos*, const FN_ref_addr_t **addr*);

int **fn_ref_delete_addr**(FN_ref_t **ref*, void ***iter_pos*);

int **fn_ref_delete_all**(FN_ref_t **ref*);

FN_ref_t ***fn_ref_create_link**(const FN_composite_name_t **link_name*);

int **fn_ref_is_link**(const FN_ref_t **ref*);

FN_composite_name_t ***fn_ref_link_name**(const FN_ref_t **link_ref*);

FN_string_t ***fn_ref_description**(const FN_ref_t **ref*, unsigned int *detail*, unsigned int **more_detail*);

**DESCRIPTION**

An XFN reference is represented by the type `FN_ref_t` . An object of this type contains a reference type and a list of addresses. The ordering in this list at the time of binding might not be preserved when the reference is returned upon lookup.

The reference type is represented by an object of type `FN_identifier_t` . The reference type is intended to identify the class of object referenced. XFN does not dictate the precise use of this.

Each address is represented by an object of type `FN_ref_addr_t` .

`fn_ref_create( )` creates a reference with no address, using *ref_type* as its reference type. Addresses can be added later to the reference using the functions described below. `fn_ref_destroy( )` releases the storage associated with *ref* . `fn_ref_copy( )` creates a copy of *ref* and returns it. `fn_ref_assign( )` creates a copy of *src* and assigns it to *dst* , releasing any old contents of *dst* . A pointer to the same object as *dst* is returned.

`fn_ref_addrcount( )` returns the number of addresses in the reference *ref* .

`fn_ref_first( )` returns the first address in *ref* and sets *iter_pos* to be after the address. It returns `0` if there is no address in the list. `fn_ref_next( )` returns the address following *iter_pos* in *ref* and sets *iter_pos* to be after the address. If the iteration marker *iter_pos* is at the end of the sequence, `fn_ref_next( )` returns `0` .

`fn_ref_prepend_addr( )` adds *addr* to the front of the list of addresses in *ref* . `fn_ref_append_addr( )` adds *addr* to the end of the list of addresses in *ref* . `fn_ref_insert_addr( )` adds *addr* to *ref* before *iter_pos* and sets *iter_pos* to be immediately after the new reference added. `fn_ref_delete_addr( )` deletes the address located before *iter_pos* in the list of addresses in *ref* and sets *iter_pos* back one address. `fn_ref_delete_all ( )` deletes all addresses in *ref* .

`fn_ref_create_link( )` creates a reference using the given composite name *link_name* as an address. `fn_ref_is_link( )` tests if *ref* is a link. It returns `1` if it is; `0` if it is not. `fn_ref_link_name( )` returns the composite name stored in a link reference. It returns `0` if *link_ref* is not a link.

`fn_ref_description( )` returns a string description of the given reference. It takes as argument an integer, *detail* , and a pointer to an integer, *more_detail* . *detail* specifies the level of detail for which the description should be generated; the higher the number, the more detail is to be provided. If *more_detail* is `0` , it is ignored. If *more_detail* is non-zero, it is set by the description operation to indicate the next level of detail available, beyond that specified by *detail* . If no higher level of detail is available, *more_detail* is set to *detail* .

**RETURN VALUES**

The following operations return `1` if the operation succeeds, `0` if the operation fails:

```
fn_ref_prepend_addr( )
fn_ref_append_addr( )
fn_ref_insert_addr( )
fn_ref_delete_addr( )
fn_ref_delete_all( )
```

**USAGE**

The reference type is intended to identify the class of object referenced. XFN does not dictate the precise use of this.

Multiple addresses in a single reference are intended to identify multiple communication endpoints for the same conceptual object. Multiple addresses may arise for various reasons, such as the object offering interfaces over more than one communication mechanism.

The client must interpret the contents of a reference based on the type of the addresses and the type of the reference. However, this interpretation is intended to occur below the application layer. Most applications developers should not have to manipulate the contents of either address or reference objects themselves. These interfaces would generally be used within service libraries.

Manipulation of references using the operations described in this manual page does not affect their representation in the underlying naming system. Changes to references in the underlying naming system can only be effected through the use of the interfaces described in `FN_ctx_t`(3XFN) .

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_composite_name_t`(3XFN) , `FN_ctx_t`(3XFN) , `FN_identifier_t`(3XFN) , `FN_ref_addr_t`(3XFN) , `FN_string_t`(3XFN) , `fn_ctx_lookup`(3XFN) , `fn_ctx_lookup_link`(3XFN) , `xfn`(3XFN) , `xfn_links`(3XFN) , `attributes`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

## NAME

FN_search_control_t, fn_search_control_create, fn_search_control_destroy, fn_search_control_copy, fn_search_control_assign, fn_search_control_scope, fn_search_control_follow_links, fn_search_control_max_names, fn_search_control_return_ref, fn_search_control_return_attr_ids – options for attribute search

## SYNOPSIS

#include <xfn/xfn.h>
FN_search_control_t ***fn_search_control_create**(unsigned int *scope*, unsigned int *follow_links*, unsigned int *max_names*, unsigned int *return_ref*, const FN_attrset_t **return_attr_ids*, unsigned int **status*);

void **fn_search_control_destroy**(FN_search_control_t **scontrol*);

FN_search_control_t ***fn_search_control_copy**(const FN_search_control_t **scontrol*);

FN_search_control_t ***fn_search_control_assign**(FN_search_control_t **dst*, const FN_search_control_t **src*);

unsigned int **fn_search_control_scope**(const FN_search_control_t **scontrol*);

unsigned int **fn_search_control_follow_links**(const FN_search_control_t **scontrol*);

unsigned int **fn_search_control_max_names**(const FN_search_control_t **scontrol*);

unsigned int **fn_search_control_return_ref**(const FN_search_control_t **scontrol*);

const FN_attrset_t ***fn_search_control_return_attr_ids**(const FN_search_control_t **scontrol*);

## DESCRIPTION

The `FN_search_control_t` object is used to specify options for the attribute search operation `fn_attr_ext_search`(3XFN) .

`fn_search_control_create( )` creates an `FN_search_control_t` object using information in *scope* , *follow_links* , *max_names* , *return_ref* , and *return_attr_ids* to set the search options. If the operation succeeds, `fn_search_control_create( )` returns a pointer to an `FN_search_control_t` object; otherwise, it returns a `NULL` pointer.

The scope of the search, *scope* , is either the named object, the named context, the named context and its subcontexts, or the named context and a context implementation defined set of subcontexts. The values for *scope* are:

| | |
|---|---|
| `FN_SEARCH_NAMED_OBJECT` | Search just the given named object. |
| `FN_SEARCH_ONE_CONTEXT` | Search just the given context. |
| `FN_SEARCH_SUBTREE` | Search given context and all its subcontexts. |

| | |
|---|---|
| `FN_SEARCH_CONSTRAINED_SUBTREE` | Search given context and its subcontexts as constrained by the context-specific policy in place at the named context. |

*follow_links* further defines the scope and nature of the search. If *follow_links* is nonzero, the search follows XFN links. If *follow_links* is `0` , XFN links are not followed. See `fn_attr_ext_search`(3XFN) for more detail about how XFN links are treated.

*max_names* specifies the maximum number of names to return in an `FN_ext_searchlist_t`(3XFN) enumeration (see `fn_attr_ext_search`(3XFN) ). The names of all objects whose attributes satisfy the filter are returned when *max_names* is `0` .

If *return_ref* is non-zero, the reference bound to the named object is returned with the object's name by `fn_ext_searchlist_next`(3XFN) (see `fn_attr_ext_search`(3XFN) ). If *return_ref* is `0` , the reference is not returned.

Attribute identifiers and values associated with named objects that satisfy the filter may be returned by `fn_ext_searchlist_next`(3XFN) . The attributes returned are those listed in *return_attr_ids* . If the value of *return_attr_ids* is `0` , all attributes are returned. If *return_attr_ids* is an empty `FN_attrset_t` object (see `FN_attrset_t`(3XFN) ), no attributes are returned. Any attribute values in *return_attr_ids* are ignored; only the attribute identifiers are relevant for this operation.

`fn_attr_ext_search`(3XFN) interprets a value of `0` for the search control argument as a default search control which has the following option settings:

| | |
|---|---|
| *scope* | `FN_SEARCH_ONE_CONTEXT` |
| *follow_links* | `0` (do not follow links) |
| *max_names* | `0` (return all named objects that match filter) |
| *return_ref* | `0` (do not return the reference of the named object) |
| *return_attr_ids* | an empty `FN_attrset_t` object (do not return any attributes of the named object) |

`fn_search_control_destroy( )` releases the storage associated with *scontrol* .

`fn_search_control_copy( )` returns a copy of the search control *scontrol* .

`fn_search_control_assign( )` makes a copy of the search control *src* and assigns it to *dst* , releasing the old contents of *dst* . A pointer to the same object as *dst* is returned.

`fn_search_control_scope()` returns the scope for the search.

`fn_search_control_follow_links()` returns non-zero if links are followed; `0` if not.

`fn_search_control_max_names()` returns the maximum number of names.

`fn_search_control_return_ref()` returns nonzero if the reference is returned; `0` if not.

`fn_search_control_return_attr_ids()` returns a pointer to the list of attributes; a `NULL` pointer indicates that all attributes and values are returned.

**ERRORS**

`fn_search_control_create()` returns a `NULL` pointer if the operation fails and sets status as follows:

| | |
|---|---|
| `FN_E_SEARCH_INVALID_OPTION` | A supplied search option was invalid or inconsistent. |

Other status codes are possible (see `xfn_status_codes`(3XFN) ).

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_attrset_t`(3XFN) , `fn_attr_ext_search`(3XFN) , `xfn_status_codes`(3XFN) , `attributes`(5)

## NAME

FN_search_filter_t, fn_search_filter_create, fn_search_filter_destroy, fn_search_filter_copy, fn_search_filter_assign, fn_search_filter_expression, fn_search_filter_arguments – filter expression for attribute search

## SYNOPSIS

#include <xfn/xfn.h>
FN_search_filter_t ***fn_search_filter_create**(unsigned int **status*, const unsigned char **estr*, .);

void **fn_search_filter_destroy**(FN_search_filter_t **sfilter*);

FN_search_filter_t ***fn_search_filter_copy**(const FN_search_filter_t **sfilter*);

FN_search_filter_t ***fn_search_filter_assign**(FN_search_filter_t **dst*, const FN_search_filter_t **src*);

const char ***fn_search_filter_expression**(const FN_search_filter_t **sfilter*);

const void ****fn_search_filter_arguments**(const FN_search_filter_t **sfilter*, size_t **number_of_arguments*);

## DESCRIPTION

The `FN_search_filter_t` type is an expression that is evaluated against the attributes of named objects bound in the scope of the search operation `fn_attr_ext_search`(3XFN) . The filter evaluates to `TRUE` or `FALSE.` If the filter is empty, it evaluates to `TRUE.` Names of objects whose attribute values satisfy the filter expression are returned by the search operation.

If the identifier in any subexpression of the filter does not exist as an attribute of an object, then the innermost logical expression containing that identifier is `FALSE.` A subexpression that is only an attribute tests for the presence of the attribute; the subexpression evaluates to `TRUE` if the attribute has been defined for the object and `FALSE` otherwise.

`fn_search_filter_create( )` creates a search filter from the expression string *estr* and the remaining arguments.

`fn_search_filter_destroy( )` releases the storage associated with the search filter *sfilter* .

`fn_search_filter_copy( )` returns a copy of the search filter *sfilter* .

`fn_search_filter_assign( )` makes a copy of the search filter *src* and assigns it to *dst* , releasing the old contents of *dst* . A pointer to the same object as *dst* is returned.

`fn_search_filter_expression( )` returns the filter expression of *sfilter.*

`fn_search_filter_arguments( )` returns an array of pointers to arguments supplied to the filter constructor. *number_of_arguments* is set to the size of this array. The types of the arguments are determined by the substitution tokens in the expression in *sfilter* .

**BNF of Filter Expression**

```
<FilterExpr> ::= [ <Expr> ]
<Expr> ::= <Expr> "or" <Expr>
        <Expr> "and" <Expr>
        | "not" <Expr>
        | "(" <Expr> ")"
       | <Attribute> [ <Rel_Op> <Value> ]
        | <Ext>
<Rel_Op> ::= "==" | "!=" | "<" | "<=" | ">" | ">=" | "[ap    ]="
<Attribute> ::= "%a"
<Value> ::= <Integer>
        | "%v"
        |<Wildcarded_string>
<Wildcarded_string> ::= "*"
        | <String>
       | {<String> "*"}+ [<String>]
       | {"*" <String>}+ ["*"]
<String> ::= "'" { <Char> } * "'"
        | "%s"
<Char> ::= <PCS> // See BNF in Section 4.1.2 for PCSdefinition
        | Characters in the repertoire of a string representation
<Identifier> ::=" "%i"
<Ext> ::= <Ext_Op> "(" [Arg_List] ")"
<Ext_Op> ::= <String> | <Identifier>
<Arg_List> ::= <Arg> | <Arg> "," <Arg_List>
<Arg> ::= <Value> | <Attribute> | <Identifier>
```

**Specification of Filter Expression**

The arguments to `fn_search_filter_create()` are a return status, an expression string, and a list of arguments. The string contains the filter expression with substitution tokens for the attributes, attribute values, strings, and identifiers that are part of the expression. The remaining list of arguments contains the attributes and values in the order of appearance of their corresponding substitution tokens in the expression. The arguments are of types `FN_attribute_t*` , `FN_attrvalue_t*` , `FN_string_t*` , or `FN_identifier_t*` . Any attribute values in an `FN_attribute_t*` type of argument are ignored; only the attribute identifier and attribute syntax are relevant. The argument type expected by each substitution token are listed in the following table.

| Token | Argument Type |
|---|---|
| %a | FN_attribute_t* |
| %v | FN_attrvalue_t* |
| %s | FN_string_t* |
| %i | FN_identifier_t* |

**Precedence**

The following precedence relations hold in the absence of parentheses, in the order of lowest to highest:

or
and
not
relational operators

These boolean and relational operators are left associative.

**Relational Operators**

Comparisons and ordering are specific to the syntax and / or rules of the supplied attribute.

Locale (code set, language, or territory) mismatches that occur during string comparisons and ordering operations are resolved in an implementation-dependent way. Relational operations that have ordering semantics may be used for strings of code sets in which ordering is meaningful, but is not of general use in internationalized environments.

An attribute that occurs in the absence of any relational operator tests for the presence of the attribute.

| Operator | Meaning |
|---|---|
| `==` | The sub-expression is TRUE if at least one value of the specified attribute is equal to the supplied value. |
| `!=` | The sub-expression is TRUE if no values of the specified attribute equal the supplied value. |
| `>=` | The sub-expression is TRUE if at least one value of the attribute is greater than or equal to the supplied value. |
| `>` | The sub-expression is TRUE if at least one value of the attribute is greater then the supplied value. |
| `<=` | The sub-expression is TRUE if at least one value of the attribute is less than or equal to the supplied value. |
| `<` | The sub-expression is TRUE if at least one value of the attribute is less than the supplied value. |
| `[ap ]=` | The sub-expression is TRUE if at least one value of the specified attribute matches the supplied value according to some context-specific approximate matching criterion. This criterion must subsume strict equality. |

**Wildcarded Strings**

A wildcarded string consists of a sequence of alternating wildcard specifiers and strings. The sequence can start with either a wildcard specifier or a string, and end with either a wildcard specifier or a string.

The wildcard specifier is denoted by the asterisk character ('* ') and means zero or more occurrences of any character.

Wildcarded strings can be used to specify substring matches. The following are examples of wildcarded strings and what they mean:

| Wildcarded String | Meaning |
|---|---|
| * | Any string |
| *'ing' | Any string ending with ing |
| Any string starting | |
| with jo , and containing the substring ph , | |
| and which contains the substring ne in the portion | |
| of the string following ph , and which ends with er | |
| T} | |
| %s* | Any string starting with the supplied string |
| Any string starting with bix and ending with the supplied string | |
| T} | |

String matches involving strings of different locales (code set, language, or territory) are resolved in an implementation-dependent way.

**Extended Operations**

In addition to the relational operators, extended operators can be specified. All extended operators return either TRUE or FALSE . A filter expression can contain both relational and extended operations.

Extended operators are specified using an identifier (see FN_identifier_t(3XFN) ) or a string. If the operator is specified using a string, the string is used to construct an identifier of format FN_ID_STRING . Identifiers of extended operators and signatures of the corresponding extended operations, as well as their suggested semantics, are registered with X/Open Company Ltd.

The following three extended operations are currently defined:

| | |
|---|---|
| `'name'(<`*Wildcarded* `String>)` | The identifier for this operation is `'name' (FN_ID_STRING)` . The argument to this operation is a wildcard string. The operation returns `TRUE` if the name of the object matches the supplied wildcard string. |
| `'reftype'(%i)` | The identifier for this operation is `'reftype' (FN_ID_STRING)` . The argument to this operation is an identifier. The operation returns `TRUE` if the reference type of the object is equal to the supplied identifier. |
| `'addrtype'(%i)` | The identifier for this operation is `'addrtype' (LM FN_ID_STRING)` . The argument to the operation is an identifier. The operation returns `TRUE` if any of the address types in the reference of the object is equal to the supplied identifier. |

Support and exact semantics of extended operations are context-specific. If a context does not support an extended operation, or if the filter expression supplies the extended operation with either an incorrect number or type of arguments, the error `FN_E_SEARCH_INVALID_OP` is returned. (Note: `FN_E_OPERATION_NOT_SUPPORTED` is returned when `fn_attr_ext_search`(3XFN) is not supported.)

The following are examples of filter expressions that contain extended operations:

| Expression | Meaning |
|---|---|
| Evaluates to<br>TRUE<br>if the name of the object starts with<br>bill .<br>T} | |

| Expression | Meaning |
|---|---|
| `%i(%a, %v)` | Evaluates to result of applying the specified operation to the supplied arguments. |
| `(%a == %v) and 'name'('joe'*)` | Evaluates to `TRUE` if the specified attribute has the given value and if the name of the object starts with `joe` . |

**RETURN VALUES**

`fn_search_filter_create( )` returns a pointer to an `FN_search_filter_t` object if the operation succeeds; otherwise it returns a `NULL` pointer.

**ERRORS**

`fn_search_filter_create( )` returns a `NULL` pointer if the operation fails and sets *status* in the following way:

| | |
|---|---|
| `FN_E_SEARCH_INVALID_FILTER` | The filter expression had a syntax error or some other problem. |
| `FN_E_SEARCH_INVALID_OP` | An operator in the filter expression is not supported or, if the operator is an extended operator, the number of types of arguments supplied does not match the signature of the operation. |
| `FN_E_INVALID_ATTR_IDENTIFIER` | The left hand side of an operator expression was not an attribute. |
| `FN_E_INVALID_ATTR_VALUE` | The right hand side of an operator expression was not an integer, attribute value, or (wildcarded) string. |

Other status codes are possible as described in the reference manual pages for `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN) .

**EXAMPLES**

**EXAMPLE 1** Creating Different Filters

The following examples illustrate how to create three different filters.

The first example shows how to construct a filter involving substitution tokens and literals in the same filter expression. This example creates a filter for named objects whose `color` attribute contains a string value of `red` , `blue` , or `white` . The first two values are specified using substitution tokens; the last value, `white` , is specified as a literal in the expression.

```
unsigned int status;
extern FN_attribute_t *attr_color;
FN_string_t *red = fn_string_from_str((unsigned char *)"red");
FN_string_t *blue = fn_string_from_str((unsigned char *)"blue");
FN_search_filter_t *sfilter;
sfilter = fn_search_filter_create(
 &status,
 "(%a == %s) or (%a == %s) or (%a == ’white’)",
 attr_color, red, attr_color, blue,
 attr_color);
```

The second example illustrates how to construct a filter involving a wildcarded string. This example creates a filter for searching for named objects whose *last_name* attribute has a value that begins with the character `m` .

```
unsigned int status;
extern FN_attribute_t *attr_last_name;
FN_search_filter_t *sfilter;
sfilter = fn_search_filter_create(
 &status, "%a == ’m’*", attr_last_name);
```

The third example illustrates how to construct a filter involving extended operations. This example creates a filter for finding all named objects whose name ends with `ton` .

```
unsigned int status;
FN_search_filter_t *sfilter;
sfilter= fn_search_filter_create(&status, "’name’(*’ton’)");
```

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_attribute_t`(3XFN) , `FN_attrvalue_t`(3XFN) , `FN_identifier_t`(3XFN) , `FN_status_t`(3XFN) , `FN_string_t`(3XFN) , `fn_attr_ext_search`(3XFN) , `xfn_status_codes`(3XFN) , `attributes`(5)

**NAME**

FN_status_t, fn_status_create, fn_status_destroy, fn_status_copy, fn_status_assign, fn_status_code, fn_status_remaining_name, fn_status_resolved_name, fn_status_resolved_ref, fn_status_diagnostic_message, fn_status_link_code, fn_status_link_remaining_name, fn_status_link_resolved_name, fn_status_link_resolved_ref, fn_status_link_diagnostic_message, fn_status_is_success, fn_status_set_success, fn_status_set, fn_status_set_code, fn_status_set_remaining_name, fn_status_set_resolved_name, fn_status_set_resolved_ref, fn_status_set_diagnostic_message, fn_status_set_link_code, fn_status_set_link_remaining_name, fn_status_set_link_resolved_name, fn_status_set_link_resolved_ref, fn_status_set_link_diagnostic_message, fn_status_append_resolved_name, fn_status_append_remaining_name, fn_status_advance_by_name, fn_status_description – an XFN status object

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>
FN_status_t ***fn_status_create**(void);

void **fn_status_destroy**(FN_status_t **stat*);

FN_status_t ***fn_status_copy**(const FN_status_t **stat*);

FN_status_t ***fn_status_assign**(FN_status_t **dst*, const FN_status_t **src*);

unsigned int **fn_status_code**(const FN_status_t **stat*);

const FN_composite_name_t ***fn_status_remaining_name**(constFN_status_t **stat*);

const FN_composite_name_t ***fn_status_resolved_name**(constFN_status_t **stat*);

const FN_ref_t ***fn_status_resolved_ref**(constFN_status_t **stat*);

const FN_string_t ***fn_status_diagnostic_message**(constFN_status_t **stat*);

unsigned int **fn_status_link_code**(const FN_status_t **stat*);

const FN_composite_name_t ***fn_status_link_remaining_name**(constFN_status_t **stat*);

const FN_composite_name_t ***fn_status_link_resolved_name**(constFN_status_t **stat*);

const FN_ref_t ***fn_status_link_resolved_ref**(constFN_status_t **stat*);

const FN_string_t ***fn_status_link_diagnostic_message**(constFN_status_t **stat*);

int **fn_status_is_success**(const FN_status_t **stat*);

int **fn_status_set_success**(FN_status_t **stat*);

int **fn_status_set**(FN_status_t **stat*, unsigned int *code*, const FN_ref_t **resolved_ref*, const FN_composite_name_t **resolved_name*, const FN_composite_name_t **remaining_name*);

int **fn_status_set_code**(FN_status_t **stat*, unsigned int *code*);

int **fn_status_set_remaining_name**(FN_status_t **stat*, const FN_composite_name_t **name*);

int **fn_status_set_resolved_name**(FN_status_t **stat*, const FN_composite_name_t **name*);

int **fn_status_set_resolved_ref**(FN_status_t **stat*, const FN_ref_t **ref*);

int **fn_status_set_diagnostic_message**(FN_status_t **stat*, const FN_string_t **msg*);

int **fn_status_set_link_code**(FN_status_t **stat*, unsigned int *code*);

int **fn_status_set_link_remaining_name**(FN_status_t **stat*, const FN_composite_name_t **name*);

int **fn_status_set_link_resolved_name**(FN_status_t **stat*, const FN_composite_name_t **name*);

int **fn_status_set_link_resolved_ref**(FN_status_t **stat*, const FN_ref_t **ref*);

int **fn_status_set_link_diagnostic_message**(FN_status_t **stat*, const FN_string_t **msg*);

int **fn_status_append_resolved_name**(FN_status_t **stat*, const FN_composite_name_t **name*);

int **fn_status_append_remaining_name**(FN_status_t **stat*, const FN_composite_name_t **name*);

int **fn_status_advance_by_name**(FN_status_t **stat*, const FN_composite_name_t **prefix*, const FN_ref_t **resolved_ref*);

FN_string_t ***fn_status_description**(const FN_status_t **stat*, unsigned int *detail*, unsigned int **more_detail*);

## DESCRIPTION

The result status of operations in the context interface and the attribute interface is encapsulated in an `FN_status_t` object. This object contains information about how the operation completed: whether an error occurred in performing the operation, the nature of the error, and information that helps locate where the error occurred. In the case that the error occurred while resolving an XFN link, the status object contains additional information about that error.

The context status object consists of several items of information:

| | |
|---|---|
| primary status code | An `unsigned int` code describing the disposition of the operation. |

| | |
|---|---|
| resolved name | In the case of a failure during the resolution phase of the operation, this is the leading portion of the name that was resolved successfully. Resolution may have been successful beyond this point, but the error might not be pinpointed further. |
| resolved reference | The reference to which resolution was successful (in other words, the reference to which the resolved name is bound). |
| remaining name | The remaining unresolved portion of the name. |
| diagnostic message | This contains any diagnostic message returned by the context implementation. This message provides the context implementation a way of notifying the end-user or administrator of any implementation-specific information related to the returned error status. The diagnostic message could then be used by the end-user or administrator to take appropriate out-of-band action to rectify the problem. |
| link status code | In the case that an error occurred while resolving an XFN link, the primary status code has the value `FN_E_LINK_ERROR` and the link status code describes the error that occurred while resolving the XFN link. |
| resolved link name | In the case of a link error, this contains the resolved portion of the name in the XFN link. |
| resolved link reference | In the case of a link error, this contains the reference to which the resolved link name is bound. |
| remaining link name | In the case of a link error, this contains the remaining unresolved portion of the name in the XFN link. |
| link diagnostic message | In the case of a link error, this contains any diagnostic message related to the resolution of the link. |

Both the primary status code and the link status code are values of type `unsigned int` that are drawn from the same set of meaningful values. XFN reserves the values `0` through `127` for standard meanings. The values and interpretations for the codes are determined by XFN. See `xfn_status_codes`(3XFN) .

`fn_status_create( )` creates a status object with status `FN_SUCCESS` . `fn_status_destroy( )` releases the storage associated with *stat* . `fn_status_copy( )` returns a copy of the status object *stat* . `fn_status_assign( )` makes a copy of the status object *src* and assigns it to *dst* , releasing any old contents of *dst* . A pointer to the same object as *dst* is returned.

`fn_status_code( )` returns the status code. `fn_status_remaining_name( )` returns the remaining part of name to be resolved. `fn_status_resolved_name( )` returns the part of the composite name that has been resolved. `fn_status_resolved_ref( )` returns the reference to which resolution was successful. `fn_status_diagnostic_message` returns any diagnostic message set by the context implementation.

`fn_status_link_code( )` returns the link status code. `fn_status_link_remaining_name( )` returns the remaining part of the link name that has not been resolved. `fn_status_link_resolved_name( )` returns the part of the link name that has been resolved. `fn_status_link_resolved_ref( )` returns the reference to which resolution of the link was successful. `fn_status_link_diagnostic_message( )` returns any diagnostic message set by the context implementation during resolution of the link.

`fn_status_is_success( )` returns `1` if the status indicates success, `0` otherwise.

`fn_status_set_success( )` sets the status code to `FN_SUCCESS` and clears all other parts of *stat* . `fn_status_set( )` sets the non-link contents of the status object *stat* . `fn_status_set_code( )` sets the primary status code field of the status object *stat* . `fn_status_set_remaining_name( )` sets the remaining name part of the status object *stat* to *name* . `fn_status_set_resolved_name( )` sets the resolved name part of the status object *stat* to *name* . `fn_status_set_resolved_ref ( )` sets the resolved reference part of the status object*stat* to *ref* . `fn_status_set_diagnostic_message( )` sets the diagnostic message part of the status object to *msg* .

`fn_status_set_link_code( )` sets the link status code field of the status object *stat* to indicate why resolution of the link failed. `fn_status_set_link_remaining_name( )` sets the remaining link name part of the status object *stat* to *name* . `fn_status_set_link_resolved_name( )` sets the resolved link name part of the status object *stat* to *name* . `fn_status_set_link_resolved_ref( )` sets the resolved link reference part of the status object *stat* to *ref* .

`fn_status_set_link_diagnostic_message()` sets the link diagnostic message part of the status object to *msg* .

`fn_status_append_resolved_name()` appends as additional components *name* to the resolved name part of the status object *stat* . `fn_status_append_remaining_name()` appends as additional components *name* to the remaining name part of the status object *stat* . `fn_status_advance_by_name()` removes *prefix* from the remaining name, and appends it to the resolved name. The resolved reference part is set to *resolved_ref* . This operation returns `1` on success, `0` if the *prefix* is not a prefix of the remaining name.

**RETURN VALUES**

The `fn_status_set_*()` operations return `1` if the operation succeeds, `0` if the operation fails.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`FN_composite_name_t`(3XFN) , `FN_ref_t`(3XFN) , `FN_string_t`(3XFN) , `xfn`(3XFN) , `xfn_status_codes`(3XFN) , `attributes`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

## NAME

FN_string_t, fn_string_create, fn_string_destroy, fn_string_from_str, fn_string_from_str_n, fn_string_str, fn_string_from_contents, fn_string_code_set, fn_string_charcount, fn_string_bytecount, fn_string_contents, fn_string_copy, fn_string_assign, fn_string_from_strings, fn_string_from_substring, fn_string_is_empty, fn_string_compare, fn_string_compare_substring, fn_string_next_substring, fn_string_prev_substring – a character string

## SYNOPSIS

`cc` [ *flag* ... ] *file* ... `–lxfn` [ *library* ... ]
#include <xfn/xfn.h>
FN_string_t ***fn_string_create**(void);

void **fn_string_destroy**(FN_string_t **str*);

FN_string_t ***fn_string_from_str**(const unsigned char **cstr*);

FN_string_t ***fn_string_from_str_n**(const unsigned char **cstr*, size_t *n*);

const unsigned char ***fn_string_str**(const FN_string_t **str*, unsigned int **status*);

FN_string_t ***fn_string_from_contents**(unsigned long *code_set*, const void **locale_info*, size_t *locale_info_len*, size_t *charcount*, size_t *bytecount*, const void **contents*, unsigned int **status*);

unsigned long **fn_string_code_set**(const FN_string_t **str*, const void ***locale_info*, size_t **locale_info_len*);

size_t **fn_string_charcount**(const FN_string_t **str*);

size_t **fn_string_bytecount**(const FN_string_t **str*);

const void ***fn_string_contents**(const FN_string_t **str*);

FN_string_t ***fn_string_copy**(const FN_string_t **str*);

FN_string_t ***fn_string_assign**(FN_string_t **dst*, const FN_string_t **src*);

FN_string_t ***fn_string_from_strings**(unsigned int **status*, const FN_string_t **s1*, const FN_string_t **s2*, ...);

FN_string_t ***fn_string_from_substring**(constFN_string_t **str*, int *first*, int *last*);

int **fn_string_is_empty**(const FN_string_t **str*);

int **fn_string_compare**(const FN_string_t **str1*, const FN_string_t **str2*, unsigned int *string_case*, unsigned int **status*);

int **fn_string_compare_substring**(const FN_string_t **str1*, int *first*, int *last*, const FN_string_t **str2*, unsigned int *string_case*, unsigned int **status*);

int **fn_string_next_substring**(const FN_string_t **str*, const FN_string_t **sub*, int *index*, unsigned int *string_case*, unsigned int **status*);

int **`fn_string_prev_substring`**(const FN_string_t **str*, const FN_string_t **sub*, int *index*, unsigned int *string_case*, unsigned int **status*);

## DESCRIPTION

The `FN_string_t` type is used to represent character strings in the XFN interface. It provides insulation from specific string representations.

The `FN_string_t` supports multiple code sets. It provides creation functions for character strings of the code set of the current locale setting and a generic creation function for arbitrary code sets. The degree of support for the functions that manipulate `FN_string_t` for arbitrary code sets is implementation-dependent. An XFN implementation is required to support the `ISO 646` code set; all other code sets are optional.

`fn_string_destroy()` releases the storage associated with the given string.

`fn_string_create()` creates an empty string.

`fn_string_from_str()` creates an `FN_string_t` object from the given null terminated string based on the code set of the current locale setting. The number of characters in the string is determined by the code set of the current locale setting. `fn_string_from_str_n()` is like `fn_string_from_str()` except only *n* characters from the given string are used. `fn_string_str()` returns the contents of the given string *str* in the form of a null terminated string in the code set and current locale setting.

`fn_string_from_contents()` creates an `FN_string_t` object using the specified code set *code_set* , locale information *locale_info* , and data in the given buffer *contents* . *bytecount* specifies the number of bytes in *contents* and *charcount* specifies the number of characters represented by *contents* .

`fn_string_code_set()` returns the code set associated with the given string object and, if present, the locale information in *locale_info* . `fn_string_charcount()` returns the number of characters in the given string object. `fn_string_bytecount()` returns the number of bytes used to represent the given string object. `fn_string_contents()` returns a pointer to the contents of the given string object.

`fn_string_copy()` returns a copy of the given string object. `fn_string_assign()` makes a copy of the string object *src* and assigns it to *dst* , releasing any old contents of *dst* . A pointer to the same object as *dst* is returned. `fn_string_from_strings()` is a function that takes a variable number of arguments (minimum of 2), the last of which must be `NULL` (`0` ); it returns a new string object composed of the left to right concatenation of the given strings, in the given order. The support for strings with different code sets and/or locales as arguments to a single invocation of `fn_string_from_strings()` is implementation-dependent. `fn_string_from_substring()` returns a new string object consisting of the characters located between *first* and `last` inclusive from *str* . Indexing begins with `0` . If `last` is `FN_STRING_INDEX_LAST` or exceeds the length of the string, the index of the last character of the string is used.

`fn_string_is_empty( )` returns whether *str* is an empty string.

Comparison of two strings must take into account code set and locale information. If strings are in the same code set and same locale, case sensitivity is applied according to the case sensitivity rules applicable for the code set and locale; case sensitivity may not necessarily be relevant for all string encodings. If *string_case* is non-zero, case is significant and equality for strings of the same code set is defined as equality between byte-wise encoded values of the strings. If *string_case* is zero, case is ignored and equality for strings of the same code set is defined using the definition of case-insensitive equality for the specific code set. Support for comparison between strings of different code sets, or lack thereof, is implementation-dependent.

`fn_string_compare( )` compares strings *str1* and *str2* and returns `0` if they are equal, non-zero if they are not equal. If two strings are not equal, `fn_string_compare( )` returns a positive value if the difference of *str2* precedes that of *str1* in terms of byte-wise encoded value (with case-sensitivity taken into account when *string_case* is non-zero), and a negative value if the difference of *str1* precedes that of *str2* , in terms of byte-wise encoded value (with case-sensitivity taken into account when *string_case* is non-zero). Such information (positive versus negative return value) may be used by applications that use strings of code sets in which ordering is meaningful; this information is not of general use in internationalized environments. `fn_string_compare_substring( )` is similar to `fn_string_compare( )` except that `fn_string_compare_substring( )` compares characters between *first* and `last` inclusive of *str2* with *str1* . Comparison of strings with incompatible code sets returns a negative or positive value (never `0` ) depending on the implementation.

`fn_string_next_substring( )` returns the index of the next occurrence of *sub* at or after *index* in the string *str* . `FN_STRING_INDEX_NONE` is returned if *sub* does not occur. `fn_string_prev_substring( )` returns the index of the previous occurrence of *sub* at or before *index* in the string *str* . `FN_STRING_INDEX_NONE` is returned if *sub* does not occur. In both of these functions, *string_case* specifies whether the search should take case-sensitivity into account.

**ERRORS**

`fn_string_str( )` returns `0` and sets *status* to `FN_E_INCOMPATIBLE_CODE_SETS` if the given string's representation cannot be converted into the code set of the current locale setting. It is implementation-dependent which code sets can be converted into the code set of the current locale.

Code set mismatches that occur during concatenation, searches, or comparisons are resolved in an implementation-dependent way. When an implementation discovers that arguments to substring searches and comparison operations have incompatible code sets, it sets *status* to `FN_E_INCOMPATIBLE_CODE_SETS` . In such cases, `fn_string_from_strings()` returns `0` . The returned value for comparison operations when there is code set or locale incompatibility is either negative or positive (greater than `0` ); it is never `0` .

`fn_string_from_contents()` returns `0` and *status* is set to `FN_E_INCOMPATIBLE_CODE_SETS` if the supplied code set and / or locale information are not supported by the XFN implementation.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`xfn`(3XFN) , `attributes`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X / Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**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*, size_t *hostlen*, char **serv*, size_t *servlen*, int *flags*);

void **freeaddrinfo**(struct addrinfo **ai*);

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 either a node name or a numeric host address string (a dotted-decimal IPv4 address or an IPv6 hex address). 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 third 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 */
int              ai_family;     /* PF_xxx */
int              ai_socktype;   /* SOCK_xxx */
int              ai_protocol;   /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
size_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* .

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.

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 the function . 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 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 structuresgetaddrinfo( ) is returned by the fourth argument and the function returns 0 . 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.

## ERRORS

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

```
EAI_ADDRFAMILY  address family for nodename not supported
EAI_AGAIN       temporary failure in name resolution
EAI_BADFLAGS    invalid value for ai_flags
EAI_FAIL        non-recoverable failure in name resolution
EAI_FAMILY      ai_family not supported
EAI_MEMORY      memory allocation failure
EAI_NODATA      no address associated with nodename
EAI_NONAME      nodename nor servname provided, or not known
EAI_SERVICE     servname not supported for ai_socktype
EAI_SOCKTYPE    ai_socktype not supported
EAI_SYSTEM      system error returned in errno
```

## FILES

/etc/inet/hosts
/etc/inet/ipnodes
/etc/netconfig
/etc/nsswitch.conf

## SEE ALSO

gethostbyname(3NSL) , getipnodebyname(3SOCKET) , htonl(3SOCKET) , inet(3SOCKET) , netdb(3HEAD) , socket(3SOCKET) , hosts(4) , ipnodes(4) , nsswitch.conf(4)

**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 may come from any of the sources for `hosts` specified in the `/etc/nsswitch.conf` file. See `nsswitch.conf`(4) . Please take note that these functions have been superseded by the newer functions, `getipnodebyname`(3SOCKET) , `getipnodebyaddr`(3SOCKET) , and `getaddrinfo`(3SOCKET) . The newer functions provide greater portability to applications when multithreading is done or technologies such as IPv6 are used. For example, the functions described below cannot be used with applications targeted to work with IPv6.

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

`gethostbyaddr( )` 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>` . The parameter *addr* must be a pointer to a buffer containing the address. The address is given in a form specific to the address family. See the `NOTES` section below for more information. Also see the `EXAMPLES` section below on how to convert a “.” separated Internet IP address notation into the *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 functions `sethostent( )` , `gethostent( )` , and `endhostent( )` are used to enumerate host entries from the database.

`sethostent()` 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 may keep allocated resources such as open file descriptors until a subsequent call to `endhostent()` .

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

`endhostent()` may be called to indicate that the caller expects to do no further host 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 host retrieval functions after calling `endhostent()` .

**Reentrant Interfaces**

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

The functions `gethostbyname_r()` , `gethostbyaddr_r()` , and `gethostent_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 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 RETURN VALUES . 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 ERRORS .

For enumeration in multi-threaded applications, the position within the enumeration is a process-wide property shared by all threads. `sethostent()` may be used in a multi-threaded 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 below for information about how to retrieve a "." separated Internet IP address string from the *h_addr_list* field of `struct hostent` .

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

The functions `gethostent( )` and `gethostent_r( )` 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 functions `gethostbyname( )` , `gethostbyaddr( )` , and `gethostent( )` 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 functions `sethostent( )` and `endhostent( )` 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.`

Note however that 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.

`gethostbyname()` will set *h_errno* to `NETDB_INTERNAL` when it returns a `NULL` value.

## EXAMPLES

**EXAMPLE 1** Using `gethostbyname()`

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 <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)
{
 ulong_t addr;
 struct hostent *hp;
 char **p;
 if (argc != 2) {
     (void) printf("usage: %s IP-address\
", 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\
");
     exit (2);
 }
 hp = gethostbyaddr((char *)&addr, sizeof (addr), AF_INET);
 if (hp == NULL) {
     (void) printf("host information for %s not found\
", 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\\t%s", inet_ntoa(in), hp->h_name);
     for (q = hp->h_aliases; *q != 0; q++)
         (void) printf(" %s", *q);
     (void) putchar('\
');
 }
 exit (0);
}
```

Note that the above sample program is unsafe for use in multithreadeded applications.

## FILES

`/etc/hosts`
`/etc/netconfig`
`/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**

`Intro`(2) , `Intro`(3) , `byteorder`(3SOCKET) , `inet`(3SOCKET) , `netdir`(3NSL) , `hosts`(4) , `netconfig`(4) , `nsswitch.conf`(4) , `attributes`(5) , `netdb`(3HEAD)

**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**

Programs that use the interfaces described in this manual page cannot be linked statically since the implementations of these functions employ dynamic loading and linking of shared objects at run time.

In order 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; otherwise the 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 may 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 will strive to put the addresses on the same subnet as that of the caller first.

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 `gethostent( )` and `gethostent_r( )` is discouraged; enumeration may 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 above, are helpful in constructing and manipulating addresses in this form.

**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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`uname`(1), `gethostid`(3C), `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*);

**DESCRIPTION**

The `getipnodebyname()` function searches the ipnodes database from the beginning and finds the first entry for which the hostname specified by name matches the `h_name` member. It takes an *af* argument which specifies the address family, which can be either `AF_INET` for IPv4 addresses or `AF_INET6` for IPv6 addresses. The *flags* argument determines what results will be returned based on the value of *flags* . If the *flags* argument is set to 0 (zero), then the default operation of this function is specified as follows:

- If the *af* argument is `AF_INET` , then a query is made for an IPv4 address. If successful, IPv4 addresses are returned and the `h_length` member of the `hostent` structure will be 4. Otherwise, the function returns a null pointer.
- If the *af* argument is `AF_INET6` , then a query is made for an IPv6 address. If successful, IPv6 addresses are returned and the `h_length` member of the `hostent` structure will be 16. Otherwise, the function returns a null pointer.

The *flags* argument will change the default actions of the function. The *flags* argument can be set by logically `OR` ing any of the following values together:

`AI_V4MAPPED`

`AI_ALL`

`AI_ADDRCONFIG`

Note that a special flags value of `AI_DEFAULT` as defined below should handle most applications. That is, porting simple applications to use IPv6 replaces the call

```
hptr = gethostbyname(name);
```

with

```
hptr = getipnodebyname(name, AF_INET6, AI_DEFAULT);
```

A *flags* of 0 implies a strict interpretation of the *af* argument:

- If flags is 0 and *af* is `AF_INET` , then the caller wants only IPv4 addresses. A query is made for `A` records. If successful, the IPv4 addresses are returned and the `h_length` member of the `hostent` structure will be 4; otherwise, the function returns a null pointer.

- If *flags* is 0, and if *af* is `AF_INET6` , then the caller wants only IPv6 addresses. A query is made for AAAA records. If successful, the IPv6 addresses are returned and the `h_length` member of the `hostent` structure will be 16; otherwise, the function returns a null pointer.

Other constants can be logically-OR ed into the *flags* argument, to modify the behavior of the function.

- If the `AI_V4MAPPED` flag is specified along with an *af* of `AF_INET6` , then the caller will accept IPv4-mapped IPv6 addresses. That is, if no AAAA records are found, then a query is made for A records, and any found are returned as IPv4-mapped IPv6 addresses (`h_length` will be 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, and is only used with the IPv6 address family. When `AI_ALL` is logically OR 'd with `AI_V4MAPPED` flag then the caller wants all addresses: IPv6 and IPv4-mapped IPv6. A query is first made for AAAA records and if successful, the IPv6 addresses are returned. Another query is then made for A records, and any found are returned as IPv4-mapped IPv6 addresses. `h_length` will be 16. Only if both queries fail does the function return a null pointer. This flag is ignored unless *af* equals `AF_INET6` .
- The `AI_ADDRCONFIG` flag specifies that a query for AAAA records should occur only if the node has at least one IPv6 source address configured and a query for A records should occur only if the node has at least one IPv4 source address configured. For example, if the node has no IPv6 source addresses configured, and *af* equals `AF_INET6` , and the node name being looked up has both AAAA and A records, then

  1. If only `AI_ADDRCONFIG` is specified, the function returns a null pointer;
  2. If `AI_ADDRCONFIG` or `AI_V4MAPPED` is specified, the A records are returned as IPv4-mapped IPv6 addresses;

The special flags value of `AI_DEFAULT` is defined as

```
#define  AI_DEFAULT  (AI_V4MAPPED | AI_ADDRCONFIG)
```

The `getipnodebyname( )` function must allow the *name* argument to be either a node name or a literal address string, that is, a dotted-decimal IPv4 address or an IPv6 hex address. This saves applications from having to call `inet_pton`(3SOCKET) to handle literal address strings.

There are four scenarios based on the type of literal address string and the value of the *af* argument. The two simple cases are when *name* is a dotted-decimal IPv4 address and *af* equals `AF_INET` , or when *name* is an IPv6 hex address and *af* equals `AF_INET6` . The members of the returned `hostent` structure are:

| | |
|---|---|
| `h_name` | points to a copy of the name argument |
| `h_aliases` | is a null pointer. |
| `h_addrtype` | is a copy of the *af* argument. |
| `h_length` | is either 4 (for `AF_INET` ) or 16 (for `AF_INET6` ). |
| `h_addr_list[0]` | is a pointer to the 4-byte or 16-byte binary address. |
| `h_addr_list[1]` | is a null pointer |

## 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 |

## RETURN VALUES

Upon successful completion, `getipnodebyname( )` and `getipnodebyaddr( )` return a `hostent` structure. Otherwise they return `NULL` .

The `hostent` structure does not change from its 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 */
          };
```

It is an error when *name* is an IPv6 hex address and *af* equals `AF_INET` . The function's return value 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 may 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 is not a soft error. Another type of *name* server request may be successful. |
| NO_RECOVERY | An unexpected server failure occurred. This is a nonrecoverable error. |
| TRY_AGAIN | This 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 may 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 all 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\
", 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 |
        AI_V4MAPPED, &error_num);
     if (hp == NULL) {
        if (error_num == TRY_AGAIN) {
            printf("%s: unknown host or invalid literal address "
                "(try again later)\
", argv[1]);
        } else {
            printf("%s: unknown host or invalid literal address\
",
                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('\
');
     }
     freehostent(hp);
     exit (0);
     }
```

**FILES**

/etc/inet/hosts
/etc/inet/ipnodes
/etc/netconfig
/etc/nsswitch.conf

**SEE ALSO**

getaddrinfo(3SOCKET) , gethostbyname(3NSL) , htonl(3SOCKET) , inet(3SOCKET) , netdb(3HEAD) , hosts(4) , ipnodes(4) , nsswitch.conf(4)

**NOTES**

Programs that use the interfaces described in this manual page cannot be linked statically since the implementations of these functions employ dynamic loading and linking of shared objects at run time.

There is no enumeration functions provided for IPv6. Existing enumeration functions, for example, sethostent(3NSL) , will not work in combination with getipnodebyname( ) and getipnodebyaddr( ) .

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 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 current implementations of these functions only return or accept addresses for the Internet address family (type AF_INET ) or the Internet address family Version 6 (type AF_INET6 ).

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 defined also 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**

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.

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

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`
`/etc/nsswitch.conf`

**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), `networks`(4), `nsswitch.conf` (4), `attributes`(5), `netdb`(3HEAD)

**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.

Programs that use the interfaces described in this manual page cannot be linked statically since the implementations of these functions employ dynamic loading and linking of shared objects at run time.

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)

*ONC+ Developer's Guide Transport Interfaces Programming Guide*

## 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)

*ONC+ Developer's Guide Transport Interfaces Programming Guide*

## NAME

getpeername – get name of connected peer

## SYNOPSIS

```
cc [ flag ... ] file ... –lsocket –lnsl [ library ... ]
#include <sys/types.h>
#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`(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 **address*, socklen_t **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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`accept`(3XNET), `bind`(3XNET), `getsockname`(3XNET), `socket`(3XNET), `attributes`(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 routines 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 routines may be used in single-threaded applications and are safe for multi-threaded 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) ), the NIS maps “protocols.byname” and “protocols.bynumber”, and the NIS+ table “protocols”. 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 NIS+ or DNS will return all aliases.

`getprotobyname_r()` and `getprotobynumber_r()` 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 routines are unsafe in a multi-threaded application.

`getprotoent_r( )` enumerates protocol entries: successive calls to `getprotoent_r( )` will return either successive protocol entries or NULL. Enumeration may not be supported by some sources. Note that 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 multi-threaded 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( )` , `getprotobyname( )` , `getprotobynumber_r( )` and `getprotobynumber( )` may leave the enumeration in an indeterminate state, so `setprotoent( )` should be called before the first `getprotoent_r( )` or `getprotoent( )` . Note that `setprotoent( )` has process-wide scope, and “rewinds” the protocol entries for all threads calling `getprotoent_r( )` as well as main-thread calls to `getprotoent( )` .

`endprotoent( )` may 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 routines 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**

`getprotobyname_r()`, `getprotobyname()`, `getprotobynumber_r()`, and `getprotobynumber( )` return a pointer to a `struct protoent` if they successfully locate the requested entry; otherwise they return NULL.

`getprotoent_r( )` and `getprotoent( )` return a pointer to a `struct protoent` if they successfully enumerate an entry; otherwise they return NULL, indicating the end of the enumeration.

**ERRORS**

`getprotobyname_r()`, `getprotobynumber_r()`, and `getprotoent_r()` will fail if the following is true:

| | |
|---|---|
| ERANGE | length of the buffer supplied by 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`(3HEAD)

**NOTES**

Although `getprotobyname_r()` , `getprotobynumber_r()` , and `getprotoent_r()` are not mentioned by POSIX.4a Draft 6, they were added to complete the functionality provided by similar thread-safe functions. These interfaces are subject to change to be compatible with the "spirit" of POSIX.4a when it is approved as a standard.

When compiling multithreaded applications, see `intro`(3) , *Notes On Multithread Applications* , for information about the use of the `_REENTRANT` flag.

The routines `getprotobyname_r()` , `getprotobynumber_r()` , and `getprotoent_r()` are reentrant and multi-thread safe. The reentrant interfaces can be used in single-threaded as well as multi-threaded applications and are therefore the preferred interfaces.

The routines `getprotobyname()` , `getprotobyaddr()` , and `getprotoent()` 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 routines are not safe for multi-threaded applications.

`setprotoent()` and `endprotoent()` 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 and 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.

Programs that call `getprotobyname_r()` or `getprotobynumber_r()` routines cannot be linked statically since the implementation of these routines requires dynamic linker functionality to access shared objects at run time.

**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**

`getpublickey( )` and `getsecretkey( )` get public and secret keys for *netname* . The key may come from one of the following sources: the `/etc/publickey` file (see `publickey`(4) ) or the NIS map “publickey.byname” or the NIS+ table “cred.org_dir”. The sources and their lookup order are specified in the `/etc/nsswitch.conf` file (see `nsswitch.conf`(4) ).

`getsecretkey( )` has an extra argument, `passwd` , 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, `0` otherwise. 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)

**WARNINGS**

If `getpublickey( )` gets the public key from any source other than NIS+, all authenticated NIS+ operations may fail. To ensure that this does not happen, edit the `nsswitch.conf`(4) file to make sure that the public key is obtained from NIS+.

## 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

Programs that use the interfaces described in this manual page cannot be linked statically since the implementations of these functions employ dynamic loading and linking of shared objects at run time.

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) .

`getservbyname( )` and `getservbyport( )` 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.

`getservbyname( )` searches for an entry with the Internet service name specified by the parameter *name* .

`getservbyport( )` 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 may be returned.

The functions `setservent( )` , `getservent( )` , and `endservent( )` are used to enumerate entries from the services database.

`setservent( )` 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( )` .

`getservent( )` 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.

`endservent( )` closes the file. `endservent( )` may be called to indicate that the caller expects to do no further service 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 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 man 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. `setservent( )` may 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( )` 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/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(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.

Programs that use the interfaces described in this manual page cannot be linked statically since the implementations of these functions employ dynamic loading and linking of shared objects at run time.

In order 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/types.h>
#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 **address*, socklen_t **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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`accept`(3XNET), `bind`(3XNET), `getpeername`(3XNET), `socket`(3XNET)
`attributes`(5)

**NAME**

getsockopt, setsockopt – get and set options on sockets

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lsocket –lnsl` [ *library* ... ]

#include <sys/types.h>

#include <sys/socket.h>

int **`getsockopt`**(int *s*, int *level*, int *optname*, void **optval*, int **optlen*);

int **`setsockopt`**(int *s*, int *level*, int *optname*, const void **optval*, int *optlen*);

**DESCRIPTION**

`getsockopt( )` and `setsockopt( )` manipulate options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level.

When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the "socket" level, *level* is specified as `SOL_SOCKET` . To manipulate options at any other level, *level* is the protocol number of the protocol that controls the option. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* is set to the TCP protocol number (see `getprotobyname`(3SOCKET) ).

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.

*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 (see below). `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_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_TYPE | get the type of the socket (get only) |
| SO_ERROR | get and clear error on the socket (get only) |

SO_DEBUG enables debugging in the underlying protocol modules. SO_REUSEADDR indicates that the rules used in validating addresses supplied in a bind(3SOCKET) call should allow reuse of local addresses. SO_KEEPALIVE 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 processes using the socket are notified using a SIGPIPE signal. SO_DONTROUTE 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.

SO_LINGER 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 process 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 process 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.

SO_SNDBUF and SO_RCVBUF are options that 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. SunOS sets the maximum buffer size for both UDP and TCP to 256 Kbytes.

By default, delayed errors (such as ICMP port unreachable packets) are returned only for connected datagram sockets. `SO_DGRAM_ERRIND` 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) .

Finally, `SO_TYPE` and `SO_ERROR` are options used only with `getsockopt()` . `SO_TYPE` returns the type of the socket (for example, `SOCK_STREAM` ). It is useful for servers that inherit sockets on startup. `SO_ERROR` 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.

## RETURN VALUES

If successful, `getsockopt()` 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. |
| `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. |

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

## SEE ALSO

`close`(2) , `ioctl`(2) , `read`(2) , `bind`(3SOCKET) , `getprotobyname`(3SOCKET) , `recv`(3SOCKET) , `send`(3SOCKET) , `socket`(3SOCKET) , `attributes`(5)

**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 **option_value*, socklen_t **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 may 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.<br><br>If the connected socket fails to respond to these messages, the connection is broken and processes writing to that socket are notified with a `SIGPIPE` signal. This option stores an `int` value.<br><br>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. |

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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`close`(2), `bind`(3XNET), `endprotoent`(3XNET), `setsockopt`(3XNET), `socket`(3XNET), `attributes`

**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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`endhostent`(3XNET) , `endservent`(3XNET) , `attributes`(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* ... −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*);

## 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 "le0", "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: "le0", ... */
};
```

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()`.

## PARAMETERS

| | |
|---|---|
| *ifname* | interface name. |
| *ifindex* | interface index. |
| *ptr* | pointer returned by `if_nameindex()`. |

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Availability | SUNWcsl (32-bit)<br>SUNWcslx (64-bit) |
| MT Level | MT Safe |
| Interface Stability | Standard |

## SEE ALSO

`ifconfig`(1M), `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*);

**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 "le0", "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: "le0", ... */
};
```

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( )` .

## PARAMETERS

| | |
|---|---|
| *ifname* | interface name. |
| *ifindex* | interface index. |
| *ptr* | pointer returned by `if_nameindex( )` . |

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Availability | SUNWcsl (32-bit)<br>SUNWcslx (64-bit) |
| MT Level | MT Safe |
| Interface Stability | Standard |

## SEE ALSO

`ifconfig`(1M) , `attributes`(5) , `if`(7P)

**NAME**

inet, inet6, inet_ntop, inet_pton, inet_addr, inet_network, inet_makeaddr, inet_lnaof, inet_netof, inet_ntoa – Internet address manipulation

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lsocket –lnsl` [ *library* ... ]
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
const char ***inet_ntop**(int *af*, const void **addr*, char **cp*, size_t *size*);

int **inet_pton**(int *af*, const char **cp*, void **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*, const int *lna*);

int **inet_lnaof**(const struct in_addr *in*);

int **inet_netof**(const struct in_addr *in*);

char ***inet_ntoa**(const struct in_addr *in*);

**DESCRIPTION**

The `inet_ntop()` and `inet_pton()` routines can manipulate both IPv4 and IPv6 addresses, whereas `inet_addr()`, `inet_network()`, `inet_makeaddr()`, `inet_lnaof()`, `inet_netof()`, and `inet_ntoa()` can only manipulate IPv4 addresses.

The `inet_ntop()` routine converts a numeric address into a string suitable for presentation. The *af* argument specifies the family of the address. This can be `AF_INET` or `AF_INET6`. The *addr* 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 address must be in network byte order. The *cp* argument points to a buffer where the routine will store the resulting string. The *size* argument specifies the size of this buffer. The application must specify a non-`NULL` *cp* argument. For IPv6 addresses, the buffer must be at least 46-octets. For IPv4 addresses, the buffer must be at least 16-octets. In order 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()` routine converts an address in its standard text presentation form 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 into which the routine 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_addr( )` and `inet_network( )` routines interpret character strings representing numbers expressed in the IPv4 standard '`.` ' notation, returning numbers suitable for use as IPv4 addresses and IPv4 network numbers, respectively. The routine `inet_makeaddr( )` takes an IPv4 network number and a local network address and constructs an IPv4 address from it. The routines `inet_netof( )` and `inet_lnaof( )` break apart IPv4 host addresses, returning the network number and local network address part, respectively.

The `inet_ntoa( )` routine returns a pointer to a string in the base 256 notation `d.d.d.d` . See INTERNET 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.

## INTERNET ADDRESSES

### 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
   ```

   Note that it is not necessary to write the leading zeros in an individual field. However, there must be at least one numeral in every field, except as described below.

2. Due to some methods of allocating certain styles of IPv6 addresses, it will be common for addresses to contain long strings of zero bits. In order to make writing addresses containing zero bits easier, a special syntax is available to compress the zeros. The use of "`::` " indicates multiple groups of 16-bits of zeros. The "`::` " can only appear once in an address. The "`::` " can also be used to compress the leading and/or trailing zeros in an address. For example,

   ```
   1080::8:800:200C:417A
   ```

3. An alternative 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 standard IPv4 representation address, for example,

   ```
   ::FFFF:129.144.52.38
   ::129.144.52.38
   ```

where "`::FFFF:d.d.d.d`" and "`::d.d.d.d`" are, respectively, the general forms of an IPv4-mapped IPv6 address and an IPv4-compatible IPv6 address. Note that 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 following form:

```
::FFFF:d
```

is valid, however it is an unconventional representation of the IPv4-compatible IPv6 address,

```
::255.255.0.d
```

while "`::d`" 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 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. This makes the three part address format convenient for specifying Class B network addresses 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. This makes the two part address format convenient for specifying Class A network addresses 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 the C language. For example, a leading `0x` or `0X` implies hexadecimal; otherwise, a leading `0` implies octal; otherwise, the number is interpreted as decimal.

For IPv4 addresses, `inet_pton()` only accepts a string in the standard IPv4 dotted-decimal form:

```
d.d.d.d
```

where each number has one to three digits with a decimal value between 0 and 255.

**RETURN VALUES**

The `inet_ntop()` routine returns a pointer to the buffer containing a string if the conversion succeeds, and `NULL` otherwise. Upon failure, `errno` is set to `EAFNOSUPPORT` if the *af* argument is invalid or `ENOSPC` if the size of the result buffer is inadequate.

`inet_pton()` returns `1` if the conversion succeeds, `0` if the input is not a valid IPv4 dotted-decimal string or a valid IPv6 address string, or `-1` with `errno` set to `EAFNOSUPPORT` if the af argument is unknown.

The value `-1` is returned by `inet_addr()` and `inet_network()` for malformed requests.

The routines `inet_netof()` and `inet_lnaof()` break apart IPv4 host addresses, returning the network number and local network address part, respectively.

The routine `inet_ntoa()` returns a pointer to a string in the base 256 notation `d.d.d.d` described in INTERNET ADDRESSES.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

**SEE ALSO**

`gethostbyname`(3NSL) , `getipnodebyname`(3SOCKET) , `getnetbyname`(3SOCKET) , `inet`(3HEAD) , `hosts`(4) , `ipnodes`(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.

**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

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 |
|---|---|
| MT-Level | Unsafe |

**SEE ALSO**

`endhostent`(3XNET) , `endnetent`(3XNET) , `attributes`(5)

## NAME

kerberos, krb_mk_req, krb_rd_req, krb_kntoln, krb_set_key, krb_get_cred, krb_mk_safe, krb_rd_safe, krb_mk_err, krb_rd_err – Kerberos authentication library

## SYNOPSIS

`cc` [ *flag* ... ] *file* ... `–lkrb` [ *library* ... ]
#include <kerberos/krb.h>
extern char *krb_err_txt[];
int **krb_mk_req**(KTEXT *authent*, const char **service*, const char **instance*, const char **realm*, const long *checksum*);

int **krb_rd_req**(const KTEXT *authent*, const char * *service*, char * *instance*, const long *from_addr*, AUTH_DAT **ad*, const char **fn*);

int **krb_kntoln**(const AUTH_DAT **ad*, char **lname*);

int **krb_set_key**(const char **key*, const int *cvt*);

int **krb_get_cred**(const char **service*, const char **instance*, const char **realm*, CREDENTIALS **c*);

long **krb_mk_safe**(const uchar_t **in*, uchar_t **out*, const ulong_t *in_length*, const des_cblock **key*, const struct sockaddr_in **sender*, const struct sockaddr_in **receiver*);

long **krb_rd_safe**(const uchar_t **in*, const ulong_t *length*, const des_cblock **key*, const struct sockaddr_in **sender*, const struct sockaddr_in **receiver*, MSG_DAT **msg_data*);

long **krb_mk_err**(uchar_t **out*, const long *code*, const char **string*);

long **krb_rd_err**(const uchar_t **in*, const ulong_t *length*, long **code*, MSG_DAT **msg_data*);

## DESCRIPTION

This library supports network authentication and various related operations. The library contains many routines beyond those described in this man page, but they are not intended to be used directly. Instead, they are called by the routines that are described, the authentication server and the login program.

`krb_err_txt[]` contains text string descriptions of various Kerberos error codes returned by some of the routines below.

`krb_mk_req()` takes a pointer to a text structure in which an authenticator is to be built. It also takes the name, instance, and realm of the service to be used and an optional checksum. It is up to the application to decide how to generate the checksum. `krb_mk_req()` then retrieves a ticket for the desired service and creates an authenticator. The authenticator is built in *authent* and is accessible to the calling procedure.

It is up to the application to get the authenticator to the service where it will be read by `krb_rd_req()` . Unless an attacker possesses the session key contained in the ticket, it will be unable to modify the authenticator. Thus, the checksum can be used to verify the authenticity of the other data that will pass through a connection.

`krb_mk_req( )` returns KSUCCESS if successful, otherwise a Kerberos error code as defined in `<kerberos/krb.h>`.

`krb_rd_req( )` takes an authenticator of type KTEXT, a service name, an instance, the address of the host originating the request, and a pointer to a structure of type AUTH_DAT which is filled in with information obtained from the authenticator. It also optionally takes the name of the file in which it will find the secret key(s) for the service. If the supplied *instance* is "*", then the first service key with the same service name found in the service key file will be used, and the *instance* argument will be filled in with the chosen instance. This means that the caller must provide space for such an instance name.

If the last argument is the null string (""), `krb_rd_req( )` will use the file `/etc/srvtab` to find its keys. If the last argument is NULL, it will assume that the key has been set by `krb_set_key( )` and will not bother looking further.

`krb_rd_req( )` is used to find out information about the principal when a request has been made to a service. It is up to the application protocol to get the authenticator from the client to the service. The authenticator is then passed to `krb_rd_req( )` to extract the desired information.

`krb_rd_req( )` returns zero (RD_AP_OK) upon successful authentication. If a packet was forged, modified, or replayed, authentication will fail. If the authentication fails, a non-zero value is returned indicating the particular problem encountered. See `<kerberos/krb.h>` for the list of error codes.

`krb_kntoln( )` converts a Kerberos name to a local name. It takes a structure of type AUTH_DAT and uses the name, instance, and realm to determine the corresponding local name. A valid local name is returned if the instance is NULL and the realm is the same as the local realm. The local name returned is the Kerberos name and can be used by an application to change uids, directories, or other parameters. This routine is not an integral part of Kerberos, but is provided to support the use of Kerberos in existing utilities. This routine returns KSUCCESS or KFAILURE.

`krb_set_key( )` takes as an argument a DES key. It then creates a key schedule from it and saves the original key to be used as an initialization vector. It is used to set the server’s key which must be used to decrypt tickets.

If called with a non-zero second argument, `krb_set_key( )` will first convert the input from a string of arbitrary length to a DES key by encrypting it with a one-way function.

In most cases it should not be necessary to call `krb_set_key( )`. The necessary keys will usually be obtained and set inside `krb_rd_req( )`. `krb_set_key( )`

is provided for those applications that do not wish to place the application keys on disk. It returns 0 for success, otherwise a non-zero value.

`krb_get_cred( )` searches the caller's ticket file for a ticket for the given *service* , *instance* , and *realm* . If a ticket is found, the given CREDENTIALS structure is filled in with the ticket information.

If the ticket was found, `krb_get_cred( )` returns GC_OK. If the ticket file cannot be found, cannot be read, does not belong to the user (other than root), is not a regular file, or is in the wrong mode, the error GC_TKFIL is returned.

`krb_mk_safe( )` creates an authenticated, but unencrypted message from any arbitrary application data, pointed to by *in* and *in_length* bytes long. The private session key, pointed to by *key,* is used to seed the `quad_cksum( )` checksum algorithm used as part of the authentication. *sender* and *receiver* point to the Internet address of the two parties. This message does not provide privacy, but does protect (via detection) against modifications, insertions or replays. The encapsulated message and header are placed in the area pointed to by *out* and the routine returns the length of the output, or -1 indicating an error.

`krb_rd_safe( )` authenticates a received `krb_mk_safe( )` message. *in* points to the beginning of the received message, whose length is specified in *in_length* . The private session key, pointed to by *key* , is used to seed the `quad_cksum( )` routine as part of the authentication. *msg_data* is a pointer to a MSG_DAT struct, defined in `<kerberos/krb.h>` . The routine fills in these MSG_DAT fields: the *app_data* field with a pointer to the application data, *app_length* with the length of the *app_data* field, *time_sec* and *time_5ms* with the timestamps in the message, and `swap` with a 1 if the byte order of the receiver is different than that of the sender. (The application must still determine if it is appropriate to byte-swap application data; the Kerberos protocol fields are already taken care of.)

The routine returns zero if successful, or a Kerberos error code. Modified messages and old messages cause errors, but it is up to the caller to check the time sequence of messages, and to check against recently replayed messages.

`krb_mk_err( )` constructs an application level error message that may be used along with `krb_mk_safe( )` . *out* is a pointer to the output buffer, *code* is an application specific error code, and *string* is an application specific error string. This routine returns the length of the error reply.

`krb_rd_err( )` unpacks a received `krb_mk_err( )` message. *in* points to the beginning of the received message, whose length is specified in *in_length* . *code* is a pointer to a value to be filled in with the error value provided by the application. *msg_data* is a pointer to a MSG_DAT struct, defined in `<kerberos/krb.h>` . The routine fills in these MSG_DAT fields: the *app_data* field with a pointer to the application error text, *app_length* with the length of the *app_data* field, and `swap` with a 1 if the byte order of the receiver is different than that of the sender. (The application must still determine if it is appropriate to byte-swap application data; the Kerberos protocol fields are already taken care of).

The routine returns zero if the error message has been successfully received, or a Kerberos error code.

The KTEXT structure is used to pass around text of varying lengths. It consists of a buffer for the data, and a length. `krb_rd_req( )` takes an argument of this type containing the authenticator, and `krb_mk_req( )` returns the authenticator in a structure of this type. KTEXT itself is really a pointer to the structure. The actual structure is of type KTEXT_ST.

The AUTH_DAT structure is filled in by `krb_rd_req( )` . It must be allocated before calling `krb_rd_req( )` , and a pointer to it is passed. The structure is filled in with data obtained from Kerberos. The MSG_DAT structure is filled in by either `krb_rd_safe( )` or `krb_rd_err( )` . It must be allocated before the call and a pointer to it is passed. The structure is filled in with data obtained from Kerberos.

**FILES**

`/usr/lib/libkrb.*`
`/etc/aname`
`/etc/srvtab`
`/tmp/tkt`
*uid*

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Unsafe |

**SEE ALSO**

`kerberos`(1) , `kerberos_rpc`(3KRB) , `krb_realmofhost`(3KRB) , `krb_sendauth`(3KRB) , `krb_set_tkt_string`(3KRB) , `krb.conf`(4) , `krb.realms`(4) , `attributes`(5)

**NOTES**

These interfaces are unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

**BUGS**

The caller of `krb_rd_req( )` and `krb_rd_safe( )` must check time order and for replay attempts.

**AUTHORS**

Clifford Neuman, MIT Project Athena Steve Miller, MIT Project Athena / Digital Equipment Corporation

**RESTRICTIONS**

COPYRIGHT 1985,1986,1989 Massachusetts Institute of Technology

**NAME**

kerberos_rpc, authkerb_getucred, authkerb_seccreate, svc_kerb_reg – library routines for remote procedure calls using Kerberos authentication

**SYNOPSIS**

`cc` [ *flag* … ] *file* … `–lkrb` [ *library* … ]
#include <rpc/rpc.h>
#include <sys/types.h>

int **`authkerb_getucred`**(const struct svc_req **rqst*, uid_t **uidp*, gid_t **gidp*, short **gidlenp*, int *gidlist* [NGROUPS]);

AUTH ***`authkerb_seccreate`**(const char **service*, const char **srv_inst*, const char **realm*, const uint_t *window*, const char **timehost*, int **status*);

int **`svc_kerb_reg`**(const SVCXPRT **xprt*, const char **name*, const char **inst*, const char **realm*);

**DESCRIPTION**

RPC library routines allow C programs to make procedure calls on other machines across the network.

RPC supports various authentication flavors. Among them are:

| | |
|---|---|
| AUTH_NONE | (none) no authentication. |
| AUTH_SYS | Traditional UNIX-style authentication. |
| AUTH_DES | DES encryption-based authentication. |
| AUTH_KERB | Kerberos encryption-based authentication. |

The `authkerb_getucred( )` , `authkerb_seccreate( )` , and `svc_kerb_reg( )` routines implement the AUTH_KERB authentication flavor. The `kerbd` daemon (see `kerbd`(1M) ) must be running for the AUTH_KERB authentication system to work for kernel based services such as NFS, and `kinit`(1) must have been run by the user in all cases. Only the AUTH_KERB style of authentication is discussed here. For information about the AUTH_NONE and AUTH_SYS styles of authentication, refer to `rpc_clnt_auth`(3NSL) . For information about the AUTH_DES style of authentication, refer to `secure_rpc`(3NSL) .

**Routines**

See `rpc`(3NSL) for the definition of the AUTH data structure.

int authkerb_getucred(const struct svc_req **rqst* , uid_t **uidp* , gid_t **gidp* , short **gidlenp* , int *gidlist* [NGROUPS]);

`authkerb_getucred( )` is used on the server side for converting an AUTH_KERB credential received in an RPC request, which is operating system independent, into an AUTH_SYS credential. This routine returns `1` if it succeeds, `0` if it fails.

* *uidp* is set to the numerical ID of the user associated with the RPC request referenced by *rqst* . * *gidp* is set to the numerical ID of the user's group. The numerical IDs of the other groups to which the user belongs are stored in

*gidlist* []. * *gidlenp* is set to the number of valid group ID entries returned in *gidlist* []. All information returned by this routine is based on the Kerberos principal name contained in *rqst* . This principal name is taken to be the login name of the user, and the IDs returned are the same as if that user had physically logged in to the system.

AUTH *authkerb_seccreate(const char **service* , const char **srv_inst* , const char **realm* , const uint_t *window* , const char **timehost* , int **status* );

`authkerb_seccreate( )` is used on the client side to return an authentication handle that will enable the use of the Kerberos authentication system. The first parameter *service* is the Kerberos principal name of the service to be used. This name is generally a constant with respect to the service being used. *srv_instance* is the instance of the service to be called, and may be `NULL` to indicate any instance. *realm* is the Kerberos realm name of the desired service. If it is `NULL,` then the local default realm will be used.

The fourth parameter is the *window* on the validity of the client credential, given in seconds. If the difference in time between the client's clock and the server's clock exceeds *window* , the server will reject 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 of a window will increase the frequency of resynchronizations because of clock drift.

The fifth parameter, *timehost* , is optional. If it is `NULL,` then the authentication system will assume that the local clock is always in sync with the *timehost* clock, and will not attempt resynchronizations. If a timehost is supplied, however, then the system will consult with the remote time service whenever resynchronization is required. This parameter is usually the name of the host on which the server is running.

The final parameter *status* is also optional. If *status* is supplied, then it will be used to return a Kerberos error status codes if an error occurs. If *status* is `NULL,` then no detailed error codes will be returned.

If `authkerb_seccreate( )` fails, it returns `NULL.`

int svc_kerb_reg(const SVCXPRT **xprt* , const char **name* , const char **inst* , const char **realm* );

`svc_kerb_reg( )` performs registration tasks in the server which are required before AUTH_KERB requests can be processed. *xprt* is the RPC transport to which this information is to be associated. If *xprt* is `NULL` then this registration will be effective for any requests arriving on transports that have not been specifically registered. The service handles associated with connection endpoints are not exposed to the programmer. Consequently, *xprt* should be `NULL` for connection-oriented transports.

The other parameters describe the Kerberos principal identity that this server will take on. This must be the same identity that the clients will use when requesting Kerberos tickets for authentication. *name* is the principal name of the service and must be provided. *inst* is the instance. This parameter may be `NULL` to specify the `NULL` instance of the service. Most common would be for *inst* to be "*" which allows the Kerberos library to determine the correct instance to use, such as the hostname that the service is running on. *realm* is the Kerberos realm name to use in validating tickets. If it is `NULL,` then the local default realm will be used.

`svc_kerb_reg()` should generally be called immediately before `svc_run()` . It returns `0` if it succeeds, and `-1` if it fails.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Unsafe |

**SEE ALSO**

`kerberos`(1) , `kinit`(1) , `kerbd`(1M) , `rpc`(3NSL) , `rpc_clnt_auth`(3NSL) , `secure_rpc`(3NSL) , `svc_run`(3NSL) `attributes`(5)

**NOTES**

These interfaces are unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

## NAME

krb_realmofhost, krb_get_phost, krb_get_krbhst, krb_get_admhst, krb_get_lrealm – additional Kerberos utility routines

## SYNOPSIS

`cc` [ *flag* ... ] *file* ... `–lkrb` [ *library* ... ]

#include <kerberos/krb.h>

#include <netinet/in.h>

char ***krb_realmofhost**(const char **host*);

char ***krb_get_phost**(const char **alias*);

int **krb_get_krbhst**(char **host*, const char **realm*, const int *n*);

int **krb_get_admhst**(char **host*, const char **realm*, const int *n*);

int **krb_get_lrealm**(char **realm*, const int *n*);

## DESCRIPTION

`krb_realmofhost( )` returns the Kerberos realm of the host *host* , as determined by the translation table `/etc/krb.realms` . *host* should be the fully-qualified domain-style primary host name of the host in question. In order to prevent certain security attacks, this routine must either have a prior knowledge of a host's realm, or obtain such information securely.

The format of the translation file is described by `krb.realms`(4) . If *host* exactly matches a host_name line, the corresponding realm is returned. Otherwise, if the domain portion of *host* matches a domain_name line, the corresponding realm is returned. If *host* contains a domain, but no translation is found, *host* 's domain is converted to upper-case and returned. If *host* contains no discernible domain, or an error occurs, the local realm name, as supplied by `krb_get_lrealm( )` , is returned.

`krb_get_phost( )` converts the hostname `alias` (which can be either an official name or an alias) into the instance name to be used in obtaining Kerberos tickets for most services, including the Berkeley rcmd suite (rlogin, rcp, rsh). The current convention is to return the first segment of the official domain-style name after conversion to lower case.

`krb_get_krbhst( )` fills in *host* with the hostname of the *n* th host running a Kerberos key distribution center (KDC) for realm *realm* , as specified in the configuration file (`/etc/krb.conf` or `krb.conf` NIS map). The configuration format is described by `krb.conf`(4) . If the host is successfully filled in, the routine returns KSUCCESS. If the file (or NIS map) cannot be accessed, and *n* equals 1, then the hostname `kerberos` is filled in, and KSUCCESS is returned. If there are fewer than *n* hosts running a Kerberos KDC for the requested realm, or the configuration file is malformed, the routine returns KFAILURE.

When there is both a local `/etc/krb.conf` and a `krb.conf` NIS map, then the entries are counted starting first with the local file, then continuing with the NIS map. For example, if the local `/etc/krb.conf` file contains two entries which match *realm* , and the NIS map contains one matching entry, then there are three possible matches that `krb_get_krbhst( )` can return. The first two (for *n* values 1 and 2) come from the file, and the third (for *n* equal to 3) comes from the map.

`krb_get_admhst( )` fills in *host* with the hostname of the *n* th host running a Kerberos KDC database administration server for realm *realm* , as specified in `/etc/krb.conf` . If the file cannot be opened or is malformed, or there are fewer than *n* hosts running a Kerberos KDC database administration server, the routine returns KFAILURE.

The character arrays used as return values for `krb_get_krbhst( )` and `krb_get_admhst( )` should be large enough to hold any hostname.

`krb_get_lrealm( )` fills in *realm* with the *n* th realm of the local host, as specified in the configuration file. *realm* should be at least REALM_SZ (from `<kerberos/krb.h>` ) characters long. The return value is either KSUCCESS or KFAILURE.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Unsafe |

## SEE ALSO

`kerberos`(3KRB) , `krb.conf`(4) , `krb.realms`(4) , `attributes`(5)

## FILES

| | |
|---|---|
| `/etc/krb.realms` | translation file for host-to-realm mapping. |
| `/etc/krb.conf` | local realm-name and realm / server configuration file. |

## NOTES

These interfaces are unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

## BUGS

The current convention for instance names is too limited; the full domain name should be used.

`krb_get_lrealm( )` currently only supports *n* equal to 1. It should really consult the user’s ticket cache to determine the user’s current realm, rather than consulting a file on the host.

**NAME**

krb_sendauth, krb_recvauth, krb_net_write, krb_net_read – Kerberos routines for sending authentication via network stream sockets

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lkrb` [ *library* ... ]
#include <kerberos/krb.h>
#include <netinet/in.h>
int **krb_sendauth**(const long *options*, const int *fd*, KTEXT *ktext*, const char **service*, const char **inst*, const char **realm*, const ulong_t *checksum*, MSG_DAT **msg_data*, CREDENTIALS **cred*, Key_schedule *schedule*, const struct sockaddr_in **laddr*, const struct sockaddr_in **faddr*, const char **version*);

int **krb_recvauth**(const long *options*, const int *fd*, KTEXT *ktext*, const char **service*, char **inst*, const struct sockaddr_in **faddr*, const struct sockaddr_in **laddr*, AUTH_DAT **auth_data*, const char **filename*, Key_schedule *schedule*, char **version*);

int **krb_net_write**(const int *fd*, const char **buf*, const int *len*);

int **krb_net_read**(const int *fd*, char **buf*, const int *len*);

**DESCRIPTION**

These functions, which are built on top of the core Kerberos library, provide a convenient means for client and server programs to send authentication messages to one another through network connections.

The `krb_sendauth( )` function sends an authenticated ticket from the client program to the server program by writing the ticket to a network socket.

The `krb_recvauth( )` function receives the ticket from the client by reading from a network socket.

**krb_sendauth()**

This function writes the ticket to the network socket specified by the file descriptor *fd* , returning KSUCCESS if the write proceeds successfully, and an error code if it does not.

The *ktext* argument should point to an allocated KTEXT_ST structure. The *service* , *inst* , and *realm* arguments specify the server program's Kerberos principal name, instance, and realm. If you are writing a client that uses the local realm exclusively, you can set the *realm* argument to NULL.

The *version* argument allows the client program to pass an application-specific version string that the server program can then match against its own version string. The *version* string can be up to KSEND_VNO_LEN (see `<kerberos/krb.h>` ) characters in length.

The *checksum* argument can be used to pass checksum information to the server program. The client program is responsible for specifying this information. This checksum information is difficult to corrupt because `krb_sendauth( )` passes it over the network in encrypted form. The *checksum* argument is passed as the checksum argument to `krb_mk_req( )` (see `kerberos`(3KRB) ).

You can set `krb_sendauth( )` 's other arguments to NULL unless you want the client and server programs to mutually authenticate themselves. In the case of mutual authentication, the client authenticates itself to the server program, and demands that the server in turn authenticate itself to the client.

**krb_sendauth() and Mutual Authentication**

If you want mutual authentication, make sure that you read all pending data from the local socket before calling `krb_sendauth( )` . Set `krb_sendauth( )` 's *options* argument to KOPT_DO_MUTUAL (this macro is defined in `<kerberos/krb.h>` ); make sure that the *laddr* argument points to the address of the local socket, and that *faddr* points to the foreign socket's network address.

`krb_sendauth( )` fills in the other arguments – *msg_data* , *cred* , and *schedule* – before sending the ticket to the server program. You must, however, allocate space for these arguments before calling the function.

`krb_sendauth( )` supports two other options: KOPT_DONT_MK_REQ and KOPT_DONT_CANON. If called with *options* set as KOPT_DONT_MK_REQ, `krb_sendauth( )` will not use the `krb_mk_req( )` (see `kerberos`(3KRB) ) function to retrieve the ticket from the Kerberos server. The *ktext* argument must point to an existing ticket and authenticator (such as would be created by `krb_mk_req( )` ), and the *service* , *inst* , and *realm* arguments can be set to NULL.

If called with *options* set as KOPT_DONT_CANON, `krb_sendauth( )` will not convert the service's instance to canonical form using `krb_get_phost( )` (see `krb_realmofhost`(3KRB) ).

If you want to call `krb_sendauth( )` with a multiple *options* specification, construct *options* as a bitwise-OR of the options you want to specify.

**krb_recvauth()**

The `krb_recvauth( )` function reads a ticket/authenticator pair from the socket pointed to by the *fd* argument. Set the *options* argument as a bitwise-OR of the options desired. Currently only KOPT_DO_MUTUAL is useful to the receiver.

The *ktext* argument should point to an allocated KTEXT_ST structure. `krb_recvauth( )` fills *ktext* with the ticket/authenticator pair read from *fd* , then passes it to `krb_rd_req( )` (see `kerberos`(3KRB) ).

The *service* and *inst* arguments specify the expected service and instance for which the ticket was generated. They are also passed to `krb_rd_req( )` (see `kerberos`(3KRB) ). The *inst* argument may be set to "*" if the caller wishes `krb_mk_req( )` (see `kerberos`(3KRB) ) to fill in the instance used (note that there must be space in the *inst* argument to hold a full instance name, see `krb_mk_req( )` on `kerberos`(3KRB) ).

The *faddr* argument should point to the address of the peer which is presenting the ticket. It is also passed to `krb_rd_req( )` (see `kerberos`(3KRB) ).

If the client and server plan to mutually authenticate one another, the *laddr* argument should point to the local address of the file descriptor. Otherwise you can set this argument to NULL.

The *auth_data* argument should point to an allocated AUTH_DAT area. It is passed to and filled in by `krb_rd_req( )` (see `kerberos`(3KRB) ). The checksum passed to the corresponding `krb_sendauth( )` is available as part of the filled-in AUTH_DAT area.

The *filename* argument specifies the filename which the service program should use to obtain its service key. `krb_recvauth( )` passes *filename* to the `krb_rd_req( )` function, see `kerberos`(3KRB) , If you set this argument to "", `krb_rd_req( )` looks for the service key in the file `/etc/srvtab` .

If the client and server are performing mutual authentication, the *schedule* argument should point to an allocated Key_schedule. Otherwise it is ignored and may be NULL.

The *version* argument should point to a character array of at least KSEND_VNO_LEN characters. It is filled in with the version string passed by the client to `krb_sendauth( )` .

**krb_net_write() and krb_net_read()**

The `krb_net_write( )` function emulates the `write`(2) system call, but guarantees that all data specified is written to *fd* before returning, unless an error condition occurs.

The `krb_net_read( )` function emulates the `read`(2) system call, but guarantees that the requested amount of data is read from *fd* before returning, unless an error condition occurs.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Unsafe |

## SEE ALSO

`read`(2) , `write`(2) , `kerberos`(3KRB) , `kerberos_rpc`(3KRB) , `krb_realmofhost`(3KRB) , `attributes` (5)

## NOTES

These interfaces are unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

## BUGS

`krb_sendauth( )` , `krb_recvauth( )` , `krb_net_write( )` , and `krb_net_read( )` will not work properly on sockets set to non-blocking I/O mode.

## AUTHOR

John T. Kohl, MIT Project Athena

## RESTRICTIONS

Copyright 1988, Massachusetts Institute of Technology. For copying and distribution information, please see the header `<kerberos/mit-copyright.h>`.

**NAME**

krb_set_tkt_string – set Kerberos ticket cache file name

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lkrb` [ *library* ... ]
#include <kerberos/krb.h>

void **`krb_set_tkt_string`**(const char **filename*);

**DESCRIPTION**

`krb_set_tkt_string( )` sets the name of the file that holds the user's cache of Kerberos server tickets and associated session keys.

The string *filename* passed in is copied into local storage. Only MAXPATHLEN-1 (see `<sys/param.h>`) characters of the filename are copied in for use as the cache file name.

This routine should be called during initialization, before other Kerberos routines are called; otherwise the routines which fetch the ticket cache file name may be called and return an undesired ticket file name until this routine is called.

**FILES**

| | |
|---|---|
| `/tmp/tkt`*uid* | default ticket file name, unless the environment variable KRBTKFILE is set. *uid* denotes the user's uid, in decimal. |

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Unsafe |

**SEE ALSO**

`kerberos`(3KRB), `attributes`(5)

**NOTES**

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

**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 provides TCP / IP access to the X.500 Directory or to a stand-alone LDAP server. The SUNWlldap package includes various LDAP clients and an LDAP client library used to provide programmatic access to the LDAP protocol. This man page gives an overview of the LDAP library functions.

Both synchronous and asynchronous APIs are provided. Also included are various functions to parse the results returned from these functions. These functions are found in the `libldap.so.3` shared object.

The basic interaction is as follows. A connection is made to an LDAP server by calling `ldap_open`(3LDAP). An LDAP bind operation is performed by calling one of `ldap_bind`(3LDAP) and friends. Next, other operations are performed by calling one of the synchronous or asynchronous functions (for example, `ldap_search_s`(3LDAP) or `ldap_search`(3LDAP) followed by `ldap_result`(3LDAP)). Results returned from these functions are interpreted by calling the LDAP parsing functions. The LDAP association is terminated by calling `ldap_unbind`(3LDAP). Errors can be interpreted by calling `ldap_perror`(3LDAP). The `ldap_set_rebind_proc`(3LDAP) function can be used to set a function to be called back when an LDAP bind operation needs to occur when handling a client referral.

**Search Filters**

Search filters to be passed to the ldap search functions can be constructed by hand, or by calling the `ldap_getfilter`(3LDAP) functions.

**Displaying Results**

Results obtained from the ldap search functions can be output by hand, by calling `ldap_first_entry`(3LDAP) and `ldap_next_entry`(3LDAP) to step through the entries returned, `ldap_first_attribute`(3LDAP) and `ldap_next_attribute`(3LDAP) to step through an entry's attributes, and `ldap_get_values`(3LDAP) to retrieve a given attribute's value, and then calling `printf`(3C) or whatever to display the values.

Alternatively, the entry can be output automatically by calling the `ldap_entry2text`(3LDAP), `ldap_entry2text_search`(3LDAP), `ldap_entry2html`(3LDAP), or `ldap_entry2html_search`(3LDAP) functions. These functions look up the object class of the entry they are passed in the `ldaptemplates.conf`(4) file to decide which attributes to display and how to display them. Output is handled via a function passed in as a parameter.

**Uniform Resource Locators (URLS)**

The `ldap_url`(3LDAP) functions can be used test a URL to see if 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 via LDAP. This scheme allows you to look up entries using fuzzy, untyped names like "mark smith, umich, us".

**Caching**

The `ldap_cache`(3LDAP) functions implement a local client caching scheme, providing a substantial performance increase for repeated queries.

## Utility Functions

Also provided are various utility functions. The `ldap_sort`(3LDAP) functions are used to sort the entries and values returned via the ldap search functions. The `ldap_friendly`(3LDAP) functions are used to map from short two letter country codes (or other strings) to longer "friendlier" names. The `ldap_charset`(3LDAP) functions can be used to translate to and from the T.61 character set used for many character strings in the LDAP protocol.

**Connectionless Access**

The `cldap_search_s`(3LDAP) function allows you to access the directory via Connectionless LDAP (CLDAP), which is similar to LDAP but operates over UDP, obviating the need to set up and tear down a connection by calling `ldap_open`(3LDAP), `ldap_bind`(3LDAP), and `ldap_unbind`(3LDAP). `cldap_open`(3LDAP) should be called before using `cldap_search_s`(3LDAP). All the same getfilter, parsing, and display that can be used with regular LDAP functions can be used with the CLDAP functions.

**BER Library**

Also included in the distribution is a set of lightweight Basic Encoding Rules functions. These functions are used by the LDAP library functions to encode and decode LDAP protocol elements using the (slightly simplified) Basic Encoding Rules defined by LDAP. They are not normally used directly by an LDAP application program. The functions provide a printf and scanf-like interface, as well as lower-level access.

**Index**

| | |
|---|---|
| `ldap_open`(3LDAP) | open a connection to an LDAP server |
| `ldap_init`(3LDAP) | initialize the LDAP library without opening a connection to a server |
| `ldap_result`(3LDAP) | wait for the result from an asynchronous operation |
| `ldap_abandon`(3LDAP) | abandon (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, return value and place message |
| `ldap_add_ext_s`(3LDAP) | synchronously add an entry, return value and place message |
| `ldap_bind`(3LDAP) | asynchronously bind to the directory |
| `ldap_bind_s`(3LDAP) | synchronously bind to the directory |
| `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 and close the connection |
| `ldap_unbind_s`(3LDAP) | equivalent to `ldap_unbind`(3LDAP) |
| `ldap_enable_cache`(3LDAP) | enable LDAP client caching |
| `ldap_disable_cache`(3LDAP) | disable LDAP client caching |
| `ldap_destroy_cache`(3LDAP) | disable LDAP client caching and destroy cache contents |
| `ldap_flush_cache`(3LDAP) | flush LDAP client cache |
| `ldap_uncache_entry`(3LDAP) | uncache requests pertaining to an entry |
| `ldap_uncache_request`(3LDAP) | uncache a request |
| `ldap_set_cache_options`(3LDAP) | set cache options |
| `ldap_compare`(3LDAP) | asynchronous compare to a directory entry |
| `ldap_compare_s`(3LDAP) | synchronous compare to a directory entry |
| `ldap_compare_ext`(3LDAP) | asynchronous compare to a directory entry, return value and place message |

| | |
|---|---|
| `ldap_compare_ext_s`(3LDAP) | synchronous compare to a directory entry, return value and place message |
| `ldap_control_free`(3LDAP) | LDAP control disposal |
| `ldap_controls_free`(3LDAP) | LDAP control disposal |
| `ldap_delete`(3LDAP) | asynchronously delete an entry |
| `ldap_delete_s`(3LDAP) | synchronously delete an entry |
| `ldap_delete_ext`(3LDAP) | asynchronously delete an entry, return value and place message |
| `ldap_delete_ext_s`(3LDAP) | synchronously delete an entry, return value and place |
| `ldap_init_templates`(3LDAP) | initialize display template functions from a file |
| `ldap_init_templates_buf`(3LDAP) | initialize display template functions from a buffer |
| `ldap_free_templates`(3LDAP) | free display template function memory |
| `ldap_first_reference`(3LDAP) | steps through `ldap_result`(3LDAP) message chain |
| `ldap_count_references`(3LDAP) | counts the messages in an `ldap_result`(3LDAP) message chain |
| `ldap_first_message`(3LDAP) | steps through `ldap_result`(3LDAP) message chain |
| `ldap_count_messages`(3LDAP) | counts the messages in an `ldap_result`(3LDAP) message chain |
| `ldap_next_message`(3LDAP) | steps through `ldap_result`(3LDAP) message chain |
| `ldap_msgtype`(3LDAP) | returns the type of LDAP message |
| `ldap_first_disptmpl`(3LDAP) | get first display template |

| | |
|---|---|
| `ldap_next_disptmpl`(3LDAP) | get next display template |
| `ldap_oc2template`(3LDAP) | return template appropriate for objectclass |
| `ldap_tmplattrs`(3LDAP) | return attributes needed by 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 using a display template |
| `ldap_entry2text_search`(3LDAP) | search for and display an entry as text using a display template |
| `ldap_vals2text`(3LDAP) | display values as text |
| `ldap_entry2html`(3LDAP) | display an entry as HTML (HyperText Markup Language) using a display template |
| `ldap_entry2html_search`(3LDAP) | search for and display an entry as HTML using a display template |
| `ldap_vals2html`(3LDAP) | display values as HTML |
| `ldap_perror`(3LDAP) | print an LDAP error indication to standard error |
| `ld_errno`(3LDAP) | LDAP error indication |
| `ldap_result2error`(3LDAP) | extract LDAP error indication from LDAP result |
| `ldap_errlist`(3LDAP) | list of ldap errors and their meanings |
| `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 (3N) |
| `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 values with lengths |
| `ldap_value_free`(3LDAP) | free memory allocated by l `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 (3N) |
| `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, place message |
| `ldap_modify_ext_s`(3LDAP) | synchronously modify an entry, return value, place message |
| `ldap_mods_free`(3LDAP) | free array of pointers to mod structures used by ldap_modify (3N) |
| `ldap_modrdn2`(3LDAP) | asynchronously modify the RDN of an entry |
| `ldap_modrdn2_s`(3LDAP) | synchronously modify the RDN of an entry |
| `ldap_modrdn`(3LDAP) | depreciated - use ldap_modrdn2 (3N) |
| `ldap_modrdn_s`(3LDAP) | depreciated - use ldap_modrdn2_s (3N) |
| `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 results allocated by ldap_result (3N) |
| `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, return value and place message |
| `ldap_search_ext_s`(3LDAP) | synchronously search the directory, return value and place message |
| `ldap_search_st`(3LDAP) | synchronously search the directory with timeout |
| `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 (3N) functions |
| `ldap_ufn_setprefix`(3LDAP) | set prefix used by ldap_ufn (3N) functions |
| `ldap_ufn_timeout`(3LDAP) | set timeout used by ldap_ufn (3N) 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_url_search`(3LDAP) | asynchronously search using an LDAP URL |
| `ldap_url_search_s`(3LDAP) | synchronously search using an LDAP URL |
| `ldap_url_search_st`(3LDAP) | synchronously search using an LDAP URL and a timeout |
| `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 |

| | |
|---|---|
| `cldap_open`(3LDAP) | open a connectionless LDAP (CLDAP) session |
| `cldap_search_s`(3LDAP) | perform a search using connectionless LDAP |
| `cldap_setretryinfo`(3LDAP) | set retry and timeout information using connectionless LDAP |
| `cldap_close`(3LDAP) | terminate a connectionless LDAP session |

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Availability | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

## 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 `–1`otherwise 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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

## SEE ALSO

`ldap(3N), ldap_result(3N), ldap_error(3N)`

## 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 (`LDAP_SUCCESS` if everything went ok, an error otherwise).

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Availability | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`ldap`(3LDAP) , `ldap_error`(3LDAP) , `ldap_modify`(3LDAP)

**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_set_rebind_proc – LDAP bind functions

**SYNOPSIS**

cc[ *flag*... ] *file*... -llldap[ *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*);

void **ldap_set_rebind_proc**(LDAP **ld*, int (**rebindproc*);

int **ldap_sasl_bind**(LDAP **ld*, char **dn*, char **mechanism*, struct berval **cred*, LDAPControl ***serverctrls*, LDAPControl ***clientctrls*, int **msgidp*);

int **ldap_sasl_bind_s**(LDAP **ld*, char **dn*, char **mechanism*, struct berval **cred*, LDAPControl ***serverctrls*, LDAPControl ***clientctrls*);

**DESCRIPTION**

These functions provide various interfaces to the LDAP bind operation. After a connection is made to an LDAP server using `ldap_open`(3LDAP) , an LDAP bind operation must be performed before other operations can be attempted over the conection. Both synchronous and asynchronous versions of each variant of the bind call are provided. There are three types of calls, providing simple authentication, kerberos authentication, and general functions to do either one. All functions take `ld` as their first parameter, as returned from `ldap_open`(3LDAP) .

**Simple Authentication**

The simplest form of the bind call is `ldap_simple_bind_s` (). It takes the DN to bind as in `who` , and the userPassword associated with the entry in `passwd` . It returns an LDAP error indication (see `ldap_error`(3LDAP) ). The `ldap_simple_bind( )` call is asynchronous, taking the same parameters but only initiating the bind operation and returning the message id of the request it 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 can be used when the authentication method to use needs to be selected at runtime. They both take an extra *method* parameter selecting the authentication method to use. It should be set to `LDAP_AUTH_SIMPLE` to select simple authentication. `ldap_bind( )`

returns the message id of the request it initiates. `ldap_bind_s( )` returns an LDAP error indication.

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 method to use, as a dotted-string representation of an OID identifying the method, and a struct `berval` holding the credentials. The special constant value `LDAP_SASL_SIMPLE` ("") can be passed to request simple authentication, or the simplified routines `ldap_simple_bind()` or `ldap_simple_bind_s()` can be use.

**Unbinding**

The `ldap_unbind( )` call is used to unbind from the directory, terminate the current association, and free the resources contained in the `ld` structure. Once it is called, the connection to the LDAP server is closed, and the `ld` structure is invalid. The `ldap_unbind_s( )` call is just another name for `ldap_unbind( )` ; both of these calls are synchronous in nature.

**Re-Binding While Following Referral**

The `ldap_set_rebind_proc( )` call is used to set a function that will be called back to obtain bind credentials used when a new server is contacted during the following of an LDAP referral. Note that this function is only available when the LDAP libraries are compiled with `LDAP_REFERRALS` defined and is only used when the ld_options field in the LDAP structure has `LDAP_OPT_REFERRALS` set (this is the default). If `ldap_set_rebind_proc( )` is never called, or if it is called with a NULL *rebindproc* parameter, an unauthenticated simple LDAP bind will always be done when chasing referrals.

*rebindproc* should be a function that is declared like this:

```
int rebindproc( LDAP *ld, char **whop, char **credp,
    int *methodp, int freeit );
```

The LDAP library will first call the rebindproc to obtain the referral bind credentials, and the *freeit* parameter will be zero. The *whop* , *credp* , and *methodp* should be set as appropriate. If the rebindproc returns `LDAP_SUCCESS` , referral processing continues, and the rebindproc will be called a second time with *freeit* non-zero to give your application a chance to free any memory allocated in the previous call.

If anything but `LDAP_SUCCESS` is returned by the first call to the rebindproc, then referral processing is stopped and that error code is returned for the original LDAP operation.

## RETURN VALUES

A call to `ldap_result`(3LDAP) , can be used to obtain the result of the bind operations.

## ERRORS

Asynchronous functions will return `-1` in case of error, setting the *ld_errno* parameter of the `ld` structure. Synchronous functions return whatever *ld_errno* is set to. See `ldap_error`(3LDAP) for more information. If no credentials are returned the result parameter is set to NULL.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Availability | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`ldap`(3LDAP) , `ldap_error`(3LDAP) , `ldap_open`(3LDAP)

**NAME**

ldap_cache, ldap_enable_cache, ldap_disable_cache, ldap_destroy_cache, ldap_flush_cache, ldap_uncache_entry, ldap_uncache_request, ldap_set_cache_options – LDAP client caching functions

**SYNOPSIS**

cc[ *flag*... ] *file*... -lldap[ *library*... ]

```
#include <lber.h>
#include <ldap.h>
ldap_enable_cache(LDAP *ld, long timeout, long maxmem);

void ldap_disable_cache(LDAP *ld);

void ldap_destroy_cache(LDAP *ld);

void ldap_flush_cache(LDAP *ld);

void ldap_uncache_entry(LDAP *ld, char *dn);

void ldap_uncache_request(LDAP *ld, int msgid);

void ldap_set_cache_options(LDAP *ld, unsigned long opts);
```

**DESCRIPTION**

These functions are used to control the behavior of client caching of `ldap_search`(3LDAP) , `cldap_search_s`(3LDAP) , and `ldap_compare`(3LDAP) operations. By default, the cache is disabled and no caching is done. Enabling the cache can greatly improve performance and reduce network bandwidth when a client DUA makes repeated requests.

`ldap_enable_cache()` should be called to turn on local caching or to change cache parameters (lifetime of cached requests and memory used). The `ld` parameter should be the result of a successful call to `ldap_open`(3LDAP) . The *timeout* is specified in seconds, and is used to decide how long to keep cached requests. The *maxmem* value is in bytes, and is used to set an upper bound on how memory the cache will use. You can specify 0 for *maxmem* to restrict the cache size by the *timeout* only. The first call to ldap_enable_cache creates the cache; subsequent calls re-enable the cache and set the timeout and memory values.

`ldap_disable_cache()` temporarily disables use of the cache (new requests are not cached and the cache is not checked when returning results). It does not delete the cache contents.

`ldap_destroy_cache()` turns off caching and completely removes the cache from memory.

`ldap_flush_cache()` deletes the cache contents, but does not effect it in any other way.

`ldap_uncache_entry( )` removes all requests that make reference to the distinguished name *dn* from the cache. It should be used, for example, after doing an `ldap_modify`(3LDAP) call involving *dn* .

`ldap_uncache_request( )` removes the request indicated by the LDAP request id *msgid* from the cache.

`ldap_set_cache_options( )` is used to change caching behavior. The current supported options are `LDAP_CACHE_OPT_CACHENOERRS` to suppress caching of any requests that result in an error, and `LDAP_CACHE_OPT_CACHEALLERRS` to enable caching of all requests. The default behavior is to not cache requests that result in errors, except that request that result in the error `LDAP_SIZELIMIT_EXCEEDED` are cached.

**ERRORS**

`ldap_enable_cache( )` returns `0` upon success, and `-1` if it is unable to allocate space for the cache. All the other calls are declared as void and return nothing.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Availability | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`ldap`(3LDAP) , `ldap_search`(3LDAP) , `ldap_compare`(3LDAP) , `cldap_search_s`(3LDAP)

**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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

## SEE ALSO

`ldap`(3LDAP)

**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 | SUNWlldap (32-bit) |

| | |
|---|---|
| | SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO** `ldap`(3LDAP) , `ldap_error`(3LDAP)

**BUGS** There is no way to compare binary values but there should be.

**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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`ldap`(3LDAP) , `ldap_error`(3LDAP)

**NAME**

ldap_disptmpl, ldap_init_templates, ldap_init_templates_buf, ldap_free_templates, ldap_first_disptmpl, ldap_next_disptmpl, ldap_oc2template, 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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO** `ldap`(3LDAP) , `ldap_entry2text`(3LDAP) , `ldaptemplates.conf`(4)

**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 long*syntaxid*, 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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`ldap`(3LDAP) , `ldap_disptmpl`(3LDAP) , `ldaptemplates.conf`(4)

## NAME

ldap_error, ldap_perror, ldap_result2error, ldap_errlist, ldap_err2string – LDAP protocol error handling functions

## SYNOPSIS

cc[ *flag*... ] *file*... -lldap[ *library*... ]

#include <lber.h>
#include <ldap.h>
struct **ldap_error**(int *e_code*, char **e_reason*);

struct ldaperror ldap_errlist[];

char ***ldap_err2string**(int *err*);

void **ldap_perror**(LDAP **ld*, char **s*);

int **ldap_result2error**(LDAP **ld*, LDAPMessage **res*, int *freeit*);

## DESCRIPTION

These functions provide interpretation of the various error codes returned by the LDAP protocol and LDAP library functions and assigned to an error field in the `ld` structure. `ldap_perror( )` and `ldap_result2error( )` functions are deprecated for all new development; `ldap_err2string( )` should be used instead.

The `ldap_result2error( )` function takes *res* , a result as produced by `ldap_result`(3LDAP) or other synchronous LDAP calls, and returns the corresponding error code. Possible error codes are listed below. If the *freeit* parameter is non zero it indicates that the *res* parameter should be freed by a call to `ldap_msgfree`(3LDAP) after the error code has been extracted. The error field in `ld` is set and returned.

The returned value can be passed to `ldap_err2string( )` or looked up in `ldap_errlist[]` to get a text description of the message. The string returned from `ldap_err2string( )` is a pointer to a static area that should not be modified. The last element in the `ldap_errlist[]` array is signaled by an error code of `-1` .

The `ldap_perror( )` function can be called to print an indication of the error on standard error, similar to the way `perror`(3C) works.

## 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` | Partial results only 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` | Filter type 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 can’t contact the LDAP server. |
| `LDAP_LOCAL_ERROR` | Some local error occurred. This is usually a failed malloc. |
| `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 timelimit was exceeded while waiting for a result. |
| `LDAP_AUTH_UNKNOWN` | The authentication method specified to ldap_bind() is not known. |
| `LDAP_FILTER_ERROR` | An invalid filter was supplied to ldap_search() (for example, unbalanced parentheses). |
| `LDAP_PARAM_ERROR` | An ldap function was called with a bad parameter (for example, a NULL ld pointer, etc.). |
| `LDAP_NO_MEMORY` | An memory allocation (for example, malloc(3N)) call failed in an ldap library function. |

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Availability | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`attributes`(5) , `ldap`(3LDAP) , `perror`(3C)

**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( ) and ldap_next_attribute( ) functions are used to step through the attributes in an LDAP entry. ldap_first_attribute( ) takes an *entry* as returned by ldap_first_entry(3LDAP) or ldap_next_entry(3LDAP) and returns a pointer to a per-connection buffer containing the first attribute type in the entry. The return value should be treated as if it is a pointer to a static area (that is, strdup(3C) it if you want to save it).

It also returns, in *berptr* , a pointer to a BerElement it has allocated to keep track of its current position. This pointer should be passed to subsequent calls to ldap_next_attribute( ) and is used used to effectively step through the entry's attributes. This pointer is freed by ldap_next_attribute( ) when there are no more attributes (that is, when ldap_next_attribute( ) returns NULL ). Otherwise, the caller is responsible for freeing the BerElement pointed to by *berptr* when it is no longer needed by calling ber_free(3LDAP) . When calling ber_free(3LDAP) in this instance, be sure the second argument is '0'.

The attribute names returned are suitable for inclusion in a call to ldap_get_values(3LDAP) to retrieve the attribute's values.

**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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

ldap(3LDAP) , ldap_first_entry(3LDAP) , ldap_get_values(3LDAP) , ldap_error(3LDAP)

**NOTES**

The ldap_first_attribute( ) function mallocs memory that may need to be freed by the caller via ber_free(3LDAP) .

## NAME

ldap_first_entry, ldap_next_entry, ldap_count_entries, ldap_count_references, ldap_first_reference, ldap_first_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*);

**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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`ldap`(3LDAP) , `ldap_result`(3LDAP) , `ldap_search`(3LDAP) , `ldap_first_attribute`(3LDAP) , `ldap_get_values`(3LDAP) , `ldap_get_dn`(3LDAP)

## 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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

## 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\
", 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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`ldap`(3LDAP)

**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 via `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 via 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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**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_getfilter, ldap_init_getfilter, ldap_init_getfilter_buf, ldap_getfilter_free, ldap_getfirstfilter, ldap_getnextfilter, 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*);

**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 expession *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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`ldap`(3LDAP) , `ldapfilter.conf`(4)

**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_option, ldap_set_option – get / 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 access to session preferences to an LDAP structure. `ldap_get_option( )` gets session preferences from the LDAP structure. `ldap_set_option( )` sets session preferences in the LDAP structure.

*ld* is the connection handle, which is a pointer to an LDAP structure containing information about the connection to the LDAP server. *option* is the name of the option to be read or modified. *optdata* is a pointer to the value of the option that you want to set / get.

The *option* parameter can have one of the values listed in the following section.

## PARAMETERS

The following are the values for the *option* parameter:

`LDAP_OPT_API_INFO`
Used to retrieve some 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 *)` . *optdata* can be one of the following values:

| | |
|---|---|
| `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`

Maximum number of entries that should be 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`
Maximum number of seconds that should be 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 or not the client should follow referrals. The data type for the *optdata* parameter is `(int *)` . *optdata* can be one of the following values:

| | |
|---|---|
| `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 they abort prematurely. It *may* be set to one of the constants `LDAP_OPT_ON` or `LDAP_OPT_OFF` .

`LDAP_OPT_PROTOCOL_VERSION`
Version of the protocol supported by your client. The data type for the *optdata* parameter is `(int *)` . You can specify either `LDAP_VERSION2` or `LDAP_VERSION3` . If no version is set, the default is `LDAP_VERSION2` . In order to use LDAP v3 features, you need to set the protocol version to `LDAP_VERSION3` .

`LDAP_OPT_SERVER_CONTROLS`
Pointer to an array of `LDAPControl` structures representing the LDAP v3 server controls you want sent with every request by default. The data type for the *optdata* parameter for `ldap_set_option( )` is `(LDAPControl **)` and for `ldap_get_option( )` is `(LDAPControl ***)` .

`LDAP_OPT_CLIENT_CONTROLS`
Pointer to an array of `LDAPControl` structures representing the LDAP v3 client controls you want sent with every request by default. The data type for the *optdata* parameter for `ldap_set_option( )` is `(LDAPControl **)` and for `ldap_get_option( )` is `(LDAPControl ***)` .

`LDAP_OPT_API_FEATURE_INFO`
: Used to retrieve version information about LDAP API extended features at execution time. The data type for the *optdata* parameter is `(LDAPAPIFeatureInfo *)` . This option is READ-ONLY and cannot be set.

`LDAP_OPT_HOST_NAME`
: This option sets the host name (or list of hosts) for the primary LDAP server. The data type for the *optdata* parameter for `ldap_set_option( )` is `(char *)` , and for `ldap_get_option( )` is `(char **)` .

`LDAP_OPT_ERROR_NUMBER`
: 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`
: 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`
: 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`
: Lets you set the last argument passed to the routine specified by `LDAP_OPT_REBIND_FN` . You can also set this option by calling the `ldap_set_rebind_proc( )` function. The data type for the *optdata* parameter is `(void * )` .

`LDAP_OPT_REBIND_FN`
: Lets you set the routine to be called when you need to authenticate a connection with another LDAP server (for example, during the course of a referral). You can also set this option by calling the `ldap_set_rebind_proc( )` function. The data type for the *optdata* parameter is `(LDAP_REBINDPROC_CALLBACK *)` .

**RETURN VALUES**

The `ldap_set_option( )` and `ldap_get_option( )` functions return:

| | |
|---|---|
| `LDAP_SUCCESS` | If successful |
| `--1` | If unsuccessful |

**ERRORS**

Upon successful completion, both functions return `LDAP_SUCCESS` , otherwise `-1` is returned.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Availability | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO** ldap_init(3LDAP) , attributes(5)

**NOTES** There are other elements in the LDAP structure that you should not change. You should not make any assumptions 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*);

**ldap_count_values**(char ***vals*);

**ldap_count_values_len**(struct berval ***vals*);

**ldap_value_free**(char ***vals*);

**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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`ldap`(3LDAP) , `ldap_first_entry`(3LDAP) , `ldap_first_attribute`(3LDAP) , `ldap_error`(3LDAP)

**NOTES**

These functions allocates memory that the caller must free.

**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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`attributes`(5) , `ldap`(3LDAP) , `ldap_add`(3LDAP) , `ldap_error`(3LDAP) , `ldap_get_option`(3LDAP)

**NAME**

ldap_modrdn, ldap_modrdn_s, ldap_modrdn2, ldap_modrdn2_s, ldap_rename, ldap_rename_s – modify LDAP entry RDN

**SYNOPSIS**

cc[ *flag*... ] *file*... -lldap[ *library*... ]

#include <lber.h>
#include <ldap.h>
int **ldap_modrdn**(LDAP ***ld*, char ***dn*, char ***newrdn*);

int **ldap_modrdn_s**(LDAP ***ld*, char ***dn*, char ***newrdn*, int *deleteoldrdn*);

int **ldap_modrdn2**(LDAP ***ld*, char ***dn*, char ***newrdn*, int *deleteoldrdn*);

int **ldap_modrdn2_s**(LDAP ***ld*, char ***dn*, char ***newrdn*, int *deleteoldrdn*);

int **ldap_rename**(LDAP **ld*, char **dn*, char **newrdn*, char **newparent*, int *deleteoldrdn*, LDAPControl ***serverctrls*, LDAPControl ***clientctrls*, int **msgidp*);

int **ldap_rename_s**(LDAP **ld*, char **dn*, char **newrdn*, char **newparent*, 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 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, returning the message id of the operation it initiates. `ldap_modrdn_s()` is synchronous, returning the LDAP error code indicating the success or failure of the operation. Use of these functions is deprecated. Use the versions described below instead.

The `ldap_modrdn2()` and `ldap_modrdn2_s()` functions also perform an LDAP modify RDN operation, taking the same parameters as above. In addition, they both take the *deleteoldrdn* parameter which is used as a boolean value to indicate whether the old RDN values should be deleted from the entry or not.

The ldap_modrdn_s() routine is deprecated and the `ldap_rename()` and `ldap_rename_s()` routines are used instead.

The `ldap_rename()` , `ldap_rename_s()` routines are used to change the name, that is, the rdn of an entry. These routines deprecate `ldap_modrdn()` and `ldap_modrdn_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 case of trouble, 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:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Availability | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

## SEE ALSO

`ldap`(3LDAP) , `ldap_error`(3LDAP)

**NAME**

ldap_open, ldap_init – initialize the LDAP library and open a connection to an LDAP server

**SYNOPSIS**

cc[ *flag*... ] *file*... -lldap[ *library*... ]

#include <lber.h>
#include <ldap.h>
LDAP ***ldap_open**(char **host*, int *port*);

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

**DESCRIPTION**

`ldap_open( )` opens a connection to an LDAP server and allocates an LDAP structure which is used to identify the connection and to maintain per-connection information. `ldap_init( )` allocates an LDAP structure but does not open an initial connection. The `ldap_open( )` function is deprecated and should no longer be used. `ldap_init( )` must be called before any operations are attempted.

`ldap_open( )` takes *host* , the hostname on which the LDAP server is running, and *port* , the port number to which to connect. If the default IANA -assigned port of 389 is desired, `LDAP_PORT` should be specified for *port* . The *host* parameter may contain a blank-separated list of hosts to try to connect to, and each host may optionally by of the form *host:port* . If present, the *:port* overrides the *port* parameter to `ldap_open( )` . Upon successfully making a connection to an LDAP server, `ldap_open( )` returns a pointer to an LDAP structure (opaque structure), which should be passed to subsequent calls to `ldap_bind( )` , `ldap_search( )` , and so forth. Certain fields in the LDAP structure can be set using `ldap_set_option( )` . See `ldap_set_option`(3LDAP) for more details.

`ldap_init( )` acts just like `ldap_open( )` , but does not open a connection to the LDAP server. The actual connection open will occur when the first operation is attempted.

**OPTIONS**

Options that affect a particular LDAP instance may be set by calling `ldap_set_option( )` . The settings of these options can be retrieved by calling `ldap_get_option( )` .

The other supported option is `LDAP_OPT_RESTART` , which if set will cause the LDAP library to restart the `select`(1) system call when it is interrupted by the system (that is `errno` is set to `EINTR` ). This option is not supported on the Macintosh and under MS-DOS.

An option can be turned off by clearing the appropriate bit in the `ld_options` field.

**ERRORS**

If an error occurs, these functions will return `NULL` and `errno` should be set appropriately.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Availability | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`select`(1) , `errno`(3C) , `ldap`(3LDAP) , `ldap_bind`(3LDAP) , `ldap_option`(3LDAP) , `attributes`(5)

**NOTES**

There are other elements in the LDAP structure that you should not change. You should not make any assumptions about the order of elements in the LDAP structure.

## 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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

## 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) , `ldap_modify`(3LDAP) , etc.). 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. NU To effect a poll, the timeout argument should be a non-null pointer, pointing to a zero-valued timeval structure. See `select`(1) 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 `-` , search entries will be returned one at a time as they come in, via separate calls to `ldap_result( )` . If it is set to `-1` , 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**

`ldap_result( )` returns `-1` if something bad happens, and zero if the timeout specified was exceeded.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Availability | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`select`(1) , `ldap`(3LDAP) , `ldap_search`(3LDAP)

**NOTES**

This function allocates memory for results that it receives. The memory can be freed by calling `ldap_msgfree` .

**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*);

**DESCRIPTION**

These functions are used to perform LDAP search operations. `ldap_search_s()` does the search synchronously (that is, not returning until the operation completes). `ldap_search_st()` does the same, but allows a *timeout* to be specified. `ldap_search()` is the asynchronous version, initiating the search and returning the message id of the operation it initiated.

*Base* is the DN of the entry at which to start the search. *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.

*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=*" will find any entries that have a mail attribute. The filter "mail=*@terminator.rs.itd.umich.edu" will find any entries that have a mail attribute ending in the specified string. To put parentheses in a filter, escape them with a backslash '\\' character. See RFC 1588 for a more complete description of allowable filters. See `ldap_getfilter`(3LDAP) for functions to help in constructing search filters automatically.

*Attrs* is a null-terminated array of attribute types to return from entries that match *filter* . If NULL is specified, all attributes will be returned. *Attrsonly* should be set to `1` if only attribute types are wanted. It should be set to `0` if both attributes types and attribute values are wanted.

The `ldap_search_ext()` function initiates an asynchronous search operation and returns `LDAP_SUCCESS` if the request was successfully sent to the server, or else it returns a 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 as such returns the result of the operation itself.

**ERRORS**

`ldap_search_s()` and `ldap_search_st()` will return the LDAP error code resulting from the search operation. See `ldap_error`(3LDAP) for details.

`ldap_search()` returns `-1` when terminating unsuccessfully.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Availability | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`ldap`(3LDAP) , `ldap_result`(3LDAP) , `ldap_getfilter`(3LDAP) , `ldap_error`(3LDAP)

**NOTES**

Note that both read and list functionality are subsumed by these functions, by using a filter like "objectclass=*" and a scope of `LDAP_SCOPE_BASE` (to emulate read) or `LDAP_SCOPE_ONELEVEL` (to emulate list).

These 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 routeines

## 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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

## SEE ALSO

`ldap`(3LDAP) , `ldapsearchprefs.conf`(4)

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>
**ldap_sort_entries**(LDAP **ld*, LDAPMessage ***chain*, char **attr*, int (**cmp* )());

**ldap_sort_values**(LDAP **ld*, char ***vals*, int (**cmp* )());

**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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`ldap`(3LDAP) , `ldap_search`(3LDAP) , `ldap_result`(3LDAP) , `qsort`(3C)

**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 | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

**SEE ALSO**

`gettimeofday`(3C) , `ldap`(3LDAP) , `ldap_search`(3LDAP) , `ldap_getfilter`(3LDAP) , `ldapfilter.conf`(4) , `ldap_error`(3LDAP)

**NOTES**

These functions may allocates memory. Return values are contained in `<ldap.h>` .

**NAME**

ldap_url, ldap_is_ldap_url, ldap_url_parse, 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*);

**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). LDAP URLs look like this:

`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) . |

Here is an example:

```
ldap://ldap.itd.umich.edu/c=US?o,description?one?o=umich
```
URLs that are wrapped in angle-brackets and / or preceded by "URL:" are also tolerated.

`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 (below) for values returned upon error.

`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 labeledURI 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_NOTLDAP` | URL doesn't begin with "ldap://". |
| `LDAP_URL_ERR_NODN` | URL has no DN (required). |
| `LDAP_URL_ERR_BADSCOPE` | URL scope string is invalid. |
| `LDAP_URL_ERR_MEM` | Can't allocate memory space. |

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Availability | SUNWlldap (32-bit)<br>SUNWldapx (64-bit) |
| Stability Level | Evolving |

## SEE ALSO

`ldap`(3LDAP) , `ldap_search`(3LDAP)

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**

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`(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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`accept`(3XNET), `connect`(3XNET), `socket`(3XNET), `attributes`(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

#include <netdir.h>

int **netdir_getbyname**(const struct netconfig **config*, const struct nd_hostserv **service*, struct nd_addrlist ***addrs*);

int **netdir_getbyaddr**(const struct netconfig **config*, struct nd_hostservlist ***service*, const struct netbuf **netaddr*);

void **netdir_free**(void **ptr*, const int *struct_type*);

int **netdir_options**(const struct netconfig **config*, const int *option*, const int *fildes*, char **point_to_args*);

char ***taddr2uaddr**(const struct netconfig **config*, const struct netbuf **addr*);

struct netbuf ***uaddr2taddr**(const struct netconfig **config*, const char **uaddr*);

void **netdir_perror**(char **s*);

char ***netdir_sperror**(void);

## DESCRIPTION

These routines 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()` routine 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 routine returns all addresses that are valid for that transport in the `nd_addrlist` structure. The `nd_hostserv` structure contains the following members:

```
char      /* host name */
*h_serv;  /* service name */
```

The `nd_addrlist` structure contains the following members:

```
int  n_cnt;       /* number of addresses */
struct netbuf *n_addrs;
```

`netdir_getbyname()` 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 will be 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()` routine with each `struct netconfig` structure returned by `getnetconfig`(3NSL) .

The `netdir_getbyaddr()` routine maps addresses to service names. This routine returns *service* , a list of host and service pairs that would yield this address. If more than one tuple of host and service name is returned, then 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 routines. *ptr* points to the structure that has to be freed. The `struct_type` identifies the structure:

```
struct netbuf            ND_ADDR
struct nd_addrlist       ND_ADDRLIST
struct hostserv          ND_HOSTSERV
struct nd_hostservlist   ND_HOSTSERVLIST
```

The universal address returned by `taddr2uaddr()` should be freed by `free` ().

The `netdir_options( )` routine 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( )` routine 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( )` routines support translation between universal addresses and TLI type `netbufs` . The `taddr2uaddr( )` routine 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 may not suppose a universal address form.

`uaddr2taddr( )` is the reverse of `taddr2uaddr( )` . 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 (i.e., the result of a `t_open` of `/dev/udp` ). *pointer_to_args* is not used. If this completes, broadcast operations may 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* will be bound to a reserved port. If *pointer_to_args* is a pointer to a `netbuf` structure, an attempt will be 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” (LIKE THE `0.0.0.0` ADDRESS THAT TCP USES) AND RETURN A “REAL ADDRESS” THAT CLIENT MACHINES CAN CONNECT TO. *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 something like `0.0.0.0.1.12` , and, if the call is successful, `m_uaddr` will be set to something like `192.11.109.89.1.12` . For most transports, `m_uaddr` is exactly what `s_uaddr` is.

**RETURN VALUES**

The `netdir_perror()` routine prints an error message on the standard output stating why one of the name-to-address mapping routines failed. The error message is preceded by the string given as an argument.

The `netdir_sperror()` routine returns a string containing an error message stating why one of the name-to-address mapping routines failed.

`netdir_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**

`gethostname`(3C) , `getnetconfig`(3NSL) , `getnetpath`(3NSL) , `netconfig`(4) , `attributes`(5)

## NAME

nis_error, nis_sperrno, nis_perror, nis_lerror, nis_sperror, nis_sperror_r – display NIS+ error messages

## SYNOPSIS

cc [ *flag* ... ] *file* ... –lnsl [ *library* ... ]
#include <rpcsvc/nis.h>
char ***nis_sperrno**(nis_error *status*);

void **nis_perror**(nis_error *status*, char **label*);

void **nis_lerror**(nis_error *status*, char **label*);

char ***nis_sperror_r**(nis_error *status*, char **label*, char **buf*, int *length*);

char ***nis_sperror**(nis_error *status*, char **label*);

## DESCRIPTION

These functions convert NIS+ status values into text strings.

nis_sperrno( ) simply returns a pointer to a string constant which is the error string.

nis_perror( ) prints the error message corresponding to *status* as "*label* : error message " on standard error.

nis_lerror( ) sends the error text to syslog(3C) at level LOG_ERR.

The function nis_sperror_r( ) , returns a pointer to a string that can be used or copied using the strdup( ) function (See string(3C) ). The caller must supply a string buffer, *buf* , large enough to hold the error string (a buffer size of 128 bytes is guaranteed to be sufficiently large). *status* and *label* are the same as for nis_perror( ) . The pointer returned by nis_sperror_r( ) is the same as *buf* , that is, the pointer returned by the function is a pointer to *buf* . *length* specifies the number of characters to copy from the error string to *buf* .

The last function, nis_sperror( ) , is similar to nis_sperror_r( ) except that the string is returned as a pointer to a buffer that is reused on each call. nis_sperror_r( ) is the preferred interface, since it is suitable for single-threaded and multi-threaded programs.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

## SEE ALSO

niserror(1) , string(3C) , syslog(3C) , attributes(5)

## NOTES

When compiling multithreaded applications, see Intro(3) , *Notes On Multithread Applications* , for information about the use of the _REENTRANT flag.

## NAME

nis_groups, nis_ismember, nis_addmember, nis_removemember, nis_creategroup, nis_destroygroup, nis_verifygroup, nis_print_group_entry – NIS+ group manipulation functions

## SYNOPSIS

`cc` [ *flag* ... ] *file* ... `–lnsl` [ *library* ... ]
#include <rpcsvc/nis.h>
bool_t **`nis_ismember`**(nis_name *principal*, nis_name *group*);

nis_error **`nis_addmember`**(nis_name *member*, nis_name *group*);

nis_error **`nis_removemember`**(nis_name *member*, nis_name *group*);

nis_error **`nis_creategroup`**(nis_name *group*, uint_t *flags*);

nis_error **`nis_destroygroup`**(nis_name *group*);

void **`nis_print_group_entry`**(nis_name *group*);

nis_error **`nis_verifygroup`**(nis_name *group*);

## DESCRIPTION

These functions manipulate NIS+ groups. They are used by NIS+ clients and servers, and are the interfaces to the group authorization object.

The names of NIS+ groups are syntactically similar to names of NIS+ objects but they occupy a separate namespace. A group named "a.b.c.d." is represented by a NIS+ group object named "a.groups_dir.b.c.d."; the functions described here all expect the name of the group, not the name of the corresponding group object.

There are three types of group members:

- An *explicit* member is just a NIS+ principal-name, for example "wickedwitch.west.oz."
- An *implicit* ("domain") member, written "*.west.oz.", means that all principals in the given domain belong to this member. No other forms of wildcarding are allowed: "wickedwitch.*.oz." is invalid, as is "wickedwitch.west.*.". Note that principals in subdomains of the given domain are *not* included.
- A *recursive* ("group") member, written "@cowards.oz.", refers to another group; all principals that belong to that group are considered to belong here.

Any member may be made *negative* by prefixing it with a minus sign ('-'). A group may thus contain explicit, implicit, recursive, negative explicit, negative implicit, and negative recursive members.

A principal is considered to belong to a group if it belongs to at least one non-negative group member of the group and belongs to no negative group members.

The `nis_ismember()` function returns TRUE if it can establish that *principal* belongs to *group* ; otherwise it returns FALSE.

The `nis_addmember( )` and `nis_removemember( )` functions add or remove a member. They do not check whether the member is valid. The user must have read and modify rights for the group in question.

The `nis_creategroup( )` and `nis_destroygroup( )` functions create and destroy group objects. The user must have create or destroy rights, respectively, for the *groups_dir* directory in the appropriate domain. The parameter *flags* to `nis_creategroup( )` is currently unused and should be set to zero.

The `nis_print_group_entry( )` function lists a group's members on the standard output.

The `nis_verifygroup( )` function returns `NIS_SUCCESS` if the given group exists, otherwise it returns an error code.

## EXAMPLES

**EXAMPLE 1** Simple Memberships

Given a group `sadsouls.oz.` with members `tinman.oz.` , `lion.oz.` , and `scarecrow.oz.` , the function call

```
bool_var = nis_ismember("lion.oz.", "sadsouls.oz.");
```

will return 1 (TRUE) and the function call

```
bool_var = nis_ismember("toto.oz.", "sadsouls.oz.");
```

will return 0 (FALSE).

**EXAMPLE 2** Implicit Memberships

Given a group `baddies.oz.` , with members `wickedwitch.west.oz.` and `*.monkeys.west.oz.` , the function callbool_var = nis_ismember("hogan.monkeys.west.oz.", "baddies.oz.");will return 1 (TRUE) because any principal from the `monkeys.west.oz.` domain belongs to the implicit group `*.monkeys.west.oz.` , but the function call

```
bool_var = nis_ismember("hogan.big.monkeys.west.oz.", "baddies.oz.");
```

will return 0 (FALSE).

**EXAMPLE 3** Recursive Memberships

Given a group `goodandbad.oz.` , with members `toto.kansas` , `@sadsouls.oz.` , and `@baddies.oz.` , and the groups `sadsouls.oz.` and `baddies.oz.` defined above, the function call

```
bool_var = nis_ismember("wickedwitch.west.oz.", "goodandbad.oz.");
```

will return 1 (TRUE), because `wickedwitch.west.oz.` is a member of the `baddies.oz.` group which is recursively included in the `goodandbad.oz.` group.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`nisgrpadm`(1) , `nis_objects`(3NSL) , `attributes`(5)

**NOTES**

These functions only accept fully-qualified NIS+ names.

A group is represented by a NIS+ object (see nis_objects(3NSL) ) with a variant part that is defined in the group_obj structure. It contains the following fields:

```
 uint_t gr_flags; /* Interpretation Flags
    (currently unused) */
 struct {
  uint_t gr_members_len;
  nis_name *gr_members_val;
 } gr_members;  /* Array of members */
```

NIS+ servers and clients maintain a local cache of expanded groups to enhance their performance when checking for group membership. Should the membership of a group change, servers and clients with that group cached will not see the change until either the group cache has expired or it is explicitly flushed. A server's cache may be flushed programmatically by calling the nis_servstate( ) function with tag TAG_GCACHE and a value of 1.

There are currently no known methods for nis_ismember( ) , nis_print_group_entry( ) , and nis_verifygroup( ) to get their answers from only the master server.

## NAME

nis_local_names, nis_local_directory, nis_local_host, nis_local_group, nis_local_principal – NIS+ local names

## SYNOPSIS

```
cc [ flag ... ] file ... –lnsl [ library ... ]
#include <rpcsvc/nis.h>
nis_name nis_local_directory(void);

nis_name nis_local_host(void);

nis_name nis_local_group(void);

nis_name nis_local_principal(void);
```

## DESCRIPTION

These functions return several default NIS+ names associated with the current process.

`nis_local_directory( )` returns the name of the NIS+ domain for this machine. This is currently the same as the Secure RPC domain returned by the `sysinfo`(2) system call.

`nis_local_host( )` returns the NIS+ name of the current machine. This is the fully qualified name for the host and is either the value returned by the `gethostname`(3C) function or, if the host name is only partially qualified, the concatenation of that value and the name of the NIS+ directory. Note that if a machine's name and address cannot be found in the local NIS+ directory, its hostname must be fully qualified.

`nis_local_group( )` returns the name of the current NIS+ group name. This is currently set by setting the environment variable `NIS_GROUP` to the groupname.

`nis_local_principal( )` returns the NIS+ principal name for the user associated with the effective UID of the calling process. This function maps the effective uid into a principal name by looking for a LOCAL type credential (see `nisaddcred`(1M) ) in the table named *cred.org_dir* in the default domain.

Note: The result returned by these routines is a pointer to a data structure with the NIS+ library, and should be considered a "read-only" result and should not be modified.

## ENVIRONMENT VARIABLES

| | |
|---|---|
| `NIS_GROUP` | This variable contains the name of the local NIS+ group. If the name is not fully qualified, the value returned by `nis_local_directory( )` will be concatenated to it. |

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`nisdefaults`(1) , `nisaddcred`(1M) , `sysinfo`(2) , `gethostname`(3C) , `nis_names`(3NSL) , `nis_objects`(3NSL) , `attributes`(5)

**NAME**

nis_names, nis_lookup, nis_add, nis_remove, nis_modify, nis_freeresult – NIS+ namespace functions

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lnsl` [ *library* ... ]
#include <rpcsvc/nis.h>
nis_result ***nis_lookup**(nis_name *name*, uint_t *flags*);

nis_result ***nis_add**(nis_name *name*, nis_object **obj*);

nis_result ***nis_remove**(nis_name *name*, nis_object **obj*);

nis_result ***nis_modify**(nis_name *name*, nis_object **obj*);

void **nis_freeresult**(nis_result **result*);

**DESCRIPTION**

These functions are used to locate and manipulate all NIS+ objects (see `nis_objects`(3NSL) ) except the NIS+ entry objects. To look up the NIS+ entry objects within a NIS+ table, refer to `nis_subr`(3NSL) .

`nis_lookup( )` resolves a NIS+ name and returns a copy of that object from a NIS+ server. `nis_add( )` and `nis_remove( )` add and remove objects to the NIS+ namespace, respectively. `nis_modify( )` can change specific attributes of an object that already exists in the namespace.

These functions should be used only with names that refer to an NIS+ Directory, NIS+ Table, NIS+ Group, or NIS+ Private object. If a name refers to an NIS+ entry object, the functions listed in `nis_subr`(3NSL) should be used.

`nis_freeresult( )` frees all memory associated with a `nis_result` structure. This function must be called to free the memory associated with a NIS+ result. `nis_lookup( )` , `nis_add(), nis_remove(),` and `nis_modify( )` all return a pointer to a `nis_result structure` which *must* be freed by calling `nis_freeresult( )` when you have finished using it. If one or more of the objects returned in the structure need to be retained, they can be copied with `nis_clone_object`(3NSL) (see `nis_subr`(3NSL) ).

`nis_lookup( )` takes two parameters, the name of the object to be resolved in *name* , and a flags parameter, *flags* , which is defined below. The object name is expected to correspond to the syntax of a non-indexed NIS+ name (see `nis_tables`(3NSL) ). The `nis_lookup( )` function is the *only* function from this group that can use a non-fully qualified name. If the parameter *name* is not a fully qualified name, then the flag `EXPAND_NAME` *must* be specified in the call. If this flag is not specified, the function will fail with the error NIS_BADNAME.

The *flags* parameter is constructed by logically ORing zero or more flags from the following list.

| | |
|---|---|
| `FOLLOW_LINKS` | When specified, the client library will “follow” links by issuing another NIS+ lookup call for the object named by the link. If the linked object is |

| | |
|---|---|
| | itself a link, then this process will iterate until the either a object is found that is not a *LINK* type object, or the library has followed 16 links. |
| `HARD_LOOKUP` | When specified, the client library will retry the lookup until it is answered by a server. Using this flag will cause the library to block until at least one NIS+ server is available. If the network connectivity is impaired, this can be a relatively long time. |
| `NO_CACHE` | When specified, the client library will bypass any object caches and will get the object from either the master NIS+ server or one of its replicas. |
| `MASTER_ONLY` | When specified, the client library will bypass any object caches and any domain replicas and fetch the object from the NIS+ master server for the object's domain. This insures that the object returned is up to date at the cost of a possible performance degradation and failure if the master server is unavailable or physically distant. |
| `EXPAND_NAME` | When specified, the client library will attempt to expand a partially qualified name by calling the function `nis_getnames( )` (see `nis_subr`(3NSL) ) which uses the environment variable `NIS_PATH` . |

The status value may be translated to ascii text using the function `nis_sperrno( )` (see `nis_error`(3NSL) ).

On return, the *objects* array in the result will contain one and possibly several objects that were resolved by the request. If the FOLLOW_LINKS flag was present, on success the function could return several entry objects if the link in question pointed within a table. If an error occurred when following a link, the objects array will contain a copy of the link object itself.

The function `nis_add( )` will take the object *obj* and add it to the NIS+ namespace with the name *name* . This operation will fail if the client making the request does not have the *create* access right for the domain in which this object will be added. The parameter *name* must contain a fully qualified NIS+ name. The object members *zo_name* and *zo_domain* will be constructed from this name. This operation will fail if the object already exists. This feature prevents the accidental addition of objects over another object that has been added by another process.

The function `nis_remove()` will remove the object with name *name* from the NIS+ namespace. The client making this request must have the *destroy* access right for the domain in which this object resides. If the named object is a link, the link is removed and *not* the object that it points to. If the parameter *obj* is not NULL, it is assumed to point to a copy of the object being removed. In this case, if the object on the server does not have the same object identifier as the object being passed, the operation will fail with the NIS_NOTSAMEOBJ error. This feature allows the client to insure that it is removing the desired object. The parameter *name* must contain a fully qualified NIS+ name.

The function `nis_modify()` will modify the object named by *name* to the field values in the object pointed to by *obj* . This object should contain a copy of the object from the name space that is being modified. This operation will fail with the error NIS_NOTSAMEOBJ if the object identifier of the passed object does not match that of the object being modified in the namespace.

Normally the contents of the member *zo_name* in the *nis_object* structure would be constructed from the name passed in the *name* parameter. However, if it is non-null the client library will use the name in the *zo_name* member to perform a rename operation on the object. This name *must not* contain any unquoted '.'(dot) characters. If these conditions are not met the operation will fail and return the NIS_BADNAME error code.

**Results**

These functions return a pointer to a structure of type `nis_result` :

```
struct nis_result {
 nis_error status;
 struct {
  uint_t objects_len;
  nis_object *objects_val;
 } objects;
 netobj cookie;
 uint32_t zticks;
 uint32_t dticks;
 uint32_t aticks;
 uint32_t cticks;
};
```

The *status* member contains the error status of the the operation. A text message that describes the error can be obtained by calling the function `nis_sperrno()` (see `nis_error`(3NSL) ).

The *objects* structure contains two members. *objects_val* is an array of *nis_object* structures; *objects_len* is the number of cells in the array. These objects will be freed by the call to `nis_freeresult()` . If you need to keep a copy of one or more objects, they can be copied with the function `nis_clone_object()` and freed with the function `nis_destroy_object()` (see `nis_server`(3NSL) ). Refer to `nis_objects`(3NSL) for a description of the `nis_object` structure.

The various ticks contain details of where the time was taken during a request. They can be used to tune one's data organization for faster access and to compare different database implementations.

| | |
|---|---|
| *zticks* | The time spent in the NIS+ service itself. This count starts when the server receives the request and stops when it sends the reply. |
| *dticks* | The time spent in the database backend. This time is measured from the time a database call starts, until the result is returned. If the request results in multiple calls to the database, this is the sum of all the time spent in those calls. |
| *aticks* | The time spent in any "accelerators" or caches. This includes the time required to locate the server needed to resolve the request. |
| *cticks* | The total time spent in the request. This clock starts when you enter the client library and stops when a result is returned. By subtracting the sum of the other ticks values from this value, you can obtain the local overhead of generating a NIS+ request. |

Subtracting the value in *dticks* from the value in *zticks* will yield the time spent in the service code itself. Subtracting the sum of the values in *zticks* and *aticks* from the value in *cticks* will yield the time spent in the client library itself. Note: all of the tick times are measured in microseconds.

**RETURN VALUES**

The client library can return a variety of error returns and diagnostics. The more salient ones are documented below.

| | |
|---|---|
| `NIS_SUCCESS` | The request was successful. |
| `NIS_S_SUCCESS` | The request was successful, however the object returned came from an object cache and not directly from the server. If you do not wish to see objects from object caches you must specify the flag `NO_CACHE` when you call the lookup function. |
| `NIS_NOTFOUND` | The named object does not exist in the namespace. |
| `NIS_CACHEEXPIRED` | The object returned came from an object cache taht has *expired* . The time to live value has gone to zero and the object may have changed. If the flag `NO_CACHE` was passed to |

| | |
|---|---|
| | the lookup function then the lookup function will retry the operation to get an unexpired copy of the object. |
| `NIS_NAMEUNREACHABLE` | A server for the directory of the named object could not be reached. This can occur when there is a network partition or all servers have crashed. See the `HARD_LOOKUP` flag. |
| `NIS_UNKNOWNOBJ` | The object returned is of an unknown type. |
| `NIS_TRYAGAIN` | The server connected to was too busy to handle your request. For the *add* , *remove* , and *modify* operations this is returned when either the master server for a directory is unavailable or it is in the process of checkpointing its database. It can also be returned when the server is updating it's internal state. And in the case of `nis_list()` if the client specifies a callback and the server does not have enough resources to handle the callback. |
| `NIS_SYSTEMERROR` | A generic system error occurred while attempting the request. Most commonly the server has crashed or the database has become corrupted. Check the syslog record for error messages from the server. |
| `NIS_NOT_ME` | A request was made to a server that does not serve the name in question. Normally this will not occur, however if you are not using the built in location mechanism for servers you may see this if your mechanism is broken. |
| `NIS_NOMEMORY` | Generally a fatal result. It means that the service ran out of heap space. |
| `NIS_NAMEEXISTS` | An attempt was made to add a name that already exists. To add the name, |

| | |
|---|---|
| | first remove the existing name and then add the new object or modify the existing named object. |
| `NIS_NOTMASTER` | An attempt was made to update the database on a replica server. |
| `NIS_INVALIDOBJ` | The object pointed to by *obj* is not a valid NIS+ object. |
| `NIS_BADNAME` | The name passed to the function is not a legal NIS+ name. |
| `NIS_LINKNAMEERROR` | The name passed resolved to a *LINK* type object and the contents of the link pointed to an invalid name. |
| `NIS_NOTSAMEOBJ` | An attempt to remove an object from the namespace was aborted because the object that would have been removed was not the same object that was passed in the request. |
| `NIS_NOSUCHNAME` | This hard error indicates that the named directory of the table object does not exist. This occurs when the server that should be the parent of the server that serves the table, does not know about the directory in which the table resides. |
| `NIS_NOSUCHTABLE` | The named table does not exist. |
| `NIS_MODFAIL` | The attempted modification failed. |
| `NIS_FOREIGNNS` | The name could not be completely resolved. When the name passed to the function would resolve in a namespace that is outside the NIS+ name tree, this error is returned with a NIS+ object of type `DIRECTORY` , which contains the type of namespace and contact information for a server within that namespace. |
| `NIS_RPCERROR` | This fatal error indicates the RPC subsystem failed in some way. Generally there will be a `syslog`(3C) |

message indicating why the RPC request failed.

**ENVIRONMENT VARIABLES**

NIS_PATH If the flag EXPAND_NAME is set, this variable is the search path used by `nis_lookup()` .

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`nis_error`(3NSL) , `nis_objects`(3NSL) , `nis_server`(3NSL) , `nis_subr`(3NSL) , `nis_tables`(3NSL) , `attributes`(5)

**NOTES**

You cannot modify the name of an object if that modification would cause the object to reside in a different domain.

You cannot modify the schema of a table object.

## NAME

nis_objects – NIS+ object formats

## SYNOPSIS

**cc** [ *flag* ... ] *file* ... –`lnsl` [ *library* ... ]

/usr/include/rpcsvc/nis_objects.x

## DESCRIPTION

### Common Attributes

The NIS+ service uses a variant record structure to hold the contents of the objects that are used by the NIS+ service. These objects all share a common structure which defines a set of attributes that all objects possess. The `nis_object` structure contains the following members:

```
typedef char *nis_name;
 struct nis_object {
  nis_oid  zo_oid;
  nis_name zo_name;
  nis_name zo_owner;
  nis_name zo_group;
  nis_name zo_domain;
  uint_t  zo_access;
  uint32_t  zo_ttl;
  objdata  zo_data;
        };
```

In this structure, the first member `zo_oid`, is a 64 bit number that uniquely identifies this instance of the object on this server. This member is filled in by the server when the object is created and changed by the server when the object is modified. When used in conjunction with the object's name and domain it uniquely identifies the object in the entire NIS+ namespace.

The second member, `zo_name`, contains the leaf name of the object. This name is *never* terminated with a '.' (dot). When an object is created or added to the namespace, the client library will automatically fill in this field and the domain name from the name that was passed to the function.

`zo_domain` contains the name of the NIS+ domain to which this object belongs. This information is useful when tracking the parentage of an object from a cache. When used in conjunction with the members `zo_name` and `zo_oid`, it uniquely identifies an object. This makes it possible to always reconstruct the name of an object by using the code fragment

```
  sprintf(buf,"%s.%s", obj⇒zo_name, obj⇒zo_domain);
```

The `zo_owner` and `zo_group` members contain the NIS+ names of the object's principal owner and group owner, respectively. Both names *must be* NIS+ fully qualified names. However, neither name can be used directly to identify the object they represent. This stems from the condition that NIS+ uses itself to store information that it exports.

The `zo_owner` member contains a fully qualified NIS+ name of the form *principal.domain.* This name is called a NIS+ principal name and is used to identify authentication information in a credential table. When the server constructs a search query of the form

```
[cname=principal],cred.org_dir.domain.
```

The query will return to the server credential information about *principal* for all flavors of RPC authentication that are in use by that principal. When an RPC request is made to the server, the authentication flavor is extracted from the request and is used to find out the NIS+ principal name of the client. For example, if the client is using the AUTH_DES authentication flavor, it will include in the authentication credentials the network name or *netname* of the user making the request. This netname will be of the form

```
unix.UID@domain
```

The NIS+ server will then construct a query on the credential database of the form

```
[auth_name=netname,auth_type=AUTH_DES],cred.org_dir.domain.
```

This query will return an entry which contains a principal name in the first column. This NIS+ principal name is used to control access to NIS+ objects.

The group owner for the object is treated differently. The group owner member is optional (it should be the null string if not present) but must be fully qualified if present. A group name takes the form

```
group.domain.
```

which the server then maps into a name of the form

```
group.groups_dir.domain.
```

The purpose of this mapping is to prevent NIS+ group names from conflicting with user specified domain or table names. For example, if a domain was called *engineering.foo.com.*, then without the mapping a NIS+ group of the same name to represent members of engineering would not be possible. The contents of groups are lists of NIS+ principal names which are used exactly like the `zo_owner` name in the object. See `nis_groups`(3NSL) for more details.

The `zo_access` member contains the bitmask of access rights assigned to this object. There are four access rights defined, and four are reserved for future use and must be zero. This group of 8 access rights can be granted to four categories of client. These categories are the object's owner, the object's group owner, all authenticated clients (world), and all unauthenticated clients (nobody). Note that access granted to "nobody" is really access granted to everyone, authenticated and unauthenticated clients.

The `zo_ttl` member contains the number of seconds that the object can "live" in a cache before it is expired. This value is called the time to live for this object. This number is particularly important on group and directory (domain) objects. When an object is cached, the current time is added to the value in `zo_ttl`. Then each time the cached object is used, the time in `zo_ttl` is compared with the current time. If the current time is later than the time in `zo_ttl` the object is said to have expired and the cached copy should not be used.

Setting the TTL is somewhat of an art. You can think of it as the "half life" of the object, or half the amount of time you believe will pass before the object changes. The benefit of setting the ttl to a large number is that the object will stay in a cache for long periods of time. The problem with setting it to a large value is that when the object changes it will take a long time for the caches to flush out old copies of that object. The problems and benefits are reversed for setting the time to a small value. Generally setting the value to 43200 (12 hrs) is reasonable for things that change day to day, and 3024000 is good for things that change week to week. Setting the value to 0 will prevent the object from ever being cached since it would expire immediately.

The `zo_data` member is a discriminated union with the following members:

```
zotypes zo_type;
 union {
  struct directory_obj di_data;
  struct group_obj gr_data;
  struct table_obj ta_data;
  struct entry_obj en_data;
  struct link_obj li_data;
  struct {
             uint_t po_data_len;
             char *po_data_val;
  } po_data;
        } objdata_u;
```

The union is discriminated based on the type value contained in `zo_type`. There six types of objects currently defined in the NIS+ service. These types are the directory, link, group, table, entry, and private types.

```
enum zotypes {
  BOGUS_OBJ = 0,
  NO_OBJ = 1,
  DIRECTORY_OBJ = 2,
  GROUP_OBJ = 3,
  TABLE_OBJ = 4,
  ENTRY_OBJ = 5,
  LINK_OBJ = 6,
  PRIVATE_OBJ = 7
 };
 typedef enum zotypes zotypes;
```

All object types define a structure that contains data specific to that type of object. The simplest are private objects which are defined to contain a variable length array of octets. Only the owner of the object is expected to understand the contents of a private object. The following section describe the other five object types in more significant detail.

**Directory Objects**

The first type of object is the *directory* object. This object's variant part is defined as follows:

```
enum nstype {
  UNKNOWN = 0,
  NIS = 1,
  SUNYP = 2,
  DNS = 4,
  X500 = 5,
  DNANS = 6,
  XCHS = 7,
 }
 typedef enum nstype nstype;
 struct oar_mask {
  uint_t oa_rights;
  zotypes oa_otype;
 }
 typedef struct oar_mask oar_mask;
 struct endpoint {
  char *uaddr;
  char *family;
  char *proto;
 }
 typedef struct endpoint endpoint;
 struct nis_server {
  nis_name name;
  struct {
   uint_t ep_len;
   endpoint *ep_val;
  } ep;
  uint_t key_type;
  netobj pkey;
 }
 typedef struct nis_server nis_server;
 struct directory_obj {
  nis_name  do_name;
  nstype  do_type;
  struct {
   uint_t do_servers_len;
   nis_server *do_servers_val;
  } do_servers;
  uint32_t do_ttl;
  struct {
   uint_t do_armask_len;
   oar_mask *do_armask_val;
  } do_armask;
 }
        typedef struct directory_obj directory_obj;
```

The main structure contains five primary members: `do_name`, `do_type`, `do_servers`, `do_ttl`, and `do_armask`. The information in the `do_servers` structure is sufficient for the client library to create a network connection with the named server for the directory.

The `do_name` member contains the name of the directory or domain represented in a format that is understandable by the type of nameservice serving that domain. In the case of NIS+ domains, this is the same as the name that can be composed using the `zo_name` and `zo_domain` members. For other name services, this name will be a name that they understand. For example, if this were a directory object describing an X.500 namespace that is "under" the NIS+ directory *eng.sun.com.*, this name might contain "/C=US, /O=Sun Microsystems, /OU=Engineering/". The type of nameservice that is being described is determined by the value of the member `do_type`.

The `do_servers` structure contains two members. `do_servers_val` is an array of *nis_server* structures; `do_servers_len` is the number of cells in the array. The *nis_server* structure is designed to contain enough information such that machines on the network providing name services can be contacted without having to use a name service. In the case of NIS+ servers, this information is the name of the machine in *name*, its public key for authentication in *pkey*, and a variable length array of endpoints, each of which describes the network endpoint for the `rpcbind` daemon on the named machine. The client library uses the addresses to contact the server using a transport that both the client and server can communicate on and then queries the `rpcbind` daemon to get the actual transport address that the server is using.

Note that the first server in the *do_servers* list is always the master server for the directory.

The *key_type* field describes the type of key stored in the *pkey* netobj (see `/usr/include/rpc/xdr.h` for a definition of the network object structure). Currently supported types are `NIS_PK_NONE` for no public key, `NIS_PK_DH` for a Diffie-Hellman type public key, and `NIS_PK_DHEXT` for an extended Diffie-Hellman public key.

The `do_ttl` member contains a copy of the `zo_ttl` member from the common attributes. This is the duplicated because the cache manager only caches the variant part of the directory object.

The `do_armask` structure contains two members. `do_armask_val` is an array of `oar_mask` structures; `do_armask_len` is the number of cells in the array. The `oar_mask` structure contains two members: `oa_rights` specifies the access rights allowed for objects of type `oa_otype`. These access rights are used for objects of the given type in the directory when they are present in this array.

The granting of access rights for objects contained within a directory is actually two-tiered. If the directory object itself grants a given access right (using the `zo_access` member in the `nis_object` structure representing the directory), then all objects within the directory are allowed that access. Otherwise, the `do_armask` structure is examined to see if the access is allowed specifically for that type of structure. This allows the administrator of a namespace to set separate policies for different object types, for example, one policy for the creation of tables and another policy for the creation of other directories. See `nis+`(1) for more details.

**Link Objects**

Link objects provide a means of providing *aliases* or symbolic links within the namespace. Their variant part is defined as follows.

```
struct link_obj {
 zotypes li_rtype;
 struct {
  uint_t li_attrs_len;
  nis_attr *li_attrs_val;
 } li_attrs;
 nis_name li_name;
}
```

The `li_rtype` member contains the object type of the object pointed to by the link. This is only a hint, since the object which the link points to may have changed or been removed. The fully qualified name of the object (table or otherwise) is specified in the member `li_name`.

NIS+ links can point to either other objects within the NIS+ namespace, or to entries within a NIS+ table. If the object pointed to by the link is a table and the member `li_attrs` has a nonzero number of attributes (index name / value pairs) specified, the table is searched when this link is followed. All entries which match the specified search pattern are returned. Note, that unless the flag FOLLOW_LINKS is specified, the nis_lookup(3NSL) function will always return non-entry objects.

**Group Objects**

Group objects contain a membership list of NIS+ principals. The group objects' variant part is defined as follows.

```
struct group_obj {
 uint_t gr_flags;
 struct {
  uint_t gr_members_len;
  nis_name *gr_members_val;
 } gr_members;
}
```

The gr_flags member contains flags that are currently unused. The gr_members structure contains the list of principals. For a complete description of how group objects are manipulated see nis_groups(3NSL).

**Table Objects**

The NIS+ table object is analogous to a YP map. The differences stem from the access controls, and the variable schemas that NIS+ allows. The table objects data structure is defined as follows:

```
#define TA_BINARY 1
#define TA_CRYPT 2
#define TA_XDR 4
#define TA_SEARCHABLE 8
#define TA_CASE 16
#define TA_MODIFIED 32
struct table_col {
 char *tc_name;
 uint_t tc_flags;
 uint_t tc_rights;
}
typedef struct table_col table_col;
struct table_obj {
 char *ta_type;
 uint_t ta_maxcol;
 uchar_t ta_sep;
 struct {
  uint_t ta_cols_len;
  table_col *ta_cols_val;
 } ta_cols;
 char *ta_path;
}
```

The ta_type member contains a string that identifies the type of entries in this table. NIS+ does not enforce any policies as to the contents of this string. However, when entries are added to the table, the NIS+ service will check to see that they have the same "type" as the table as specified by this member.

The structure ta_cols contains two members. ta_cols_val is an array of table_col structures. The length of the array depends on the number of columns in the table; it is defined when the table is created and is stored in ta_cols_len. ta_maxcol also contains the number of columns in the table and always has the same value as ta_cols_len. Once the table is created, this length field cannot be changed.

The ta_sep character is used by client applications that wish to print out an entry from the table. Typically this is either space (" ") or colon (":").

The ta_path string defines a concatenation path for tables. This string contains an ordered list of fully qualified table names, separated by colons, that are to be searched if a search on this table fails to match any entries. This path is only used with the flag FOLLOW_PATH with a nis_list( ) call. See nis_tables(3NSL) for information on these flags.

In addition to checking the type, the service will check that the number of columns in an entry is the same as those in the table before allowing that entry to be added.

Each column has associated with it a name in tc_name, a set of flags in tc_flags, and a set of access rights in tc_rights. The name should be indicative of the contents of that column.

The TA_BINARY flag indicates that data in the column is binary (rather than text). Columns that are searchable cannot contain binary data. The TA_CRYPT flag specifies that the information in this column should be encrypted prior to sending it over the network. This flag has no effect in the export version of NIS+. The TA_XDR flag is used to tell the client application that the data in this column is encoded using the XDR protocol. The TA_BINARY flag must be specified with the XDR flag. Further, by convention, the name of a column that has the TA_XDR flag set is the name of the XDR function that will decode the data in that column.

The TA_SEARCHABLE flag specifies that values in this column can be searched. Searchable columns must contain textual data and must have a name associated with them. The flag TA_CASE specifies that searches involving this column ignore the case of the value in the column. At least one of the columns in the table should be searchable. Also, the combination of all searchable column values should uniquely select an entry within the table. The TA_MODIFIED flag is set only when the table column is modified. When TA_MODIFIED is set, and the object is modified again, the modified access rights for the table column must be copied, not the default access rights.

**Entry Objects**

Entry objects are stored in tables. The structure used to define the entry data is as follows.

```
#define EN_BINARY 1
#define EN_CRYPT 2
```

```
#define EN_XDR 4
#define EN_MODIFIED 8
struct entry_col {
 uint_t ec_flags;
 struct {
  uint_t ec_value_len;
  char *ec_value_val;
 } ec_value;
}
typedef struct entry_col entry_col;
struct entry_obj {
 char *en_type;
 struct {
  uint_t en_cols_len;
  entry_col *en_cols_val;
 } en_cols;
}
```

The en_type member contains a string that specifies the type of data this entry represents. The NIS+ server will compare this string to the type string specified in the table object and disallow any updates or modifications if they differ.

The en_cols structure contains two members: en_cols_len and en_cols_val. en_cols_val is an array of entry_col structures. en_cols_len contains a count of the number of cells in the en_cols_val array and reflects the number of columns in the table – it always contains the same value as the table_obj.ta_cols.ta_cols_len member from the table which contains the entry.

The entry_col structure contains information about the entry's per-column values. ec_value contains information about a particular value. It has two members: ec_value_val, which is the value itself, and ec_value_len, which is the length (in bytes) of the value. entry_col also contains the member ec_flags, which contains a set of flags for the entry.

The flags in ec_flags are primarily used when adding or modifying entries in a table. All columns that have the flag EN_CRYPT set will be encrypted prior to sending them over the network. Columns with EN_BINARY set are presumed to contain binary data. The server will ensure that the column in the table object specifies binary data prior to allowing the entry to be added. When modifying entries in a table, only those columns that have changed need be sent to the server. Those columns should each have the EN_MODIFIED flag set to indicate this to the server.

## SEE ALSO

nis+(1), nis_groups(3NSL), nis_names(3NSL), nis_server(3NSL), nis_subr(3NSL), nis_tables(3NSL)

**NAME**

nis_ping, nis_checkpoint – misc NIS+ log administration functions

**SYNOPSIS**

```
cc [ flag ... ] file ... –lnsl [ library ... ]
#include <rpcsvc/nis.h>
void nis_ping(nis_name dirname, uint32_t utime, nis_object *dirobj);

nis_result *nis_checkpoint(nis_name dirname);
```

**DESCRIPTION**

`nis_ping( )` is called by the master server for a directory when a change has occurred within that directory. The parameter `dirname` identifies the directory with the change. If the parameter *dirobj* is `NULL,` this function looks up the directory object for `dirname` and uses the list of replicas it contains. The parameter *utime* contains the timestamp of the last change made to the directory. This timestamp is used by the replicas when retrieving updates made to the directory.

The effect of calling `nis_ping( )` is to schedule an update on the replica. A short time after a ping is received, typically about two minutes, the replica compares the last update time for its databases to the timestamp sent by the ping. If the ping timestamp is later, the replica establishes a connection with the master server and request all changes from the log that occurred after the last update that it had recorded in its local log.

`nis_checkpoint( )` is used to force the service to checkpoint information that has been entered in the log but has not been checkpointed to disk. When called, this function checkpoints the database for each table in the directory, the database containing the directory and the transaction log. Care should be used in calling this function since directories that have seen a lot of changes may take several minutes to checkpoint. During the checkpointing process, the service will be unavailable for updates for all directories that are served by this machine as master.

`nis_checkpoint( )` returns a pointer to a *nis_result* structure (described in `nis_tables`(3NSL) ). This structure should be freed with `nis_freeresult( )` (see `nis_names`(3NSL) ). The only items of interest in the returned result are the status value and the statistics.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`nislog`(1M) , `nis_names`(3NSL) , `nis_tables`(3NSL) , `nisfiles`(4) , `attributes`(5)

**NAME**

nis_server, nis_mkdir, nis_rmdir, nis_servstate, nis_stats, nis_getservlist, nis_freeservlist, nis_freetags – miscellaneous NIS+ functions

**SYNOPSIS**

```
cc [ flag ... ] file ... -lnsl [ library ... ]
#include <rpcsvc/nis.h>
nis_error nis_mkdir(nis_name dirname, nis_server *machine);

nis_error nis_rmdir(nis_name dirname, nis_server *machine);

nis_error nis_servstate(nis_server *machine, nis_tag *tags, int numtags, nis_tag **result);

nis_error nis_stats(nis_server *machine, nis_tag *tags, int numtags, nis_tag **result);

void nis_freetags(nis_tag *tags, int numtags);

nis_server **nis_getservlist(nis_name dirname);

void nis_freeservlist(nis_server **machines);
```

**DESCRIPTION**

These functions provide a variety of services for NIS+ applications.

`nis_mkdir()` is used to create the necessary databases to support NIS+ service for a directory, *dirname*, on a server, *machine*. If this operation is successful, it means that the directory object describing *dirname* has been updated to reflect that server *machine* is serving the named directory. For a description of the `nis_server` structure, refer to `nis_objects`(3NSL).

Per-server and per-directory access restrictions may apply to `nis_mkdir()`. See `nisopaccess`(1)

`nis_rmdir()` is used to delete the directory, *dirname*, from the specified server machine. The *machine* parameter cannot be `NULL`. Note that `nis_rmdir()` does not remove the directory *dirname* from the namespace or remove a server from the server list in the directory object. To remove a directory from the namespace you must call `nis_remove()` to remove the directory `dirname` from the namespace and call `nis_rmdir()` for each server in the server list to remove the directory from the server. To remove a replica from the server list, you need to first call `nis_modify()` to remove the server from the directory object and then call `nis_rmdir()` to remove the replica.

Per-server and per-directory access restrictions may apply to `nis_rmdir()`. See `nisopaccess`(1)

For a description of the `nis_server` structure, refer to `nis_objects`(3NSL).

`nis_servstate()` is used to set and read the various state variables of the NIS+ servers. In particular the internal debugging state of the servers may be set and queried.

The `nis_stats()` function is used to retrieve statistics about how the server is operating. Tracking these statistics can help administrators determine when they need to add additional replicas or to break up a domain into two or more subdomains. For more information on reading statistics, see `nisstat`(1M)

`nis_servstate( )` and `nis_stats( )` use the tag list. This tag list is a variable length array of *nis_tag* structures whose length is passed to the function in the *numtags* parameter. The set of legal tags are defined in the file `<rpcsvc/nis_tags.h>` which is included in `<rpcsvc/nis.h>` . Because these tags can and do vary between implementations of the NIS+ service, it is best to consult this file for the supported list. Passing unrecognized tags to a server will result in their *tag_value* member being set to the string “unknown.” Both of these functions return their results in malloced tag structure, **result* . If there is an error, **result* is set to `NULL` . The *tag_value* pointers points to allocated string memory which contains the results. Use `nis_freetags( )` to free the tag structure.

Per-server and per-directory access restrictions may apply to the NIS_SERVSTATE or NIS_STATUS (`nis_stats( )` ) operations and their sub-operations (*tags* ). See `nisopaccess`(1)

`nis_getservlist( )` returns a null terminated list of *nis_server* structures that represent the list of servers that serve the domain named *dirname* . Servers from this list can be used when calling functions that require the name of a NIS+ server. For a description of the `nis_server` refer to `nis_objects`(3NSL) . `nis_freeservlist( )` frees the list of servers list of servers returned by `nis_getservlist( )` . Note that this is the only legal way to free that list.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`nisopaccess`(1) , `nisstat`(1M) , `nis_names`(3NSL) , `nis_objects`(3NSL) , `nis_subr`(3NSL) , `attributes`(5)

## NAME

nis_subr, nis_leaf_of, nis_name_of, nis_domain_of, nis_getnames, nis_freenames, nis_dir_cmp, nis_clone_object, nis_destroy_object, nis_print_object – NIS+ subroutines

## SYNOPSIS

`cc` [ *flag* ... ] *file* ... `−lnsl` [ *library* ... ]
#include <rpcsvc/nis.h>
nis_name **nis_leaf_of**(const nis_name *name*);

nis_name **nis_name_of**(const nis_name *name*);

nis_name **nis_domain_of**(const nis_name *name*);

nis_name ***nis_getnames**(const nis_name *name*);

void **nis_freenames**(nis_name **namelist*);

name_pos **nis_dir_cmp**(const nis_name *n1*, const nis_name *n2*);

nis_object ***nis_clone_object**(const nis_object **src*, nis_object **dest*);

void **nis_destroy_object**(nis_object **obj*);

void **nis_print_object**(const nis_object **obj*);

## DESCRIPTION

These subroutines are provided to assist in the development of NIS+ applications. They provide several useful operations on both NIS+ names and objects.

The first group, `nis_leaf_of( )` , `nis_domain_of( )` , and `nis_name_of( )` provide the functions for parsing NIS+ names. `nis_leaf_of( )` will return the first label in an NIS+ name. It takes into account the double quote character ‘"’ which can be used to protect embedded ‘.’ (dot) characters in object names. Note that the name returned will never have a trailing dot character. If passed the global root directory name ".", it will return the null string.

`nis_domain_of( )` returns the name of the NIS+ domain in which an object resides. This name will always be a fully qualified NIS+ name and ends with a dot. By iteratively calling `nis_leaf_of( )` and `nis_domain_of( )` it is possible to break a NIS+ name into its individual components.

`nis_name_of( )` is used to extract the unique part of a NIS+ name. This function removes from the tail portion of the name all labels that are in common with the local domain. Thus if a machine were in domain `foo.bar.baz.` and `nis_name_of( )` were passed a name `bob.friends.foo.bar.baz` , then `nis_name_of( )` would return the unique part, `bob.friends` . If the name passed to this function is not in either the local domain or one of its children, this function will return null.

`nis_getnames( )` will return a list of candidate names for the name passed in as *name* . If this name is not fully qualified, `nis_getnames( )` will generate a list of names using the default NIS+ directory search path, or the environment variable `NIS_PATH` if it is set. The returned array of pointers is terminated by a NULL pointer, and the memory associated with this array should be freed by calling `nis_freenames().`

Though `nis_dir_cmp()` can be used to compare any two NIS+ names, it is used primarily to compare domain names. This comparison is done in a case independent fashion, and the results are an enum of type `name_pos` . When the names passed to this function are identical, the function returns a value of SAME_NAME. If the name *n1* is a direct ancestor of name *n2* , then this function returns the result HIGHER_NAME. Similarly, if the name *n1* is a direct descendant of name *n2* , then this function returns the result LOWER_NAME. When the name *n1* is neither a direct ancestor nor a direct descendant of *n2* , as it would be if the two names were siblings in separate portions of the namespace, then this function returns the result NOT_SEQUENTIAL. Finally, if either name cannot be parsed as a legitimate name then this function returns the value BAD_NAME.

The second set of functions, consisting of `nis_clone_object()` and `nis_destroy_object()` , are used for manipulating objects. `nis_clone_object()` creates an exact duplicate of the NIS+ object *src* . If the value of *dest* is non-null, it creates the clone of the object into this object structure and allocate the necessary memory for the variable length arrays. If this parameter is null, a pointer to the cloned object is returned. Refer to `nis_objects`(3NSL) for a description of the `nis_object` structure.

`nis_destroy_object()` can be used to destroy an object created by `nis_clone_object()` . This will free up all memory associated with the object and free the pointer passed. If the object was cloned into an array (using the *dest* parameter to `nis_clone_object()` ) then the object *cannot* be freed with this function. Instead, the function `xdr_free(xdr_nis_object,` *dest* ) must be used.

`nis_print_object()` prints out the contents of a NIS+ object structure on the standard output. Its primary use is for debugging NIS+ programs.

## ENVIRONMENT VARIABLES

`NIS_PATH` This variable overrides the default NIS+ directory search path used by `nis_getnames().` It contains an ordered list of directories separated by ':' (colon) characters. The '$' (dollar sign) character is treated specially. Directory names that end in '$' have the default domain appended to them, and a '$' by itself is replaced by the list of directories between the default domain and the global root that are at least two levels deep. The default NIS+ directory search path is '$'.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

**SEE ALSO** `nis_names`(3NSL) , `nis_objects`(3NSL) , `nis_tables`(3NSL) , `attributes`(5)

**NOTES** `nis_leaf_of()` , `nis_name_of()` and `nis_clone_object()` return their results as thread-specific data in multithreaded applications.

**NAME**

nis_tables, nis_list, nis_add_entry, nis_remove_entry, nis_modify_entry, nis_first_entry, nis_next_entry – NIS+ table functions

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lnsl` [ *library* ... ]
#include <rpcsvc/nis.h>
nis_result ***nis_list**(nis_name *name*, uint_t*flags*, int (*callback)(nis_name *table_name*, nis_object **object*, void **userdata* ), void **userdata*);

nis_result ***nis_add_entry**(nis_name *table_name*, nis_object **object*, uint_t *flags*);

nis_result ***nis_remove_entry**(nis_name *name*, nis_object **object*, uint_t*flags*);

nis_result ***nis_modify_entry**(nis_name *name*, nis_object **object*, uint_t*flags*);

nis_result ***nis_first_entry**(nis_name *table_name*);

nis_result ***nis_next_entry**(nis_name *table_name*, netobj **cookie*);

void **nis_freeresult**(nis_result **result*);

**DESCRIPTION**

These functions are used to search and modify NIS+ tables. `nis_list( )` is used to search a table in the NIS+ namespace. `nis_first_entry( )` and `nis_next_entry( )` are used to enumerate a table one entry at a time. `nis_add_entry( )` , `nis_remove_entry( )` , and `nis_modify_entry( )` are used to change the information stored in a table. `nis_freeresult( )` is used to free the memory associated with the `nis_result` structure.

Entries within a table are named by NIS+ indexed names. An indexed name is a compound name that is composed of a search criteria and a simple NIS+ name that identifies a table object. A search criteria is a series of column names and their associated values enclosed in bracket '[]' characters. Indexed names have the following form:

```
[ colname=value, ... ],tablename
```

The list function, `nis_list( )` , takes an indexed name as the value for the *name* parameter. Here, the tablename should be a fully qualified NIS+ name unless the `EXPAND_NAME` flag (described below) is set. The second parameter, *flags* , defines how the function will respond to various conditions. The value for this parameter is created by logically `OR` ing together one or more flags from the following list.

`FOLLOW_LINKS` If the table specified in *name* resolves to be a `LINK` type object (see `nis_objects`(3NSL) ), this flag specifies that the client library follow that link and do the search at that object. If this flag is not set and the name resolves to a link, the error `NIS_NOTSEARCHABLE` will be returned.

| | |
|---|---|
| `FOLLOW_PATH` | This flag specifies that if the entry is not found within this table, the list operation should follow the path specified in the table object. When used in conjunction with the `ALL_RESULTS` flag below, it specifies that the path should be followed regardless of the result of the search. When used in conjunction with the `FOLLOW_LINKS` flag above, named tables in the path that resolve to links will be followed until the table they point to is located. If a table in the path is not reachable because no server that serves it is available, the result of the operation will be either a "soft" success or a "soft" failure to indicate that not all tables in the path could be searched. If a name in the path names is either an invalid or non-existent object then it is silently ignored. |
| `HARD_LOOKUP` | This flag specifies that the operation should continue trying to contact a server of the named table until a definitive result is returned (such as `NIS_NOTFOUND` ). |
| `ALL_RESULTS` | This flag can only be used in conjunction with `FOLLOW_PATH` and a callback function. When specified, it forces all of the tables in the path to be searched. If *name* does not specify a search criteria (imply that all entries are to be returned), then this flag will cause all of the entries in all of the tables in the path to be returned. |
| `NO_CACHE` | This flag specifies that the client library should bypass any client object caches and get its information directly from either the master server or a replica server for the named table. |
| `MASTER_ONLY` | This flag is even stronger than `NO_CACHE` in that it specifies that the client library should *only* get its information from the master server for a particular table. This guarantees that the information will be up to date. However, there may be severe performance penalties associated with contacting the master server directly on large networks. When used in conjunction with the `HARD_LOOKUP` flag, this will block the list operation until the master server is up and available. |
| `EXPAND_NAME` | When specified, the client library will attempt to expand a partially qualified name by calling `nis_getnames( )` (see `nis_local_names`(3NSL) ) which uses the environment variable `NIS_PATH` . |

| | |
|---|---|
| `RETURN_RESULT` | This flag is used to specify that a copy of the returning object be returned in the `nis_result` structure if the operation was successful. |

The third parameter to `nis_list()` , *callback* , is an optional pointer to a function that will process the `ENTRY` type objects that are returned from the search. If this pointer is `NULL,` then all entries that match the search criteria are returned in the *nis_result* structure, otherwise this function will be called once for each entry returned. When called, this function should return `0` when additional objects are desired and `1` when it no longer wishes to see any more objects. The fourth parameter, *userdata* , is simply passed to callback function along with the returned entry object. The client can use this pointer to pass state information or other relevant data that the callback function might need to process the entries.

The `nis_list()` function is not MT-Safe with callbacks. See `NOTES` .

`nis_add_entry()` will add the NIS+ object to the NIS+ *table_name* . The *flags* parameter is used to specify the failure semantics for the add operation. The default (*flags* equal 0) is to fail if the entry being added already exists in the table. The `ADD_OVERWRITE` flag may be used to specify that existing object is to be overwritten if it exists, (a modify operation) or added if it does not exist. With the `ADD_OVERWRITE` flag, this function will fail with the error `NIS_PERMISSION` if the existing object does not allow modify privileges to the client.

If the flag `RETURN_RESULT` has been specified, the server will return a copy of the resulting object if the operation was successful.

`nis_remove_entry()` removes the identified entry from the table or a set of entries identified by *table_name* . If the parameter *object* is non-null, it is presumed to point to a cached copy of the entry. When the removal is attempted, and the object that would be removed is not the same as the cached object pointed to by *object* then the operation will fail with an `NIS_NOTSAMEOBJ` error. If an object is passed with this function, the search criteria in name is optional as it can be constructed from the values within the entry. However, if no object is present, the search criteria must be included in the *name* parameter. If the flags variable is null, and the search criteria does not uniquely identify an entry, the `NIS_NOTUNIQUE` error is returned and the operation is aborted. If the flag parameter `REM_MULTIPLE` is passed, and if remove permission is allowed for each of these objects, then all objects that match the search criteria will be removed. Note that a null search criteria and the `REM_MULTIPLE` flag will remove all entries in a table.

`nis_modify_entry()` modifies an object identified by *name* . The parameter *object* should point to an entry with the `EN_MODIFIED` flag set in each column that contains new information.

The owner, group, and access rights of an entry are modified by placing the modified information into the respective fields of the parameter, *object* : `zo_owner` , `zo_group` , and `zo_access` .

These columns will replace their counterparts in the entry that is stored in the table. The entry passed must have the same number of columns, same type, and valid data in the modified columns for this operation to succeed.

If the flags parameter contains the flag `MOD_SAMEOBJ` then the object pointed to by *object* is assumed to be a cached copy of the original object. If the OID of the object passed is different than the OID of the object the server fetches, then the operation fails with the `NIS_NOTSAMEOBJ` error. This can be used to implement a simple read-modify-write protocol which will fail if the object is modified before the client can write the object back.

If the flag `RETURN_RESULT` has been specified, the server will return a copy of the resulting object if the operation was successful.

`nis_first_entry( )` fetches entries from a table one at a time. This mode of operation is extremely inefficient and callbacks should be used instead wherever possible. The table containing the entries of interest is identified by *name* . If a search criteria is present in *name* it is ignored. The value of *cookie* within the `nis_result` structure must be copied by the caller into local storage and passed as an argument to `nis_next_entry( )` .

`nis_next_entry( )` retrieves the "next" entry from a table specified by *table_name* . The order in which entries are returned is not guaranteed. Further, should an update occur in the table between client calls to `nis_next_entry( )` there is no guarantee that an entry that is added or modified will be seen by the client. Should an entry be removed from the table that would have been the "next" entry returned, the error `NIS_CHAINBROKEN` is returned instead.

## RETURN VALUES

These functions return a pointer to a structure of type `nis_result` :

```
struct nis_result {
  nis_error status;
  struct {
   uint_t objects_len;
   nis_object *objects_val;
  } objects;
  netobj cookie;
  uint32_t zticks;
  uint32_t dticks;
  uint32_t aticks;
  uint32_t cticks;
        };
```

The *status* member contains the error status of the the operation. A text message that describes the error can be obtained by calling the function `nis_sperrno()` (see `nis_error`(3NSL) ).

The `objects` structure contains two members. *objects_val* is an array of *nis_object* structures; *objects_len* is the number of cells in the array. These objects will be freed by a call to `nis_freeresult()` (see `nis_names`(3NSL) ). If you need to keep a copy of one or more objects, they can be copied with the function `nis_clone_object()` and freed with the function `nis_destroy_object()` (see `nis_server`(3NSL) ).

The various ticks contain details of where the time (in microseconds) was taken during a request. They can be used to tune one's data organization for faster access and to compare different database implementations.

*zticks* The time spent in the NIS+ service itself, this count starts when the server receives the request and stops when it sends the reply.

*dticks* The time spent in the database backend, this time is measured from the time a database call starts, until a result is returned. If the request results in multiple calls to the database, this is the sum of all the time spent in those calls.

*aticks* The time spent in any "accelerators" or caches. This includes the time required to locate the server needed to resolve the request.

*cticks* The total time spent in the request, this clock starts when you enter the client library and stops when a result is returned. By subtracting the sum of the other ticks values from this value you can obtain the local overhead of generating a NIS+ request.

Subtracting the value in *dticks* from the value in *zticks* will yield the time spent in the service code itself. Subtracting the sum of the values in *zticks* and *aticks* from the value in *cticks* will yield the time spent in the client library itself. Note: all of the tick times are measured in microseconds.

**ERRORS**

The client library can return a variety of error returns and diagnostics. The more salient ones are documented below.

| | |
|---|---|
| `NIS_BADATTRIBUTE` | The name of an attribute did not match up with a named column in the table, or the attribute did not have an associated value. |
| `NIS_BADNAME` | The name passed to the function is not a legal NIS+ name. |
| `NIS_BADREQUEST` | A problem was detected in the request structure passed to the client library. |

| | |
|---|---|
| NIS_CACHEEXPIRED | The entry returned came from an object cache that has *expired* . This means that the time to live value has gone to zero and the entry may have changed. If the flag NO_CACHE was passed to the lookup function then the lookup function will retry the operation to get an unexpired copy of the object. |
| NIS_CBERROR | An RPC error occurred on the server while it was calling back to the client. The transaction was aborted at that time and any unsent data was discarded. |
| NIS_CBRESULTS | Even though the request was successful, all of the entries have been sent to your callback function and are thus not included in this result. |
| NIS_FOREIGNNS | The name could not be completely resolved. When the name passed to the function would resolve in a namespace that is outside the NIS+ name tree, this error is returned with a NIS+ object of type DIRECTORY . The returned object contains the type of namespace and contact information for a server within that namespace. |
| NIS_INVALIDOBJ | The object pointed to by *object* is not a valid NIS+ entry object for the given table. This could occur if it had a mismatched number of columns, or a different data type (for example, binary or text) than the associated column in the table. |
| NIS_LINKNAMEERROR | The name passed resolved to a LINK type object and the contents of the object pointed to an invalid name. |
| NIS_MODFAIL | The attempted modification failed for some reason. |
| NIS_NAMEEXISTS | An attempt was made to add a name that already exists. To add the name, first remove the existing name and then add the new name or modify the existing named object. |
| NIS_NAMEUNREACHABLE | This soft error indicates that a server for the desired directory of the named table object could not be reached. This can occur when there is a network partition or the server has crashed. |

| | |
|---|---|
| | Attempting the operation again may succeed. See the `HARD_LOOKUP` flag. |
| `NIS_NOCALLBACK` | The server was unable to contact the callback service on your machine. This results in no data being returned. |
| `NIS_NOMEMORY` | Generally a fatal result. It means that the service ran out of heap space. |
| `NIS_NOSUCHNAME` | This hard error indicates that the named directory of the table object does not exist. This occurs when the server that should be the parent of the server that serves the table, does not know about the directory in which the table resides. |
| `NIS_NOSUCHTABLE` | The named table does not exist. |
| `NIS_NOT_ME` | A request was made to a server that does not serve the given name. Normally this will not occur, however if you are not using the built in location mechanism for servers, you may see this if your mechanism is broken. |
| `NIS_NOTFOUND` | No entries in the table matched the search criteria. If the search criteria was null (return all entries) then this result means that the table is empty and may safely be removed by calling the `nis_remove()` .<br><br>If the FOLLOW_PATH flag was set, this error indicates that none of the tables in the path contain entries that match the search criteria. |
| `NIS_NOTMASTER` | A change request was made to a server that serves the name, but it is not the master server. This can occur when a directory object changes and it specifies a new master server. Clients that have cached copies of the directory object in the `/var/nis/NIS_SHARED_DIRCACHE` file will need to have their cache managers restarted (use `nis_cachemgr -i` ) to flush this cache. |
| `NIS_NOTSAMEOBJ` | An attempt to remove an object from the namespace was aborted because the object that would have been removed was not the same object that was passed in the request. |

| | |
|---|---|
| `NIS_NOTSEARCHABLE` | The table name resolved to a NIS+ object that was not searchable. |
| `NIS_PARTIAL` | This result is similar to `NIS_NOTFOUND` except that it means the request succeeded but resolved to zero entries. When this occurs, the server returns a copy of the table object instead of an entry so that the client may then process the path or implement some other local policy. |
| `NIS_RPCERROR` | This fatal error indicates the RPC subsystem failed in some way. Generally there will be a `syslog`(3C) message indicating why the RPC request failed. |
| `NIS_S_NOTFOUND` | The named entry does not exist in the table, however not all tables in the path could be searched, so the entry may exist in one of those tables. |
| `NIS_S_SUCCESS` | Even though the request was successful, a table in the search path was not able to be searched, so the result may not be the same as the one you would have received if that table had been accessible. |
| `NIS_SUCCESS` | The request was successful. |
| `NIS_SYSTEMERROR` | Some form of generic system error occurred while attempting the request. Check the `syslog`(3C) record for error messages from the server. |
| `NIS_TOOMANYATTRS` | The search criteria passed to the server had more attributes than the table had searchable columns. |
| `NIS_TRYAGAIN` | The server connected to was too busy to handle your request. `add_entry( )` , `remove_entry( )` , and `modify_entry( )` return this error when the master server is currently updating its internal state. It can be returned to `nis_list( )` when the function specifies a callback and the server does not have the resources to handle callbacks. |
| `NIS_TYPEMISMATCH` | An attempt was made to add or modify an entry in a table, and the entry passed was of a different type than the table. |

## ENVIRONMENT VARIABLES

NIS_PATH When set, this variable is the search path used by nis_list( ) if the flag EXPAND_NAME is set.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe with exceptions |

## SEE ALSO

niscat(1) , niserror(1) , nismatch(1) , nis_cachemgr(1M) , nis_clone_object(3NSL) , n, nis_destroy_object(3NSL) , nis_error(3NSL) , nis_getnames(3NSL) , nis_local_names(3NSL) , nis_names(3NSL) , nis_objects(3NSL) , nis_server(3NSL) , rpc_svc_calls(3NSL) , syslog(3C) , attributes(5)

## WARNINGS

Use the flag HARD_LOOKUP carefully since it can cause the application to block indefinitely during a network partition.

## NOTES

The path used when the flag FOLLOW_PATH is specified, is the one present in the *first* table searched. The path values in tables that are subsequently searched are ignored.

It is legal to call functions that would access the nameservice from within a list callback. However, calling a function that would itself use a callback, or calling nis_list( ) with a callback from within a list callback function is not currently supported.

There are currently no known methods for nis_first_entry( ) and nis_next_entry( ) to get their answers from only the master server.

The nis_list( ) function is not MT-Safe with callbacks. nis_list( ) callbacks are serialized. A call to nis_list( ) with a callback from within nis_list( ) will deadlock. nis_list( ) with a callback cannot be called from an rpc server. See rpc_svc_calls(3NSL) . Otherwise, this function is MT-Safe.

**NAME**

nlsgetcall – get client’s data passed via the listener

**SYNOPSIS**

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

struct t_call *nlsgetcall(int fildes);
```

**DESCRIPTION**

`nlsgetcall( )` allows server processes started by the listener process to access the client’s `t_call` structure, that is, the *sndcall* argument of `t_connect`(3NSL).

The `t_call` structure returned by `nlsgetcall( )` can be released using `t_free`(3NSL).

`nlsgetcall( )` returns the address of an allocated `t_call` structure or NULL if a `t_call` structure cannot be allocated. If the `t_alloc( )` succeeds, undefined environment variables are indicated by a negative *len* field in the appropriate `netbuf` structure. A *len* field of zero in the `netbuf` structure is valid and means that the original buffer in the listener’s `t_call` structure was NULL.

**RETURN VALUES**

A NULL pointer is returned if a `t_call` structure cannot be allocated by `t_alloc( )`. `t_errno` can be inspected for further error information. Undefined environment variables are indicated by a negative length field (*len*) in the appropriate `netbuf` structure.

**FILES**

`/usr/lib/libnsl_s.a`
`/usr/lib/libslan.a`
`/usr/lib/libnls.a`

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Unsafe |

**SEE ALSO**

`nlsadmin`(1M), `getenv`(3C), `t_alloc`(3NSL), `t_connect`(3NSL), `t_error`(3NSL), `t_free`(3NSL), `t_sync`(3NSL), `attributes`(5)

**WARNINGS**

The *len* field in the `netbuf` structure is defined as being unsigned. In order to check for error returns, it should first be cast to an int.

The listener process limits the amount of user data (*udata*) and options data (*opt*) to 128 bytes each. Address data *addr* is limited to 64 bytes. If the original data was longer, no indication of overflow is given.

**NOTES**

Server processes must call `t_sync`(3NSL) before calling this routine.

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

**NAME**

nlsprovider – get name of transport provider

**SYNOPSIS**

char ***nlsprovider**(void);

**DESCRIPTION**

`nlsprovider( )` returns a pointer to a null-terminated character string which contains the name of the transport provider as placed in the environment by the listener process. If the variable is not defined in the environment, a NULL pointer is returned.

The environment variable is only available to server processes started by the listener process.

**RETURN VALUES**

If the variable is not defined in the environment, a NULL pointer is returned.

**FILES**

`/usr/lib/libslan.a` (7300)
`/usr/lib/libnls.a` (3B2 Computer)
`/usr/lib/libnsl_s.a`

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Unsafe |

**SEE ALSO**

`nlsadmin`(1M), `attributes`(5)

**NOTES**

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

## NAME

nlsrequest – format and send listener service request message

## SYNOPSIS

```
#include <listen.h>

int nlsrequest(int fildes, char *service_code);
extern int _nlslogt_errno;
extern char *_nlsrmsg;
```

## DESCRIPTION

Given a virtual circuit to a listener process (*fildes*) and a service code of a server process, `nlsrequest()` formats and sends a *service request message* to the remote listener process requesting that it start the given service. `nlsrequest()` waits for the remote listener process to return a *service request response message*, which is made available to the caller in the static, null-terminated data buffer pointed to by `_nlsrmsg`. The *service request response message* includes a success or failure code and a text message. The entire message is printable.

## RETURN VALUES

The success or failure code is the integer return code from `nlsrequest()`. Zero indicates success, other negative values indicate `nlsrequest()` failures as follows:

| | |
|---|---|
| `–1` | Error encountered by `nlsrequest()`, see `t_errno`. |

Positive values are error return codes from the *listener* process. Mnemonics for these codes are defined in `<listen.h>`.

| | |
|---|---|
| `2` | Request message not interpretable. |
| `3` | Request service code unknown. |
| `4` | Service code known, but currently disabled. |

If non-null, `_nlsrmsg` contains a pointer to a static, null-terminated character buffer containing the *service request response message*. Note that both `_nlsrmsg` and the data buffer are overwritten by each call to `nlsrequest()`.

If `_nlslog` is non-zero, `nlsrequest()` prints error messages on stderr. Initially, `_nlslog` is zero.

## FILES

`/usr/lib/libnls.a`
`/usr/lib/libslan.a`
`/usr/lib/libnsl_s.a`

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Unsafe |

## SEE ALSO

`nlsadmin`(1M), `t_error`(3NSL), `t_snd`(3NSL), `t_rcv`(3NSL), `attributes`(5)

**WARNINGS**

`nlsrequest()` cannot always be certain that the remote server process has been successfully started. In this case, `nlsrequest()` returns with no indication of an error and the caller will receive notification of a disconnect event by way of a `T_LOOK` error before or during the first `t_snd()` or `t_rcv()` call.

**NOTES**

These interfaces are unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

## NAME

rcmd, rcmd_af, rresvport, rresvport_af, ruserok – routines for returning a stream to a remote command

## SYNOPSIS

`cc` [ *flag* ... ] *file* ... `–lsocket –lnsl` [ *library* ... ]

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

`rcmd( )` is a routine used by the superuser to execute a command on a remote machine using an authentication scheme based on reserved port numbers. It is assumed that an `AF_INET` socket is returned with `rcmd( )` . `rcmd_af( )` allows the application to choose which type of socket is returned by passing in the address family, either `AF_INET` or `AF_INET6` .

`rresvport( )` is a routine that returns a descriptor to a socket with an address in the privileged port space. `rresvport_af( )` is equivalent to `rresvport( )` , except that you can choose the type of socket address family that will be returned by `rresvport_af( )` , either `AF_INET` or `AF_INET6` .

`ruserok( )` is a routine used by servers to authenticate clients requesting service with `rcmd` .

All of these functions are present in the same file and are used by the `in.rshd`(1M) server (among others).

`rcmd( )` and `rcmd_af( )` look up the host **ahost* using `getipnodebyname`(3SOCKET) , returning `-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 well-known Internet port *inport* .

If the connection succeeds, a socket in the Internet domain of type `SOCK_STREAM` is returned to the caller, and given to the remote command as its standard input (file descriptor 0) and standard output (file descriptor 1). If *fd2p* is non-zero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p* . The control process will return diagnostic output from the command (file descriptor 2) on this channel, and will also accept bytes on this channel as signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the standard error (file descriptor 2) of the remote command will be made the same as its standard output and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

The protocol is described in detail in `in.rshd`(1M) .

The `rresvport( )` and `rresvport_af( )` routines are used to obtain a socket bound to a privileged port number. This 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.

`ruserok( )` takes a remote host's name, as returned by a `gethostbyaddr( )` routine, two user names and a flag indicating whether the local user's name is that of the superuser. See `gethostbyname`(3NSL) . It 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. `0` is returned if the machine name is listed in the `/etc/hosts.equiv` file, or the host and remote user name are found in the `.rhosts` file; otherwise `ruserok( )` returns `-1` . If the superuser flag is `1` , the checking of the `/etc/hosts.equiv` file is bypassed.

**RETURN VALUES**

`rcmd( )` and `rcmd_af( )` return a valid socket descriptor upon success. They returns `-1` upon error and print a diagnostic message to standard error.

`rresvport( )` and `rresvport_af( )` return a valid, bound socket descriptor upon success. They 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 |

**SEE ALSO**

`rlogin`(1) , `rsh`(1) , `in.rexecd`(1M) , `in.rshd`(1M) , `intro`(2) , `gethostbyname`(3NSL) , `getipnodebyname`(3SOCKET) , `rexec`(3SOCKET) , `attributes`(5)

**NOTES**

The error code `EAGAIN` is overloaded to mean "All network ports in use."

These interfaces are unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

**NAME**

recv, recvfrom, recvmsg – receive a message from a socket

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lsocket –lnsl` [ *library* ... ]
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/uio.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*, int **fromlen*);

ssize_t **recvmsg**(int *s*, struct msghdr **msg*, int *flags*);

**DESCRIPTION**

`recv( )` , `recvfrom( )` , and `recvmsg( )` are used to receive messages from another socket. `recv( )` may be used only on a *connected* socket (see `connect`(3SOCKET) ), while `recvfrom( )` and `recvmsg( )` may be used to receive data on a socket whether it is in a connected state or not. *s* is a socket created with `socket`(3SOCKET) .

If *from* is not a NULL pointer, the source address of the message is filled in. *fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from* , and modified on return to indicate the actual size of the address stored there. 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 the message is received from (see `socket`(3SOCKET) ).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see `fcntl`(2) ) in which case `-1` is returned with the external variable `errno` set to `EWOULDBLOCK` .

The `select( )` call may be used to determine when more data arrives.

The *flags* parameter is formed by ORing 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, so that a subsequent receive operation will see the same data. |

The `recvmsg( )` call uses a `msghdr` structure to minimize the number of directly supplied parameters. This structure is defined in `<sys/socket.h>` and includes the following members:

```
caddr_t         msg_name;          /* optional address */
int             msg_namelen;       /* size of address */
struct iovec    *msg_iov;          /* scatter/gather array */
int             msg_iovlen;        /* # elements in msg_iov */
caddr_t         msg_accrights;     /* access rights sent/received */
int             msg_accrightslen;
```

Here `msg_name` and `msg_namelen` specify the destination address if the socket is unconnected; `msg_name` may be given as a NULL pointer if no names are desired or required. The `msg_iov` and `msg_iovlen` describe the scatter-gather locations, as described in `read`(2) . A buffer to receive any access rights sent along with the message is specified in `msg_accrights` , which has length `msg_accrightslen` .

## RETURN VALUES

These calls return the number of bytes received, or -1 if an error occurred.

## ERRORS

The calls fail if:

| | |
|---|---|
| `EBADF` | *s* is an invalid file descriptor. |
| `EINTR` | The operation was interrupted by delivery of a signal before any data was available to be received. |
| `EIO` | An I/O error occurred while reading from or writing to the file system. |
| `ENOMEM` | There was insufficient user memory available for the operation to complete. |
| `ENOSR` | There were insufficient STREAMS resources 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. |

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

## SEE ALSO

`fcntl`(2) , `ioctl`(2) , `read`(2) , `connect`(3SOCKET) , `getsockopt`(3SOCKET) , `send`(3SOCKET) , `socket`(3SOCKET) , `attributes`(5) , `socket`(3HEAD)

**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 |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`poll`(2), `recvmsg`(3XNET), `recvfrom`(3XNET), `select`(3C), `send`(3XNET), `sendmsg`(3XNET), `sendto`(3XNET), `shutdown`(3XNET), `socket`(3XNET), `attributes`(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 **buffer*, size_t *length*, int *flags*, struct sockaddr **address*, socklen_t **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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`poll`(2), `recv`(3XNET), `recvmsg`(3XNET), `select`(3C) `send`(3XNET), `sendmsg`(3XNET), `sendto`(3XNET), `shutdown`(3XNET), `socket`(3XNET), `attributes`(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 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. |

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, 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_iov* and *msg_iovlen* fields are used to specify where the received data will be stored. *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. 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.

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. |

## 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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`poll`(2), `recv`(3XNET), `recvfrom`(3XNET), `select`(3C), `send`(3XNET), `sendmsg`(3XNET), `sendto`(3XNET), `shutdown`(3XNET), `socket`(3XNET), `attributes`(5)

**NAME**

resolver, res_init, res_mkquery, res_mkupdate, res_mkupdrec, res_query, res_search, res_send, res_update, dn_comp, dn_expand – resolver routines

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lresolv –lsocket –lnsl` [ *library* ... ]
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>
int **res_init**(void);

int **res_mkquery**(int *op*, const char **dname*, int *class*, int *type*, const char **data*, int *datalen*, struct rrec **newrr*, uchar_t **buf*, int *buflen*);

int **res_mkupdate**(ns_updrec ***rrecp_in*, uchar **buf*, int *length*);

ns_updrec ***res_mkupdrec**(int *section*, const char **dname*, uint_t *class*, uint_t *type*, uint_t *ttl*);

int **res_query**(const char **dname*, int *class*, int *type*, uchar_t **answer*, int *anslen*);

int **res_search**(const char **dname*, int *class*, int *type*, uchar_t **answer*, int *anslen*);

int **res_send**(uchar_t **msg*, int *msglen*, uchar_t **answer*, int *anslen*);

int **res_update**(ns_updrec **rrecp_in*);

int **dn_comp**(const char **exp_dn*, uchar_t **comp_dn*, int *length*, uchar_t ***dnptrs*, uchar_t ***lastdnptr*);

int **dn_expand**(const uchar_t **msg*, const uchar_t **eomorig*, uchar_t **comp_dn*, char *exp_dn*, int *length*);

**DESCRIPTION**

These routines are used for making, sending, and interpreting query and reply messages passed to and from Internet domain name servers. The `res_update()` and `res_mkupdrec()` routines are used to dynamically update the name server with resource records.

The global structure `_res` holds options and state information. Option values can be set to affect the collective behavior of groups of resolver library routines. However, most resolver library routines use reasonable defaults so that the explicit enabling of an option is rarely required.

The library manual page entry for the resolver library includes public domain routines beyond those described here. See `libresolv`(3LIB) . Those function names that are exported but are not explained here are lower-level routines called by these routines. Their direct use is discouraged. If you do make direct use of unsupported routines, you do so at considerable added risk and with no expectation of documentation or other support beyond that available publicly.

Options for the resolver library are stored as a single bit mask containing the bitwise- OR sum of the options enabled. The options stored in `_res.options` are those defined in `<resolv.h>` and as follows. The field `_res.options` is a member of the `_res` structure.

| | |
|---|---|
| `RES_INIT` | True if the initial name server address and default domain name are initialized, that is, `res_init( )` 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_PRIMARY` | Query primary server only. This option is not implemented. |
| `RES_IGNTC` | Unused currently. 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( )` does not do iterative queries and expects the name server to handle recursion. |
| `RES_DEFNAMES` | If set, `res_search( )` appends the default domain name to single-component names (names that do not contain a dot). This is useful only in programs that regularly do many queries. UDP should be the normal mode used. |
| `RES_DNSRCH` | Enables searching up through the current domain tree. If this option is set, `res_search( )` searches for host names in the current domain and in parent domains. This 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_init`

If the system initialization file `resolv.conf` exists, `res_init( )` reads it to get the default domain name, the search list, and the Internet address of the local name server or servers. See `resolv.conf`(4) . If no server is configured by the local `resolv.conf` file, `res_init` tries to obtain name resolution services from the host on which it is running.

The `res_init( )` function also sets the RES_INIT field of the `_res` global structure so that other service routines (`res_search( )` ) can determine for certain whether it needs to be called first before other processing begins.

In the absence of a `resolv.conf` configuration file, the current domain is either set to the value of the environmental variable LOCALDOMAIN, derived from the domain name or derived from the host name. See `domainname`(1M) . The current domain name as defined in the system initialization file `resolv.conf` can be overridden by the environment variable LOCALDOMAIN . 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. Another environment variable ( RES_OPTIONS ) can be set to override certain internal resolver options. Otherwise, these options are set by changing fields in the global `_res` structure or they are inherited from the configuration file's *options* command. The syntax of the RES_OPTIONS environment variable is explained in `resolv.conf`(4) .

The initializations performed by `res_init( )` can be invoked explicitly with this function. However, they are normally performed automatically as a result of a call to one of the other resolver routines.

`res_search`

`res_search( )` formulates and sends a normal query (QUERY ) message, and stores the response in a buffer supplied by the caller.

The parameters *class* and `type` define the class and type of query. See `<arpa/nameser.h>` . The response is returned in the user-supplied buffer *answer* . `res_search` returns the length of *answer* in *anslen* . Like the other resolver routines, `res_search( )` calls `res_init( )` if the RES_INIT flag is not enabled.

The `res_search( )` function acts in a similar manner as `res_query( )` , except that it can implement the default and search rules controlled by the RES_DEFNAMES and RES_DNSRCH options.

The parameter *dname* is the domain name. However, if *dname* consists of a single-component name and the RES_DEFNAMES flag is enabled (the default), *dname* is appended with the current domain name.

If the RES_DNSRCH flag is enabled, `res_search( )` searches up the current domain tree until an answer has been retrieved or an unrecoverable error has been encountered. `res_search( )` returns the first successful reply.

`res_mkquery` The `res_mkquery( )` function constructs a standard query message and places it in *buf* .

`res_mkquery( )` returns the size of the query or `-1` if the query is larger than *buflen* . The *op* parameter is usually `QUERY` , but can be any of the query types defined in `<arpa/nameser.h>` .

The parameter *dname* is the domain name for the query.

The parameters *class* and `type` define the class and type of query (see `<arpa/nameser.h>` ). The parameter *data* is the resource record; *datalen* is the length of the record.

*newrr* is unused and should be specified as a null pointer (0).

`res_query` The *res_query* function provides an interface to most of the server query mechanism. It constructs a query, sends it to the local server, awaits a response, and makes preliminary checks on the reply. The query requests information of the specified `type` and *class* for the specified fully-qualified domain name *dname* . The reply message is left in the *answer* buffer with length *anslen* supplied by the caller. The *res_mkquery* and *res_send* routines described elsewhere in this section are lower-level routines that *res_query* calls.

`res_send` `res_send( )` sends a pre-formatted query to name servers and returns an answer. It calls `res_init( )` if `RES_INIT` is not set, send the query to the local name server, and handle timeouts and retries. *msg* is the query sent; *msglen* is its length. *answer* is the response returned. The length of the response is stored in *anslen* . `res_send( )` returns the length of the response or `-1` if there were errors.

`dn_expand` `dn_expand( )` expands the compressed domain name given by the pointer *comp_dn* into a full domain name. Expanded names are converted to upper case. The compressed name is contained in a query or reply message; *msg* is a pointer to the beginning of that message. Expanded names are stored in the buffer referenced by the *exp_dn* buffer of size *length* , which should be large enough to hold the expanded result.

`dn_expand( )` returns the size of the compressed name, or `-1` if there was an error.

`dn_comp` `dn_comp( )` compresses the domain name *exp_dn* and stores it in *comp_dn* . `dn_comp( )` 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* . *dnptrs* 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. However, if *lastdnptr* is `NULL` , `dn_comp( )` does not update the list of labels. If *dnptrs* is `NULL` , names are not compressed.

**res_mkupdrec** `res_mkupdrec( )` takes the elements of a resource record as its arguments, for instance, *section* , `domainname` , *class* , `type` , and *ttl* , and returns a pointer to a linked list `ns_updrec` . It returns `NULL` upon failure.

**res_update** `res_update( )` takes a linked list of resource records `ns_updrec` as its only argument and separates the records into groups such that all records in a group will belong to a single name server. It creates a dynamic update packet for each zone and sends it to the name server and awaits an answer. Upon success, `res_update( )` returns the number of zones updated. It returns `<0` upon error.

**res_mkupdate** `res_mkupdate( )` creates update packets by running through the elements of the `ns_updrec` link list. It is called by `res_update( )` to create packets for `res_send( )` to send to the name server. `res_mkupdate( )` returns the actual size of the packet, or

`-1` Error in reading a word or number in the `rdata` portion for update packets.

`-2` The length of the packet passed is insufficient.

`-3` The zone section is not the first section in the linked list, or the section order has a problem.

`-4` A number overflow.

`-5` Unknown operations or no records.

If an error occurs it could leave the zones being updated in an inconsistent state. For example, a record might be successfully added to the forward lookup zone but the corresponding PTR record might have failed to update in the reverse lookup tables.

## FILES

`/etc/resolv.conf`

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Unsafe |

## SEE ALSO

`domainname`(1M) , `in.named`(1M) , `nstest`(1M) , `gethostbyname`(3NSL) , `libresolv`(3LIB) , `resolv.conf`(4) , `attributes`(5)

Lottor, M., *Domain Administrators Operators Guide* , RFC 1033, SRI International, Menlo Park, Calif., November 1987.

Mockapetris, Paul, *Domain Names - Concepts and Facilities* , RFC 1034, Network Information Center, SRI International, Menlo Park, Calif., November 1987.

Mockapetris, Paul, *Domain Names - Implementation and Specification* , RFC 1035, Network Information Center, SRI International, Menlo Park, Calif., November 1987.

Partridge, Craig, *Mail Routing and the Domain System* , RFC 974, Network Information Center, SRI International, Menlo Park, Calif., January 1986. Stahl, M., *Domain Administrators Guide* , RFC 1032, SRI International, Menlo Park, Calif., November 1987.

Vixie, Paul;Dunlap, Keven J., Karels, Michael J., *Name Server Operations Guide for BIND* (public domain), Internet Software Consortium, 1996.

**NOTES**

These interfaces are unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

**NAME**

rexec, rexec_af – return stream to a remote command

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lsocket –lnsl` [ *library* ... ]

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**

`rexec( )` and `rexec_af( )` look up the host*ahost* using `getipnodebyname`(3SOCKET) , returning `-1` if the host does not exist. Otherwise *ahost* is set to the standard name of the host. If a username and password are both specified, then these are used to authenticate to the foreign host; otherwise the user's `.netrc` file in his home directory is searched for appropriate information. If all this fails, the user is prompted for the information.

The difference between `rexec( )` and `rexec_af( )` is that while `rexec( )` always returns a socket of the `AF_INET` address family, with `rexec_af( )` the application can choose which type of address family the socket returned should be. `rexec_af( )` supports both `AF_INET` and `AF_INET6` address families.

The port *inport* specifies which well-known DARPA Internet port to use for the connection. 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, then an auxiliary channel to a control process will be setup, and a file descriptor for it will be placed in **fd2p* . The control process will return diagnostic output (file descriptor 2, the standard error) from the command on this channel, and will also accept bytes on this channel as signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the standard error (file descriptor 2 of the remote command) will be made the same as its standard output and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

**RETURN VALUES**

If `rexec( )` succeeds, a file descriptor number, which is a socket of type `SOCK_STREAM` and address family `AF_INET` is returned by the routine. **ahost* is set to the standard name of the host, and if *fd2p* is not `NULL` , a file descriptor number is placed in **fd2p* which represents the command's standard error stream.

If `rexec_af( )` succeeds, the routine returns a filed descriptor number, which is a socket of type `SOCK_STREAM` and of address family type `AF_INET` or `AF_INET` , as determined by the value of the *af* parameter that the caller passes in.

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 |

**SEE ALSO**

`in.rexecd`(1M) , `gethostbyname`(3NSL) , `getipnodebyname`(3SOCKET) , `getservbyname`(3SOCKET) , `socket`(3SOCKET) , `attributes`(5)

**NOTES**

There is no way to specify options to the `socket()` call that `rexec()` or `rexec_af()` makes.

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

## 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 Considerations

In the case of multithreaded applications, the `_REENTRANT` flag must be defined on the command line at compilation time ( `–D_REENTRANT).` Defining this flag enables a thread-specific version of `rpc_createerr`. See `rpc_clnt_create`(3NSL).

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) |
| authkerb_getucred | kerberos_rpc(3KRB) |
| authkerb_seccreate | kerberos_rpc(3KRB) |
| 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) |
| `rac_drop` | `rpc_rac`(3RAC) |
| `rac_poll` | `rpc_rac`(3RAC) |
| `rac_recv` | `rpc_rac`(3RAC) |

| | |
|---|---|
| rac_send | rpc_rac(3RAC) |
| 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_kerb_reg | kerberos_rpc(3KRB) |
| 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), `kerberos_rpc`(3KRB), `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), `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

**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) . For the `Kerberos` style authentication, please refer to `kerberos_rpc`(3KRB) .

The `NULL` and `SYS` style of authentication are safe in multithreaded applications. For the MT-level of the `DES` and `Kerberos` styles, see their respective 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>
```

void auth_destroy(AUTH **auth* );
  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( )` .

AUTH *authnone_create(void);
  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.

AUTH *authsys_create(const char **host* , const uid_t *uid* , const gid_t *gid* , const int *len* , const gid_t **aup_gids* );
  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.

AUTH *authsys_create_default(void);
  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**

`kerberos_rpc`(3KRB) , `rpc`(3NSL) , `rpc_clnt_calls`(3NSL) , `rpc_clnt_create`(3NSL) , `secure_rpc`(3NSL) , `attributes`(5)

## NAME

rpc_clnt_calls, clnt_call, 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

## 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 in the case of errors.

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 (that is, 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.

```
#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* );

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.

bool_t clnt_freeres(CLIENT **clnt* , const xdrproc_t *outproc* , caddr_t *out* );

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, and `0` otherwise.

void clnt_geterr(const CLIENT **clnt* , struct rpc_err **errp* );

A function macro that copies the error structure out of the client handle to the structure at address *errp* .

void clnt_perrno(const enum clnt_stat *stat* );

Print a message to standard error corresponding to the condition indicated by *stat* . A newline is appended. Normally used after a procedure call fails for a routine for which a client handle is not needed, for instance `rpc_call().`

void clnt_perror(const CLIENT **clnt* , const char **s* );

Print 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. Normally used after a remote procedure call fails for a routine which requires a client handle, for instance `clnt_call( )` .

char *clnt_sperrno(const enum clnt_stat *stat* );

Take the same arguments as `clnt_perrno( )` , but instead of sending a message to the standard error indicating why an RPC call failed, return a pointer to a string which 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), or if the programmer does not want the message to be output with `printf( )` (see `printf`(3C) ), or if a message format different than that supported by `clnt_perrno( )` is to be used. Note: unlike `clnt_sperror( )` and `clnt_spcreaterror( )` (see `rpc_clnt_create`(3NSL) ), `clnt_sperrno( )` does not return pointer to static data so the result will not get overwritten on each call.

char *clnt_sperror(const CLIENT **clnt* , const char **s* );

Like `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.

Warning: returns pointer to a buffer that is overwritten on each call. In multithread applications, this buffer is implemented as thread-specific data.

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* );

Like `rpc_call( )` , except 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.

Warning: 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) ).

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* );

Like `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 re-transmission interval increases exponentially until it exceeds *waittime* .

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* );

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(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 an appropriate status is returned. Use the `clnt_perrno( )` routine to translate failure status into error messages.

Warning: `rpc_call( )` 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 |
|---|---|
| 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 – library routines for dealing with creation and manipulation of CLIENT handles

**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 `_REENTRANT` flag must be defined on the command line at compilation time ( `-D_REENTRANT ).` When the `_REENTRANT` flag is defined, `rpc_createerr` becomes a macro which enables each thread to have its own `rpc_createerr` . See `rpc`(3NSL) for the definition of the `CLIENT` data structure.

**Routines**

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

bool_t clnt_control(CLIENT **clnt* , const uint_t *req* , char **info* `);`

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 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.

CLIENT *clnt_create(const char **host* , const rpcprog_t *prognum* , const rpcvers_t *versnum* , const char **nettype* `);`

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: `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) ).

CLIENT *clnt_create_timed(const char **host* , const rpcprog_t *prognum* , const rpcvers_t *versnum* , const char **nettype* , const struct timeval **timeout* );

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.

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* , char **nettype* `);`

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.

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* );

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.

void clnt_destroy(CLIENT **clnt* `);`

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) ).

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 *recvsz* `);`

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.

void clnt_pcreateerror(const char **s* `);`

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.

CLIENT *clnt_raw_create(const rpcprog_t *prognum* , const rpcvers_t *versnum* );

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()` .

char *clnt_spcreateerror(const char **s* );

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.

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 *recvsz* );

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.

CLIENT *clnt_tp_create(const char **host* , const rpcprog_t *prognum* , const rpcvers_t *versnum* , const struct netconfig **netconf* `);`

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.

CLIENT *clnt_tp_create_timed(const char **host* , const rpcprog_t *prognum* , const rpcvers_t *versnum* , const struct netconfig **netconf* , const struct timeval **timeout* );

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.

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 *recvsz* );

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.

struct rpc_createerr 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` .

**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_clnt_auth`(3NSL) , `rpc_clnt_calls`(3NSL) , `rpc_svc_create`(3NSL) , `svc_raw_create`(3NSL) , `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 returns`FALSE` 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 |
|---|---|
| MT-Level | MT-Safe |
| Packages | SUNWrsg, SUNWrsgx |

**SEE ALSO**

`rpc`(3NSL), `rpc_gss_set_callback`(3NSL), `rpc_gss_set_svc_name`(3NSL), `rpcsec_gss`(3NSL), `attributes`(5)

*ONC+ Developer's Guide*

*Network Working Group RFC 2078*

**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 |
|---|---|
| MT-Level | MT-Safe |
| Packages | SUNWrsg, SUNWrsgx |

**SEE ALSO**

perror(3C), rpc(3NSL), rpcsec_gss(3NSL), attributes(5)

*ONC+ Developer's Guide*

*Network Working Group RFC 2078*

**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 (e.g., "kerberosv5") and a pointer to a `rpc_gss_service_t` enum. Returns a null-terminated list of character strings of supported Quality of Protections (QOPs) for this mechanism. |
| `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 |
|---|---|
| MT-Level | MT-Safe |
| Packages | SUNWrsg, SUNWrsgx |

**SEE ALSO**

`rpc`(3NSL) , `rpcsec_gss`(3NSL) , `mech`(4) , `qop`(4) , `attributes`(5)

*ONC+ Developer's Guide*

*Network Working Group RFC 2078*

**NOTES**

This function will change in a future release.

**BUGS**

The *service* argument for `rpc_gss_get_mech_info( )` is currently irrelevant.

**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, NIS, 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 |
|---|---|
| MT-Level | MT-Safe |
| Packages | SUNWrsg, SUNWrsgx |

**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*

*Network Working Group RFC 2078*

**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 |
|---|---|
| MT-Level | MT-Safe |
| Packages | SUNWrsg, SUNWrsgx |

**SEE ALSO**

`rpc`(3NSL) , `rpcsec_gss`(3NSL) , `attributes`(5)

*ONC+ Developer's Guide*

*Network Working Group RFC 2078*

**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 |
|---|---|
| MT-Level | MT-Safe |
| Packages | SUNWrsg, SUNWrsgx |

**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*

*Network Working Group RFC 2078*

**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 |
|---|---|
| MT-Level | MT-Safe |
| Packages | SUNWrsg, SUNWrsgx |

## 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*

*Network Working Group RFC 2078*

## NOTES

Contexts may be destroyed normally, with `auth_destroy()`. See `auth_destroy`(3NSL)

Currently, the `GSS-API` interface is not exposed. Therefore, use `NULL` for both the *options_req* and *options_ret* parameters.

**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 |
|---|---|
| MT-Level | MT-Safe |
| Packages | SUNWrsg, SUNWrsgx |

**SEE ALSO**

`rpc`(3NSL), `rpc_gss_getcred`(3NSL), `rpcsec_gss`(3NSL), `attributes`(5)

*ONC+ Developer’s Guide*

*Network Working Group RFC 2078*

**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 |
|---|---|
| MT-Level | MT-Safe |
| Packages | SUNWrsg, SUNWrsgx |

**SEE ALSO**

`rpc`(3NSL), `rpc_gss_get_mech_info`(3NSL), `rpcsec_gss`(3NSL), `qop`(4), `attributes`(5)

*ONC+ Developer's Guide*

*Network Working Group RFC 2078*

**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 |
|---|---|
| MT-Level | MT-Safe |
| Packages | SUNWrsg, SUNWrsgx |

**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

rpc_rac, rac_drop, rac_poll, rac_recv, rac_send – remote asynchronous calls

## SYNOPSIS

`cc` [ *flag* ... ] *file* ... `–lrac –lnsl` [ *library* ... ]
#include <rpc/rpc.h>
#include <rpc/rac.h>
void **rac_drop**(CLIENT **cl*, void **h*);

enum clnt_stat **rac_poll**(CLIENT **cl*, void **h*);

enum clnt_stat **rac_recv**(CLIENT **cl*, void **h*);

void ***rac_send**(CLIENT **cl*, rpcproc_t *proc*, xdrproc_t *xargs*, void **argsp*, xdrproc_t *xresults*, void **resultsp*, struct timeval *timeout*);

## DESCRIPTION

The remote asynchronous calls (RAC) package is a special interface to the RPC library that allows messages to be sent using the RPC protocol without blocking during the time between when the message is sent and the reply is received. To RPC servers, RAC messages are indistinguishable from RPC messages.

A client establishes an RPC session in the usual way (see `rpc_clnt_create`(3NSL) ). A RAC message is sent using `rac_send( )` . This routine returns immediately, allowing the client to conduct other processing. When the client wants to determine whether the returned value from the call has been received, `rac_poll( )` is used. `rac_recv( )` is used to collect the returned value; it can also be used to block while waiting for the returned value to arrive. `rac_drop( )` is used to inform the RPC library that the client is no longer interested in the results of a particular RAC message.

`rac_drop( )`
: `rac_drop( )` should be called when the user is no longer interested in the result of a `rac_send( )` currently in progress. No message to the server is generated by this call, but any subsequent reply received for this handle will be silently dropped. It also frees any space occupied by the asynchronous call handle *h* .

  After a call to `rac_drop( )` the handle referred to by *h* is invalid. It may no longer be used in any asynchronous operation.

`rac_poll( )`
: `rac_poll( )` returns the status of the call currently in progress on the <CLIENT, asynchronous handle> tuple referred to by *cl* and *h* .

  `rac_poll( )` return values are:

| | |
|---|---|
| `RPC_SUCCESS` | A reply has been received and is available for reading by `rac_recv( )` . |

| | |
|---|---|
| `RPC_INPROGRESS` | No reply has been received. The call referred to by the given handle has not yet timed out. |
| `RPC_TIMEDOUT` | No reply has been received. The call referred to by the given handle has exceeded the maximum timeout value specified in `rac_send()` . |
| `RPC_STALERACHANDLE` | Either the handle referred to by *h* is invalid or no call is currently in progress for the given <CLIENT, asynchronous handle> tuple. |
| `RPC_CANTRECV` | Either the file descriptor associated with the given CLIENT handle is bad, or an error occurred while attempting to receive a packet. |
| `RPC_SYSTEMERROR` | Space could not be allocated to receive a packet. |

On unreliable transports, a call to `rac_poll()` will trigger a retransmission when necessary (that is, if a `rac_send()` is in progress, no reply has been received, the per-call timeout has expired, and the total timeout has not yet expired).

The return value for `rac_poll()` is independent of the RPC return value in the reply packet. Although a combination of `clnt_control()` 's CLGET_FD request and `poll`(2) may be used to extract the proper file descriptor and poll for packets, `rac_poll()` is still useful since it will determine whether a reply is available for a specific <CLIENT, asynchronous handle> tuple.

`rac_recv()`

`rac_recv()` retrieves the results of a previous asynchronous RPC call, placing them in the buffer indicated in the `rac_send()` call and using the XDR decode function supplied there. It depends on the application to have ensured that a reply is present (using `rac_poll()` ). If `rac_recv()` is called before a reply has been received, it will block awaiting a reply.

All errors normally returned by the RPC client call functions may be returned here. In addition:

| | |
|---|---|
| `RPC_STALERACHANDLE` | Either the handle referred to by *h* is invalid or no call is currently in progress for the given <CLIENT, asynchronous handle> tuple. Additionally, if a packet is present and its status is not RPC_SUCCESS, it is possible that the client credentials need refreshing. In this case, RPC_AUTHERROR is returned and the client should attempt to resend the call. |

When a reply has been received, `rac_recv( )` will invoke the XDR decode procedure specified in the `rac_send( )` call. After a call to `rac_recv( )` , the handle referred to by *h* is invalid. It may no longer be used in any asynchronous operation.

`rac_send( )`

`rac_send( )` initiates (sends to the server) an RPC call to the specified procedure. It does not await a reply from the server. *argsp* is the address of the procedure's arguments, *resultsp* is the address in which to place the results, `xargs` and *xresults* are XDR functions used to encode and decode respectively. Note: *resultsp* must be a valid pointer when `rac_recv( )` is called. *timeout* should contain the total amount of time the application is willing to wait for a reply.

Upon success, an opaque handle, known as the asynchronous handle, is returned. This handle is to be used in subsequent asynchronous calls to poll for the status of the call (`rac_poll( )` ), receive the returned results of the call (`rac_recv( )` ), or cancel the call (`rac_drop( )` ).

On failure, *(void *) 0* is returned.

In case of failure, the application may retrieve the RPC failure code by calling `clnt_geterr( )` immediately after a `rac_send( )` failure (see `rpc`(3NSL) ). Possible errors include both transient problems (such as transport failures) and permanent ones (such as XDR encoding failures).

Multiple `rac_send` s on the same client handle are permitted, but may introduce unpredictable perturbations to the current timeout and retry model used by the RPC library.

The interface imposes a limit on the amount of time a call may be in progress before it is considered to have failed. This method was chosen over limitations on the number of retries because of a desire for transport independence.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Unsafe |

**SEE ALSO**

`poll`(2) , `rpc`(3NSL) , `rpc_clnt_create`(3NSL) , `rpc_clnt_calls`(3NSL) , `xdr`(3NSL) , `attributes`(5)

**WARNINGS**

The RAC interface is not the recommended interface for having multiple RPC requests outstanding. The preferred method of accomplishing this in the Solaris environment is to use synchronous RPC calls with threads. The RAC interface is provided as a service to developers interested in porting RPC applications to Solaris 2.0. Use of this interface will degrade the performance of normal synchronous RPC calls (see `rpc_clnt_calls`(3NSL) ). For these reasons, use of this interface is disparaged.

The library `librac` must be linked before `libnsl` to use RAC. If the libraries are not linked in the correct order, then the results are indeterminate.

**NOTES**

These interfaces are unsafe in multithreaded applications. Unsafe interfaces should be called only from the main thread.

**NAME**

rpcsec_gss – security flavor incorporating

**SYNOPSIS**

cc [ *flag...* ] *file...*– lnsl [ *library...* ]#include <rpc/rpcsec_gss.h>
(void);

**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_DES, 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. (Because the GSS-API is not currently exposed, this data type is mentioned for informational purposes only. The programmer should set this to `NULL`.

```
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. (Because the GSS-API is not currently exposed, this data type is mentioned for informational purposes only. Set this to `NULL` in order to use default values.)

```
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 |
|---|---|
| MT-Level | MT-Safe |
| Packages | SUNWrsg, SUNWrsgx |

## SEE ALSO

`gsscred`(1M), `rpc`(3NSL), `rpc_clnt_auth`(3NSL), `xdr`(3NSL), `attributes`(5), `environ`(5)

*ONC+ Developer's Guide*

*Network Working Group RFC 2078*

**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

**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.

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

AUTH *authdes_create(char **name* , uint_t *window* , struct sockaddr_in **syncaddr* , des_block **ckey* );

`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.

Warning: this routine exists for backward compatibility only, and is obsoleted by `authdes_seccreate( )` (see `secure_rpc`(3NSL) ).

AUTH *authunix_create(char **host* `,` uid_t *uid* , gid_t *gid* , int *grouplen* , gid_t **gidlistp* `);`

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.

Warning: it is not very difficult to impersonate a user.

Warning: this routine exists for backward compatibility only, and is obsoleted by `authsys_create( )` (see `rpc_clnt_auth`(3NSL) ).

AUTH *authunix_create_default(void)

Call `authunix_create( )` with the appropriate parameters.

Warning: this routine exists for backward compatibility only, and is obsoleted by `authsys_create_default( )` (see `rpc_clnt_auth`(3NSL) ).

callrpc(char **host* , rpcprog_t *prognum* , rpcvers_t *versnum* , rpcproc_t *procnum* , xdrproc_t *inproc* , char **in* , xdrproc_t *outproc* , char **out* );

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( )` (see `rpc_clnt_calls`(3NSL) ) is handy for translating failure statuses into messages.

Warning: you do not have control of timeouts or authentication using this routine. This routine exists for backward compatibility only, and is obsoleted by `rpc_call( )` (see `rpc_clnt_calls`(3NSL) ).

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* );

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.

Warning: 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) ).

Warning: this routine exists for backward compatibility only, and is obsoleted by `rpc_broadcast( )` (see `rpc_clnt_calls`(3NSL) ).

CLIENT *clntraw_create(rpcprog_t *prognum* , rpcvers_t *versnum* );

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.

Warning: this routine exists for backward compatibility only, and has the same functionality as `clnt_raw_create( )` (see `rpc_clnt_create`(3NSL) ), which obsoletes it.

CLIENT *clnttcp_create(struct sockaddr_in **addr* , rpcprog_t *prognum* , rpcvers_t *versnum* , int **fdp* , uint_t *sendsz* , uint_t *recvsz* );

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.

Warning: this routine exists for backward compatibility only. `clnt_create( )` , `clnt_tli_create( )` , or `clnt_vc_create( )` (see `rpc_clnt_create`(3NSL) ) should be used instead.

`CLIENT *clntudp_bufcreate(struct sockaddr_in *` *addr* , rpcprog_t *prognum* , rpcvers_t *versnum* , struct timeval *wait* , int **fdp,* uint_t *sendsz* , uint_t *recvsz* );

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( )` (see `rpc_clnt_create`(3NSL) ) routine.

The user can specify the maximum packet size for sending and receiving by using *sendsz* and *recvsz* arguments for UDP-based RPC messages.

Warning: 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) ).

Warning: this routine exists for backward compatibility only. `clnt_tli_create( )` or `clnt_dg_create( )` (see `rpc_clnt_create`(3NSL) ) should be used instead.

CLIENT *clntudp_create(struct sockaddr_in **addr* , rpcprog_t *prognum* , rpcvers_t *versnum* , struct timeval *wait* , int **fdp* );

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()` (see `rpc_clnt_create`(3NSL) ) routine.

Warning: 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.

Warning: this routine exists for backward compatibility only. `clnt_create()` , `clnt_tli_create()` , or `clnt_dg_create()` (see `rpc_clnt_create`(3NSL) ) should be used instead.

void get_myaddress(struct sockaddr_in **addr* );

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)` .

Warning: 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.

Warning: this routine remains for backward compatibility only. The routine `netdir_getbyname()` (see `netdir`(3NSL) ) should be used with the name `HOST_SELF` to retrieve the local system's network address as a *netbuf* structure.

u_short getrpcport(char **host* , rpcprog_t *prognum* , rpcvers_t *versnum* , rpcprot_t *proto* );

`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.

Warning: This routine exists for backward compatibility only. Enhanced functionality is provided by `rpcb_getaddr( )` (see `rpcbind`(3NSL) ).

struct pmaplist *pmap_getmaps(struct sockaddr_in **addr* );

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.

Warning: this routine exists for backward compatibility only, enhanced functionality is provided by `rpcb_getmaps( )` (see `rpcbind`(3NSL) ).

u_short pmap_getport(struct sockaddr_in **addr* , rpcprog_t *prognum* , rpcvers_t *versnum* , rpcprot_t *protocol* );

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.

Warning: this routine exists for backward compatibility only, enhanced functionality is provided by `rpcb_getaddr( )` (see `rpcbind`(3NSL) ).

enum clnt_stat pmap_rmtcall(struct sockaddr_in **addr* , rpcprog_t *prognum* , rpcvers_t *versnum* , rpcproc_t *procnum* , caddr_t *in* , xdrproct_t *inproc* , caddr_t *out* , xdrproct_t *outproc* , struct timeval *tout* , rpcport_t **portp* );

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) ).

Note: this procedure is only available for the UDP transport.

Warning: 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.

Warning: this routine exists for backward compatibility only, enhanced functionality is provided by `rpcb_rmtcall( )` (see `rpcbind`(3NSL) ).

bool_t pmap_set(rpcprog_t *prognum* , rpcvers_t *versnum* , rpcprot_t *protocol* , u_short *port* );

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( )` .

Warning: this routine exists for backward compatibility only, enhanced functionality is provided by `rpcb_set( )` (see `rpcbind`(3NSL) ).

bool_t pmap_unset(rpcprog_t *prognum* , rpcvers_t *versnum* );

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.

Warning: this routine exists for backward compatibility only, enhanced functionality is provided by `rpcb_unset( )` (see `rpcbind`(3NSL) ).

int 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 (do not pass its address to `select( )` !), yet it may change after calls to `svc_getreq( )` or any creation routines. Similar to `svc_fdset` , but limited to 32 descriptors.

Warning: this interface is obsoleted by `svc_fdset` (see `rpc_svc_calls`(3NSL) ).

struct sockaddr_in * svc_getcaller(SVCXPRT **xprt* );

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* .

Warning: 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` .

void svc_getreq(int *rdfds* );

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.

Warning: this interface is obsoleted by `svc_getreqset().`

SVCXPRT *svcfd_create(int *fd* , uint_t *sendsz* , uint_t *recvsz* );

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.

Warning: this interface is obsoleted by `svc_fd_create()` (see `rpc_svc_create`(3NSL) ).

SVCXPRT *svcraw_create(void);

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.

Warning: this routine exists for backward compatibility only, and has the same functionality of `svc_raw_create()` (see `rpc_svc_create`(3NSL) ), which obsoletes it.

SVCXPRT *svctcp_create(int *fd* , uint_t *sendsz* , uint_t *recvsz* `);`

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.

Warning: this routine exists for backward compatibility only. `svc_create()` , `svc_tli_create()` , or `svc_vc_create()` (see `rpc_svc_create`(3NSL) ) should be used instead.

SVCXPRT *svcudp_bufcreate(int *fd* , uint_t *sendsz* , uint_t *recvsz* );

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.

Warning: this routine exists for backward compatibility only. `svc_tli_create( )` , or `svc_dg_create( )` (see `rpc_svc_create`(3NSL) ) should be used instead.

SVCXPRT *svcudp_create(int *fd* );

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.

Warning: 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.

Warning: this routine exists for backward compatibility only. `svc_create( )` , `svc_tli_create( )` , or `svc_dg_create( )` (see `rpc_svc_create`(3NSL) ) should be used instead.

registerrpc(rpcprog_t *prognum* , rpcvers_t *versnum* , rpcproc_t *procnum* , char *(**procname* )(), xdrproc_t *inproc* , xdrproc_t *outproc* );

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.

Warning: this routine exists for backward compatibility only, and is obsoleted by `rpc_reg( )` .

bool_t svc_register(SVCXPRT **xprt* , rpcprog_t *prognum* `,` rpcvers_t *versnum* , void (**dispatch* )(), int *protocol* );

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.

Warning: this routine exists for backward compatibility only; enhanced functionality is provided by svc_reg( ) .

void svc_unregister(rpcprog_t *prognum* , rpcvers_t *versnum* );
Remove all mapping of the double [*prognum* , *versnum* ] to dispatch routines, and of the triple [*prognum* , *versnum* , *all-protocols* ] to port number from portmap .

Warning: this routine exists for backward compatibility, enhanced functionality is provided by svc_unreg( ) .

bool_t xdr_authunix_parms(XDR **xdrs* , struct authunix_parms **aupp* );
Used for describing UNIX credentials. This routine is useful for users who wish to generate these credentials without using the RPC authentication package.

Warning: this routine exists for backward compatibility only, and is obsoleted 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) , librpcsoc(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 – library routines for RPC servers

**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, while others (such as `svc_run( )` ) are called when the server is initiated.

In the current implementation, the service transport handle SVCXPRT contains a single data area for decoding arguments and encoding results. Therefore, this structure cannot be freely shared between threads that call functions that do this. However, when a server is operating in the Automatic or User MT modes, 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 MT applications, and are not to be used by such applications.

**Routines**

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

int svc_dg_enablecache(SVCXPRT **xprt* , const uint_t *cache_size* );

This 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. This routine returns `1` if space necessary for a cache of the given size was successfully allocated, and `0` otherwise.

This function is safe in MT applications.

int svc_done(SVCXPRT **xprt* );

This 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 MT applications. It will have no effect if invoked in modes other than the User MT mode.

void svc_exit(void);

This function when called by any of the RPC server procedure 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) .

`svc_exit` () has global scope and ends all RPC server activity.

fd_set svc_fdset;

A global variable reflecting 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, and it may change after calls to `svc_getreqset( )` or any creation routines. Do not pass its address to `select`(3C) ! Instead, pass the address of a copy.

MT 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.

`svc_fdset` is limited to 1024 file descriptors and is considered obsolete. Use of `svc_pollfd` is recommended instead.

pollfd_t *svc_pollfd;

A global variable pointing to an array of `pollfd_t` structures reflecting 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.

MT 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.

int svc_max_pollfd;

A global variable containing 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.

bool_t svc_freeargs(const SVCXPRT **xprt* , const xdrproc_t *inproc* , caddr_t *in* );

A function macro that 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 MT applications utilizing the Automatic or User MT modes.

bool_t svc_getargs(const SVCXPRT **xprt* , const xdrproc_t *inproc* , caddr_t *in* );
A function macro that 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 MT applications utilizing the Automatic or User MT modes.

void svc_getreq_common(const int *fd* );
This routine is called to handle a request on the given file descriptor.

void svc_getreq_poll(struct pollfd **pfdp* , const int *pollretval* );
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 `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.

This function macro is unsafe in MT applications.

void svc_getreqset(fd_set **rdfds* );
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 `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 MT applications.

struct netbuf *svc_getrpccaller(const SVCXPRT **xprt* );
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 MT applications.

void svc_run(void);
This routine never returns. In single threaded mode, it waits for RPC requests to arrive, and calls the appropriate service procedure using

`svc_getreq_poll()` when one arrives. This procedure is usually waiting for the `poll`(2) library call to return.

Applications executing in the Automatic or User MT modes should invoke this function exactly once. It the Automatic MT mode, it will create threads to service client requests. In the User MT mode, it will provide a framework for service developers to create and manage their own threads for servicing client requests.

bool_t svc_sendreply(const SVCXPRT **xprt* , const xdrproc_t *outproc* , const caddr_t *out* );

Called by an RPC service's dispatch routine to send the results of a remote procedure call. The parameter *xprt* is the request's associated transport handle; *outproc* is the XDR routine which is used to encode the results; and *out* is the address of the results. This routine returns `TRUE` if it succeeds, `FALSE` otherwise.

This function macro is safe in MT applications utilizing the Automatic or User MT modes.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | See `NOTES` below. |

**SEE ALSO**

`rpcgen`(1) , `poll`(2) , `rpc`(3NSL) , `rpc_control`(3NSL) , `rpc_svc_create`(3NSL) , `rpc_svc_err`(3NSL) , `rpc_svc_reg`(3NSL) , `select`(3C) , `xprt_register`(3NSL) , `attributes`(5)

**NOTES**

`svc_dg_enablecache()` and `svc_getrpccaller()` are safe in multithreaded applications. `svc_freeargs()` , `svc_getargs()` , and `svc_sendreply()` are safe in MT applications utilizing the Automatic or User MT modes. `svc_getreq_common()` , `svc_getreqset()` , and `svc_getreq_poll()` are unsafe in multithreaded applications and should be called only from the main thread.

**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 – library routines for the creation of server handles

**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*);

**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 various 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 (vc) and connectionless (dg) 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 transation 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_CONNMAXREC`

Set the maximum record size, in bytes, for RPC requests and enable non-blocking mode for this service handle. The value can be set and read for both connection and non-connection oriented transports, but it is silently ignored for the non-connection oriented case. *info* is a pointer to an argument of type `int` .

`SVCGET_CONNMAXREC`
Get the maximum RPC request record size for this service handle. Zero means no maximum is in effect, and the connection is in blocking mode. The result is not significant for non-connection oriented transports. *info* is a pointer to an argument of type `int` .

`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.

## 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_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_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 => nc_netid]* 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

## 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>
```

bool_t xdr_accepted_reply(XDR **xdrs* , const struct accepted_reply **ar* );
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.

bool_t xdr_authsys_parms(XDR **xdrs* , struct authsys_parms **aupp* );
Used for describing UNIX operating system credentials. It includes machine-name, uid, gid list, etc.

void xdr_callhdr(XDR **xdrs* , struct rpc_msg **chdr* );
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.

bool_t xdr_callmsg(XDR **xdrs* , struct rpc_msg **cmsg* );
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.

bool_t xdr_opaque_auth(XDR **xdrs* , struct opaque_auth **ap* );
Used for describing RPC opaque authentication information messages.

bool_t xdr_rejected_reply(XDR **xdrs* , const struct rejected_reply **rr* );
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.

bool_t xdr_replymsg(XDR **xdrs* , const struct rpc_msg **rmsg* );
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

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

## DESCRIPTION

RPC library routines allow C programs to make procedure calls on other machines across the network.

RPC supports various authentication flavors. Among them are:

| | |
|---|---|
| `AUTH_NONE` | (none) no authentication. |
| `AUTH_SYS` | Traditional UNIX-style authentication. |
| `AUTH_DES` | DES encryption-based authentication. |
| `AUTH_KERB` | Kerberos encryption-based authentication. |

The `authdes_getucred( )` and `authdes_seccreate( )` routines implement the `AUTH_DES` authentication flavor. The keyserver daemon `keyserv` (see `keyserv`(1M) ) must be running for the `AUTH_DES` authentication system to work, and `keylogin`(1) must have been run. Only the `AUTH_DES` style of authentication is discussed here. For information about the `AUTH_NONE` and `AUTH_SYS` styles of authentication, refer to `rpc_clnt_auth`(3NSL) . For information about the `AUTH_KERB` style of authentication, refer to `kerberos_rpc`(3KRB) .

The routines documented on this page are MT-Safe. See the pages of the other authentication styles for their MT-level.

### Routines

See `rpc`(3NSL) for the definition of the `AUTH` data structure.

```
#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* `);`

`authdes_getucred( )` is the first of the two routines which interface to the RPC secure authentication system known as `AUTH_DES` . The second is `authdes_seccreate( )` , below. `authdes_getucred( )` is used on the server side for converting an `AUTH_DES` credential, which is operating system independent, into an `AUTH_SYS` credential. This routine returns `1` if it succeeds, `0` if it fails.

* *uidp* is set to the user's numerical ID associated with *adc* . * *gidp* is set to the numerical ID of the user's group. * *gidlist* contains the numerical IDs of the other groups to which the user belongs. * *gidlenp* is set to the number of valid group ID entries in * *gidlist* (see `netname2user( )` , below).

Warning: `authdes_getucred( )` will fail if the authdes_cred structure was created with the netname of a host. In such a case, `netname2host( )`

should be used on the host netname in the authdes_cred structure to get the host name.

AUTH *authdes_seccreate(const char **name* ,const uint_t *window* , const char **timehost* ,const des_block **ckey* );

`authdes_seccreate( )` , the second of two `AUTH_DES` authentication routines, is used on the client side to return 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( )` , described below.

The second field is *window* on the validity of the client credential, given in seconds. If the difference in time between the client's clock and the server's clock exceeds *window* , the server will reject 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 of a window will increase the frequency of resynchronizations because of clock drift.

The third parameter, *timehost* , the host's name, is optional. If it is `NULL` , then the authentication system will assume that the local clock is always in sync with the *timehost* clock, and will not attempt resynchronizations. If a timehost is supplied, however, then the system will consult with the remote time service whenever resynchronization is required. This parameter is usually the name of the host on which the server is running.

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 *ckey* is supplied, then it will be used instead.

If `authdes_seccreate( )` fails, it returns `NULL.`

`int getnetname(char` *name* `[MAXNETNAMELEN+1]);`

`getnetname( )` returns the unique, operating system independent netname of the caller in the fixed-length array *name* . Returns `1` if it succeeds, and `0` if it fails.

`int host2netname(char` *name* `[MAXNETNAMELEN+1], const char *` *host* `, const char *` *domain* `);`

Convert from a domain-specific hostname *host* to an operating system independent netname. Returns `1` if it succeeds, and `0` if it fails. Inverse of `netname2host( )` . If *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 a NIS name like "host1.ssi.sun.com," `host2netname( )` uses the domain "ssi.sun.com" rather than the default domain name of the machine.

`int key_decryptsession(const char *` *remotename* `, des_block *` *deskey* `);`

`key_decryptsession( )` 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 it, or its associated routines `key_encryptsession( )` , `key_gendes( )` , and `key_setsecret( )` .

`key_decryptsession( )` takes a server netname *remotename* and a DES key *deskey* , and decrypts the key by using the the public key of the the server and the secret key associated with the effective UID of the calling process. It is the inverse of `key_encryptsession( )` .

`int key_encryptsession(const char *` *remotename* `, des_block *` *deskey* `);`

`key_encryptsession( )` is a keyserver interface routine. It takes a server netname *remotename* and a DES key *deskey* , and encrypts it using the public key of the the server and the secret key associated with the effective UID of the calling process. It is the inverse of `key_decryptsession( )` . This routine returns `0` if it succeeds, `-1` if it fails.

`int key_gendes(des_block *` *deskey* `);`

`key_gendes( )` is a keyserver interface routine. It is used to ask the keyserver for a secure conversation key. Choosing one at random is usually not good enough, because the common ways of choosing random numbers, such as using the current time, are very easy to guess. This routine returns `0` if it succeeds, `-1` if it fails.

`int key_setsecret(const char *` *key* `);`

`key_setsecret( )` is a keyserver interface routine. It is used to set the key for the effective UID of the calling process. This routine returns `0` if it succeeds, `-1` if it fails.

`int key_secretkey_is_set(void);`

`key_secretkey_is_set( )` is a keyserver interface routine that may be used to determine whether 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, this routine returns `1` . Otherwise it returns `0` .

`int netname2host(const char *` *name* `, char *` *host* `, const int` *hostlen* `);`

Convert from an operating system independent netname *name* to a domain-specific hostname *host* . *hostlen* is the maximum size of *host* . Returns `1` if it succeeds, and `0` if it fails. Inverse of `host2netname( )` .

`int netname2user(const char *` *name* `, uid_t *` *uidp* `, gid_t *` *gidp* `, int *` *gidlenp* `, gid_t` *gidlist* `[NGRPS]);`

Convert from an operating system independent netname to a domain-specific user ID. Returns `1` if it succeeds, and `0` if it fails. Inverse of `user2netname()` .

`*` *uidp* is set to the user's numerical ID associated with *name* . `*` *gidp* is set to the numerical ID of the user's group. *gidlist* contains the numerical IDs of the other groups to which the user belongs. `*` *gidlenp* is set to the number of valid group ID entries in *gidlist* .

`int user2netname(char` *name* `[MAXNETNAMELEN+1], const uid_t` *uid* `, const char *` *domain* `);`

Convert from a domain-specific username to an operating system independent netname. Returns `1` if it succeeds, and `0` if it fails. Inverse of `netname2user()` .

## 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) , `kerberos_rpc`(3KRB) , `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/types.h>

#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*, int *tolen*);

ssize_t **sendmsg**(int *s*, const struct msghdr **msg*, int *flags*);

**DESCRIPTION**

`send()` , `sendto()` , and `sendmsg()` are used to transmit a message to another transport end-point. `send()` may be used only when the socket is in a *connected* state, while `sendto()` and `sendmsg()` may be used at any time. *s* is a socket created with `socket`(3SOCKET) .

The address of the target is given by *to* with *tolen* specifying its size. The length of the message is given by *len* . If the message is too long to pass atomically through the underlying protocol, then the error `EMSGSIZE` is returned, and the message is not transmitted.

A return value of -1 indicates locally detected errors only. It does not implicitly mean the message was not delivered.

If the socket does not have enough buffer space available to hold the message being sent, `send()` blocks, unless the socket has been placed in non-blocking I/O mode (see `fcntl`(2) ). The `select`(3C) or `poll`(2) call may 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` and `AF_INET` address families 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**

These calls return the number of bytes sent, or -1 if an error occurred.

**ERRORS**

The calls fail if:

| | |
|---|---|
| `EBADF` | *s* is an invalid file descriptor. |

| | |
|---|---|
| `EINTR` | The operation was interrupted by delivery of a signal before any data could be buffered to be sent. |
| `EINVAL` | *tolen* is not the size of a valid address for the specified address family. |
| `EMSGSIZE` | The socket requires that message be sent atomically, and the message was too long. |
| `ENOMEM` | There was insufficient memory available to complete the operation. |
| `ENOSR` | There were insufficient STREAMS resources 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. |

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

**SEE ALSO**

`fcntl`(2) , `poll`(2) , `write`(2) , `connect`(3SOCKET) , `getsockopt`(3SOCKET) , `recv`(3SOCKET) , `select`(3C) , `socket`(3SOCKET) , `attributes`(5) , `socket`(3HEAD)

## 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);
```

## DESCRIPTION

| | |
|---|---|
| *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. |

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<br>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 process. |

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 |
|---|---|
| MT-Level | MT-Safe |

## 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)

**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*);

**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 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. |

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 process. |

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 |
|---|---|
| MT-Level | MT-Safe |

**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)

## 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.

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. |

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.

**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 can not 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 process. |

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 |
|---|---|
| MT-Level | MT-Safe |

**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)

**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 follows:

| | |
|---|---|
| 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

| | |
|---|---|
| | processes 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. |

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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`bind`(3XNET), `endprotoent`(3XNET), `getsockopt`(3XNET), `socket`(3XNET), `attributes`(5)

**NAME**

shutdown – shut down part of a full-duplex connection

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lsocket –lnsl` [ *library* ... ]

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 `0`, then further receives will be disallowed. If *how* is `1`, then further sends will be disallowed. If *how* is `2`, then further sends and receives will be disallowed.

**RETURN VALUES**

A 0 is returned if the call succeeds, –1 if it fails.

**ERRORS**

The call succeeds unless:

| | |
|---|---|
| `EBADF` | *s* is not a valid file descriptor. |
| `ENOMEM` | There was insufficient user memory available for the operation to complete. |
| `ENOSR` | There were insufficient STREAMS resources available for the operation to complete. |
| `ENOTCONN` | The specified socket is not connected. |
| `ENOTSOCK` | *s* 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), `attributes`(5), `socket`(3HEAD)

**NOTES**

The *how* values should be defined constants.

**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**

| | |
|---|---|
| *socket* | Specifies the file descriptor of the socket. |
| *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. |

The `shutdown( )` function disables subsequent send and/or receive operations on a socket, depending on the value of the *how* argument.

**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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`getsockopt`(3XNET), `recv`(3XNET), `recvfrom`(3XNET), `recvmsg`(3XNET), `select`(3C), `send`(3XNET), `sendto`(3XNET), `setsockopt`(3XNET), `socket`(3XNET), `attributes`(5)

**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). |
| Maximum results | The user has set the `net.slp.maxResults` property and that number of replies has been collected and returned. |

**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.<br><br>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* | Memory 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 | SUNWslpu |
| CSI | CSI-enabled |
| Interface Stability | Standard |
| MT-Level | Safe |

**SEE ALSO**

`slpd`(1M), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

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

```
#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 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 | SUNWslpu |

## SEE ALSO

`slpd`(1M), `slp_api`(3SLP), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

Kempf, J. and Guttman, E., *RFC 2614, An API for Service Location*, The Internet Society, June 1999.

**NAME**

SLPDelAttrs – delete attributes

**SYNOPSIS**

#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 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 | SUNWslpu |

**SEE ALSO**

`slpd`(1M), `slp_api`(3SLP), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

Kempf, J. and Guttman, E., *RFC 2614, An API for Service Location*, The Internet Society, June 1999.

**NAME**

SLPDereg – deregister the SLP advertisement

**SYNOPSIS**

#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 URL to 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 | SUNWslpu |

**SEE ALSO**

`slpd`(1M), `slp_api`(3SLP), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

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**

#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 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 | SUNWslpu |

**SEE ALSO**

`slpd`(1M), `slp_api`(3SLP), `SLPFree`(3SLP), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

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**

#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",
     "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 | SUNWslpu |

## SEE ALSO

`slpd`(1M), `slp_api`(3SLP), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

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

#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 | SUNWslpu |

**SEE ALSO**

`slpd`(1M), `slp_api`(3SLP), `SLPFree`(3SLP), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

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**

#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;

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 | SUNWslpu |

## SEE ALSO

`slpd`(1M), `slp_api`(3SLP), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

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

#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:

```
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 | SUNWslpu |

## SEE ALSO

`slpd`(1M), `slp_api`(3SLP), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

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**

```
#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 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 | SUNWslpu |

**SEE ALSO**

`slpd`(1M), `SLPEscape`(3SLP), `SLPFindScopes`(3SLP), `SLPParseSrvURL`(3SLP), `SLPUnescape`(3SLP ), `slp_api`(3SLP), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

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**

```
#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 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 | SUNWslpu |

**SEE ALSO**

`slpd`(1M), `slp_api`(3SLP), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

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**

```
#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 | SUNWslpu |

**SEE ALSO**

`slpd`(1M), `slp_api`(3SLP), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

Kempf, J. and Guttman, E., *RFC 2614, An API for Service Location*, The Internet Society, June 1999.

**NAME**

SLPOpen – open an SLP handle

**SYNOPSIS**

#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 | SUNWslpu |

**SEE ALSO**

`slpd`(1M), `slp_api`(3SLP), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

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**

#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 | SUNWslpu |

**SEE ALSO**

`slpd`(1M), `slp_api`(3SLP), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

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**

#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 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 | SUNWslpu |

**SEE ALSO**

`slpd`(1M), `slp_api`(3SLP), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

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

#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 | SUNWslpu |

## SEE ALSO

`slpd`(1M), `slp_api`(3SLP), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

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 | SUNWslpu |

## SEE ALSO

`slpd`(1M), `slp_api`(3SLP), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

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**

#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 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 | SUNWslpu |

**SEE ALSO**

`slpd`(1M), `SLPFree`(3SLP), `slp_api`(3SLP), `slp.conf`(4), `slpd.reg`(4), `attributes`(5)

*Service Location Protocol Administration Guide*

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**

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**

`socket()` creates an endpoint for communication and returns a descriptor.

The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file `<sys/socket.h>`. There must be an entry in the `netconfig`(4) file for at least each protocol family and type required. If *protocol* has been specified, but no exact match for the tuplet family, type, protocol is found, then the first entry containing the specified family and type with zero for protocol will be used. The currently understood formats are:

| | |
|---|---|
| `PF_UNIX` | UNIX system internal protocols |
| `PF_INET` | Internet Protocol Version 4 (IPv4) |
| `PF_INET6` | Internet Protocol Version 6 (IPv6) |

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
```

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 the superuser, and `SOCK_RDM`, for which no implementation currently exists, are not described here.

*protocol* specifies a particular protocol to be used with the socket. Normally 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 must be specified in this manner. The protocol number to use is particular to the “communication domain” in which communication is to take place. If a protocol is specified by the caller, then it will be packaged into a socket level option request and sent to the underlying protocol layers.

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 process 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 may also enable non-blocking I/O and asynchronous notification of I/O events with `SIGIO` signals.

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**

A `−1` is returned if an error occurs. Otherwise the return value is a descriptor referencing the socket.

**ERRORS**

The `socket( )` call fails if:

| | |
|---|---|
| `EACCES` | Permission to create a socket of the specified type and/or protocol is denied. |
| `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. |
| `EPROTONOSUPPORT` | The protocol type or the specified protocol is not supported within this domain. |

**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), `listen`(3SOCKET), `recv`(3SOCKET), `setsockopt`(3SOCKET), `send`(3SOCKET), `shutdown`(3SOCKET), `socketpair`(3SOCKET), `attributes`(5), `in`(3HEAD), `socket`(3HEAD)

**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 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.

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.

**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 |
|---|---|
| MT-Level | MT-Safe |

**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)

**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 *d*, of the specified `type`, and using the optionally specified *protocol*. The descriptors 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. |
| `EOPNOSUPPORT` | The specified protocol does not support creation of socket pairs. |
| `EPROTONOSUPPORT` | The specified protocol is not supported on this machine. |

**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), `attributes`(5), `socket`(3HEAD)

**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`.

| | |
|---|---|
| *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. |

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.

**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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO** `socket`(3XNET), `attributes`(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 */
  }
  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.

For more information refer to the *Transport Interfaces Programming Guide*

## 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)

*Transport Interfaces Programming Guide*

## 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)).

For more information refer to the *Transport Interfaces Programming Guide*

## 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.

For more information refer to the *Transport Interfaces Programming Guide*

## 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`

For more information refer to the *Transport Interfaces Programming Guide*

## 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)

*Transport Interfaces Programming Guide*

## 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.

For more information refer to the *Transport Interfaces Programming Guide*

## 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`

*Transport Interfaces Programming Guide*

**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
```

For more information refer to the *Transport Interfaces Programming Guide*

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`attributes`(5)

*Transport Interfaces Programming Guide*

## 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`

For more information refer to the *Transport Interfaces Programming Guide*

**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)

*Transport Interfaces Programming Guide*

**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`

For more information refer to the *Transport Interfaces Programming Guide*

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT Level | Safe |

## SEE ALSO

`t_alloc`(3NSL), `attributes`(5)

*Transport Interfaces Programming Guide*

**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.

For more information refer to the *Transport Interfaces Programming Guide*

**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)

*Transport Interfaces Programming Guide*

**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)

*Transport Interfaces Programming Guide*

**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`

For more information refer to the *Transport Interfaces Programming Guide*

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT Level | Safe |

## SEE ALSO

`t_open`(3NSL), `attributes`(5)

*Transport Interfaces Programming Guide*

## 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 Values**

The `t_errno` values `TPROTO`, `TBADQLEN`, and `TQFULL` 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.

For more information refer to the *Transport Interfaces Programming Guide*

## 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)

*Transport Interfaces Programming Guide*

## 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 Values

The `t_errno` value that can be set by the XTI interface and cannot be set by the TLI interface is:
`TPROTO`

For more information refer to the *Transport Interfaces Programming Guide*

## 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)

*Transport Interfaces Programming Guide*

**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.

For more information refer to the *Transport Interfaces Programming Guide*

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT Level | Safe |

## SEE ALSO

`open`(2), `attributes`(5)

*Transport Interfaces Programming Guide*

**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.<br><br>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.

For more information refer to the *Transport Interfaces Programming Guide*

## 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)

*Transport Interfaces Programming Guide*

## 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 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 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. |
| 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

For more information refer to the *Transport Interfaces Programming Guide*

**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_snd`(3NSL), `attributes`(5)

*Transport Interfaces Programming Guide*

**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.

For more information refer to the *Transport Interfaces Programming Guide*

## 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)

*Transport Interfaces Programming Guide*

## 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.

For more information refer to the *Transport Interfaces Programming Guide*

## 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)

*Transport Interfaces Programming Guide*

## 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`

For more information refer to the *Transport Interfaces Programming Guide*

## 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)

*Transport Interfaces Programming Guide*

## 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)

*Transport Interfaces Programming Guide*

## 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.

For more information refer to the *Transport Interfaces Programming Guide*

## 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)

*Transport Interfaces Programming Guide*

**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.

For more information refer to the *Transport Interfaces Programming Guide*

## 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)

*Transport Interfaces Programming Guide*

## 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)

*Transport Interfaces Programming Guide*

**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 "*iov*`0`" 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)

*Transport Interfaces Programming Guide*

## 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.

For more information refer to the *Transport Interfaces Programming Guide*

## 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)

*Transport Interfaces Programming Guide*

## 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.

For more information refer to the *Transport Interfaces Programming Guide*

## 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)

*Transport Interfaces Programming Guide*

**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 Values**

The `t_errno` values that can be set by the XTI interface and cannot be set by the TLI interface are:

`TPROTO`
`TLOOK`
`TOUTSTATE`

**Notes**

Whenever this function fails with `t_error` set to `TFLOW`, `O_NONBLOCK` must have been set.

For more information refer to the *Transport Interfaces Programming Guide*

## 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)

*Transport Interfaces Programming Guide*

**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)

*Transport Interfaces Programming Guide*

**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.

For more information refer to the *Transport Interfaces Programming Guide*

## 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)

*Transport Interfaces Programming Guide*

## 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(intfd, const struct t_iovec * iov, unsigned intiovcount, intflags);
```

## 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,

```
(iov0.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)

*Transport Interfaces Programming Guide*

**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 *iov*`0` 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 `TBADDADDR` 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)

*Transport Interfaces Programming Guide*

**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>

For more information refer to the *Transport Interfaces Programming Guide*

**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)

*Transport Interfaces Programming Guide*

## 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`

For more information refer to the *Transport Interfaces Programming Guide*

## 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)

*Transport Interfaces Programming Guide*

**NAME**

t_sysconf – get configurable XTI variables

**SYNOPSIS**

```
#include <xti.h>

int t_sysconf(intname);
```

**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)

*Transport Interfaces Programming Guide*

## 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`

For more information refer to the *Transport Interfaces Programming Guide*

**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)

*Transport Interfaces Programming Guide*

**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.

### Routines

These routines deal specifically with the management of the XDR stream. 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 * *aarp* is `NULL` when decoding, `xdr_array()` allocates memory and * *aarp* 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

**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 with the creation of XDR streams. XDR streams have to 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. 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.

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

void xdr_destroy(XDR **xdrs* );

A macro that invokes the destroy routine associated with the XDR stream, *xdrs* . Destruction usually involves freeing private data structures associated with the stream. Using *xdrs* after invoking `xdr_destroy( )` is undefined.

void xdrmem_create(XDR **xdrs* , const caddr_t *addr* `,` const uint_t*size* , const enum xdr_op *op* );

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 (either `XDR_ENCODE` , `XDR_DECODE` , or `XDR_FREE` ).

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* ));

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, *readit* is called. The behavior of these two routines is similar to the system calls `read( )` and `write( )` (see `read`(2) and `write`(2) , respectively), 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. Note: the XDR stream's *op* field must be set by the caller.

Warning: this XDR stream implements an intermediate record stream. Therefore there are additional bytes in the stream to provide record boundary information.

void xdrstdio_create(XDR **xdrs* , FILE **file* , const enum xdr_op *op* );

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 (either `XDR_ENCODE` , `XDR_DECODE` , or `XDR_FREE` ).

Warning: the destroy routine associated with such XDR streams calls `fflush()` on the `file` stream, but never `fclose()` (see `fclose`(3C) ).

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. After the return from the `xdr*_create()` function, if the *x_ops* field is still `NULL,` the call has failed. If the *x_ops* field contains some other value, the call can be assumed to have 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*);

bool_t **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**(, 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 `enum` s (actually integers) and their external representations. This routine returns TRUE if it succeeds, FALSE otherwise. |
| `xdr_float()` | `xdr_float()` translates between C `float` s 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.<br><br>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.<br><br>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**

xfn – overview of the XFN interface

**DESCRIPTION**

The primary service provided by a federated naming system is to map a *composite name* to a *reference*. A composite name is composed of name components from one or more naming systems. A reference consists of one or more communication end points. An additional service provided by a federated naming system is to provide access to attributes associated with named objects. This extension is to satisfy most applications' additional naming service needs without cluttering the basic naming service model. XFN is a programming interface for a federated naming service.

To use the XFN interface, include the `xfn/xfn.h` header file and link the application with `-lxfn`.

The `xfn/xfn.h` header file contains the interface declarations for:

- the XFN base context interface,
- the XFN base attribute interface,
- status object and status codes used by operations in these two interfaces,
- abstract data types passed as parameters to and returned as values from operations in these two interfaces, and
- the interface for the XFN standard syntax model for parsing compound names.

**FILES**

`/usr/include/xfn/xfn.h`

**SEE ALSO**

`FN_ctx_t`(3XFN), `FN_status_t`(3XFN), `xfn_attributes`(3XFN), `xfn_composite_names`(3XFN), `xfn_compound_names`(3XFN), `xfn_status_codes`(3XFN), `fns`(5), `fns_policies`(5)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

xfn_attributes – an overview of XFN attribute operations

**DESCRIPTION**

XFN assumes the following model for attributes. A set of zero or more attributes is associated with a named object. Each attribute in the set has a unique attribute identifier, an attribute syntax, and a (possibly empty) set of distinct data values. Each attribute value has an opaque data type. The attribute identifier serves as a name for the attribute. The attribute syntax indicates how the value is encoded in the buffer.

The operations of the base attribute interface may be used to examine and modify the settings of attributes associated with existing named objects. These objects may be contexts or other types of objects. The attribute operations do not create names or remove names from contexts.

The range of support for attribute operations may vary widely. Some naming systems may not support any attribute operations. Other naming systems may only support read operations, or operations on attributes whose identifiers are in some fixed set. A naming system may limit attributes to have a single value, or may require at least one value. Some naming systems may only associate attributes with context objects, while others may allow associating attributes with non-context objects.

These are the interfaces:

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

FN_attribute_t *fn_attr_get(FN_ctx_t *ctx, const FN_composite_name_t *name,
    const FN_identifier_t *attribute_id, FN_status_t *status);

int fn_attr_modify(FN_ctx_t *ctx, const FN_composite_name_t *name,
    unsigned int mod_op, const FN_attribute_t *attr, FN_status_t *status);

FN_attrset_t *fn_attr_get_ids(FN_ctx_t *ctx, const FN_composite_name_t *name,
    FN_status_t *status);

FN_valuelist_t *fn_attr_get_values(FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_identifier_t *attribute_id, FN_status_t *status);

FN_attrvalue_t *fn_valuelist_next(FN_valuelist_t *vl,
    FN_identifier_t **attr_syntax,
    FN_status_t *status);

void fn_valuelist_destroy(FN_valuelist_t *vl, FN_status_t *status);

FN_multigetlist_t *fn_attr_multi_get(FN_ctx_t *ctx,
    const FN_composite_name_t *name, const FN_attrset_t *attr_ids,
    FN_status_t *status);

FN_attribute_t *fn_multigetlist_next(FN_multigetlist_t *ml,
```

(continued)

**(Continuation)**

```
  FN_status_t *status);

void fn_multigetlist_destroy(FN_multigetlist_t *ml, FN_status_t *status);

int fn_attr_multi_modify(FN_ctx_t *ctx, const FN_composite_name_t *name,
    const FN_attrmodlist_t *mods, FN_status_t *status,
    FN_attrmodlist_t **unexecuted_mods);

FN_attrset_t *fn_ctx_get_syntax_attrs(FN_ctx_t *ctx,
    const FN_composite_name_t *name, FN_status_t *status);
```

The following describes briefly the operations in the base attribute interface. Detailed descriptions are given in the respective reference manual pages for these operations.

`fn_attr_get( )` returns the attribute identified. `fn_attr_modify( )` modifies the attribute identified as described by *mod_op.*

`fn_attr_get_ids( )` returns the identifiers of the attributes of the named object.

`fn_attr_get_values( )` and its set of related operations are used for returning the individual values of an attribute.

`fn_attr_multi_get( )` and its set of related operations are used for returning the requested attributes associated with the named object. `fn_attr_multi_modify( )` modifies multiple attributes associated with the named object in a single invocation.

`fn_ctx_get_syntax_attrs( )` returns the syntax attributes associated with the named context.

**ERRORS**

*status* is set as described in `FN_status_t`(3XFN) and `xfn_status_codes`(3XFN). The following status codes are of special relevance to attribute operations:

| | |
|---|---|
| `FN_E_ATTR_VALUE_REQUIRED` | The operation attempted to create an attribute without a value, and the specific naming system does not allow this. |
| `FN_E_ATTR_NO_PERMISSION` | The caller did not have permission to perform the attempted attribute operation. |
| `FN_E_INSUFFICIENT_RESOURCES` | There are insufficient resources to retrieve the requested attribute(s). |

| | |
|---|---|
| `FN_E_INVALID_ATTR_IDENTIFIER` | The attribute identifier was not in a format acceptable to the naming system, or its contents was not valid for the format specified for the identifier. |
| `FN_E_INVALID_ATTR_VALUE` | One of the values supplied was not in the appropriate form for the given attribute. |
| `FN_E_NO_SUCH_ATTRIBUTE` | The object did not have an attribute with the given identifier. |
| `FN_E_TOO_MANY_ATTR_VALUES` | The operation attempted to associate more values with an attribute than the naming system supported. |

## USAGE

Except for `fn_ctx_get_syntax_attrs()`, an attribute operation using a composite name is not necessarily equivalent to an independent `fn_ctx_lookup()` operation followed by an attribute operation in which the caller supplies the resulting reference and an empty name. This is because there is a range of attribute models in which an attribute is associated with a name in a context, or an attribute is associated with the object named, or both. XFN accommodates all of these alternatives. Invoking an attribute operation using the target context and the terminal atomic name accesses either the attributes that are associated with the target name or target named object; this is dependent on the underlying attribute model. This document uses the term *attributes associated with a named object* to refer to all of these cases.

XFN specifies no guarantees about the relationship between the attributes and the reference associated with a given name. Some naming systems may store the reference bound to a name in one or more attributes associated with a name. Attribute operations might affect the information used to construct a reference.

To avoid undefined results, programmers must use the operations in the context interface and not attribute operations when the intention is to manipulate a reference. Programmers should avoid the use of specific knowledge about how an XFN context implementation over a particular naming system constructs references.

## SEE ALSO

`FN_attribute_t`(3XFN), `FN_attrset_t`(3XFN), `FN_attrvalue_t`(3XFN), `FN_composite_name_t`(3XFN), `FN_ctx_t`(3XFN), `FN_identifier_t`(3XFN), `FN_status_t`(3XFN), `fn_attr_get`(3XFN), `fn_attr_get_ids`(3XFN), `fn_attr_get_values`(3XFN), `fn_attr_modify`(3XFN), `fn_attr_multi_get`(3XFN), `fn_attr_multi_modify`(3XFN), `fn_ctx_get_syntax_attrs`(3XFN), `fn_ctx_lookup`(3XFN), `xfn`(3XFN), `xfn_status_codes`(3XFN)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

xfn_composite_names – XFN composite syntax: an overview of the syntax for XFN composite name

**DESCRIPTION**

An *XFN composite name* consists of an ordered list of zero or more components. Each component is a string name from the namespace of a single naming system. It may be an atomic or a compound name in that namespace.

XFN defines an abstract data type, `FN_composite_name_t`, for representing the structural form of a composite name. XFN also defines a standard string form for composite names. This form is the concatenation of the components of a composite name from left to right with the *XFN component separator* ('/') character to separate each component.

These are the interfaces:

```
#include <xfn/xfn.h>
FN_composite_name_t *fn_composite_name_from_string( const FN_string_t *str);
FN_string_t *fn_string_from_composite_name( const
FN_composite_name_t *name);
```

The function `fn_composite_name_from_string` parses the string representation of a composite name into its corresponding composite name object `FN_composite_name_t`. The function `fn_string_from_composite_name` composes the string representation of a composite name given its composite name object form `FN_composite_name_t`.

**APPLICATION USAGE**

Special characters used in the XFN composite name syntax, such as the separator or escape characters, have the same encoding as they would in `ISO 646`.

All XFN implementations are required to support the portable representation, `ISO 646`. All other representations are optional.

All characters of the string form of a XFN composite name use a single encoding. This does not preclude component names of a composite name in its structural form from having different encodings. Code set mismatches that occur during the process of coverting a composite name structure to its string form are resolved in an implementation-dependent way. When an implementation discovers that a composite name has components with incompatible code sets, it returns the error code `FN_E_INCOMPATIBLE_CODE_SETS.`

**SEE ALSO**

`FN_string_t`(3XFN), `FN_compound_name_t`(3XFN), `xfn`(3XFN)

## NAME

xfn_compound_names – XFN compound syntax: an overview of XFN model for compound name parsing

## DESCRIPTION

Each naming system in an XFN federation has a naming convention. XFN defines a standard model of expressing compound name syntax that covers a large number of specific name syntaxes and is expressed in terms of syntax properties of the naming convention.

The model uses the attributes in the following table to describe properties of the syntax. Unless otherwise qualified, these syntax attributes have attribute identifiers that use the `FN_ID_STRING` format. A context that supports the XFN standard syntax model has an attribute set containing the `fn_syntax_type` (with identifier format `FN_ID_STRING`) attribute with the value "standard" (ASCII attribute syntax).

These are the interfaces:

```
#include <xfn/xfn.h>
FN_attrset_t *fn_ctx_get_syntax_attrs(FN_ctx_t *ctx,
const FN_composite_name_t *name,
FN_status_t *status);
FN_compound_name_t *fn_compound_name_from_syntax_attrs( const FN_attrset_t *aset,
const FN_string_t *name, FN_status_t *status);
```

| | |
|---|---|
| `fn_syntax_type` | Its value is the ASCII string "standard" if the context supports the XFN standard syntax model. Its value is an implementation-specific value if another syntax model is supported. |
| `fn_std_syntax_direction` | Its value is an ASCII string, one of "left_to_right", "right_to_left", or "flat". This determines whether the order of components in a compound name string goes from left to right, right to left, or whether the namespace is flat (in other words, not hierarchical; em all names are atomic). |
| `fn_std_syntax_separator` | Its value is the separator string for this name syntax. This attribute is required unless the `fn_std_syntax_direction` is "flat". |

| | |
|---|---|
| `fn_std_syntax_escape` | If present, its value is the escape string for this name syntax. |
| `fn_std_syntax_case_insensitive` | If this attribute is present, it indicates that names that differ only in case are considered identical. If this attribute is absent, it indicates that case is significant. If a value is present, it is ignored. |
| `fn_std_syntax_begin_quote` | If present, its value is the begin-quote string for this syntax. There can be multiple values for this attribute. |
| `fn_std_syntax_end_quote` | If present, its value is the end-quote string for this syntax. There can be multiple values for this attribute. |
| `fn_std_syntax_ava_separator` | If present, its value is the attribute value assertion separator string for this syntax. |
| `fn_std_syntax_typeval_separator` | If present, its value is the attribute type-value separator string for this syntax. |
| `fn_std_syntax_code_sets` | If present, its value identifies the code sets of the string representation for this syntax. Its value consists of a structure containing an array of code sets supported by the context; the first member of the array is the preferred code set of the context. The values for the code sets are defined in the X/Open code set registry. If this attribute is not present, or if the value is empty, the default code set is `ISO 646` (same encoding as ASCII). |
| `fn_std_syntax_locale_info` | If present, identifies locale information, such as character |

set information, of the string representation for this syntax. The interpretation of its value is implementation-dependent.

The XFN standard syntax attributes are interpreted according to the following rules:

1. In a string without quotes or escapes, any instance of the separator string delimits two atomic names.
2. A separator, quotation or escape string is escaped if preceded immediately (on the left) by the escape string.
3. A non-escaped begin-quote which precedes a component must be matched by a non-escaped end-quote at the end of the component. Quotes embedded in non-quoted names are treated as simple characters and do not need to be matched. An unmatched quotation fails with the status code `FN_E_ILLEGAL_NAME.`
4. If there are multiple values for begin-quote and end-quote, a specific begin-quote value must be matched with its corresponding end-quote value.
5. When the separator appears between a (non-escaped) begin quote and the end quote, it is ignored.
6. When the separator is escaped, it is ignored. An escaped begin-quote or end-quote string is not treated as a quotation mark. An escaped escape string is not treated as an escape string.
7. A non-escaped escape string appearing within quotes is interpreted as an escape string. This can be used to embed an end-quote within a quoted string.

After constructing a compound name from a string, the resulting component atoms have one level of escape strings and quotations interpreted and consumed.

`fn_ctx_get_syntax_attrs( )` is used to obtain the syntax attributes associated with a context.

`fn_compound_name_from_syntax( )` is used to construct a compound name object using the string form of the name and the syntax attributes of the name.

**ERRORS**

| | |
|---|---|
| `FN_E_ILLEGAL_NAME` | The name supplied to the operation was not a well-formed component according to the name syntax of the context. |
| `FN_E_INCOMPATIBLE_CODE_SETS` | Code set mismatches that occur during the construction |

| | |
|---|---|
| | of the compound name's string form are resolved in an implementation-dependent way. When an implementation discovers that a compound name has components with incompatible code sets, it returns this error code. |
| `FN_E_INVALID_SYNTAX_ATTRS` | The syntax attributes supplied are invalid or insufficient to fully specify the syntax. |
| `FN_E_SYNTAX_NOT_SUPPORTED` | The syntax specified is not supported. |

**USAGE**

Most applications treat names as opaque data. Hence, the majority of clients of the XFN interface will not need to parse compound names from specific naming systems. Some applications, however, such as browsers, need such capabilities. These applications would use `fn_ctx_get_syntax_attrs()` to obtain the syntax-related attributes of a context and, if the context uses the XFN standard syntax model, it would examine these attributes to determine the name syntax of the context.

**SEE ALSO**

`FN_attribute_t`(3XFN), `FN_attrset_t`(3XFN), `FN_compound_name_t`(3XFN), `FN_identifier_t`(3XFN), `FN_string_t`(3XFN) `fn_ctx_get_syntax_attrs` (3XFN), `xfn`(3XFN)

**NOTES**

The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

**NAME**

xfn_links – XFN links: an overview of XFN links

**DESCRIPTION**

An *XFN link* is a special form of reference that contains a composite name, the *link name*, and that may be bound to an atomic name in an XFN context. Because the link name is a composite name, it may span multiple namespaces.

Normal resolution of names in context operations always follows XFN links. If the first composite name component of the link name is the atomic name ".", the link name is resolved relative to the same context in which the link is bound, otherwise, the link name is resolved relative to the XFN Initial Context of the client. The link name may itself cause resolution to pass through other XFN links. This gives rise to the possibility of a cycle of links whose resolution could not terminate normally. As a simple means to avoid such non-terminating resolutions, implementations may define limits on the number of XFN links that may be resolved in any single operation invoked by the caller.

These are the interfaces:

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

FN_ref_t *fn_ref_create_link(const FN_composite_name_t *link_name);

int fn_ref_is_link(const FN_ref_t *ref);

FN_composite_name_t *fn_ref_link_name( const FN_ref_t *link_ref);

FN_ref_t *fn_ctx_lookup_link(FN_ctx_t *ctx, const FN_composite_name_t *name,
    FN_status_t *status);

unsigned int fn_status_link_code(const FN_status_t *stat);

const FN_composite_name_t *fn_status_link_remaining_name(
    const FN_status_t *stat);

const FN_composite_name_t *fn_status_link_resolved_name(
    const FN_status_t *stat);

const FN_ref_t *fn_status_link_resolved_ref( const FN_status_t *stat);

int fn_status_set_link_code(FN_status_t *stat,
    unsigned int code);

int fn_status_set_link_remaining_name(FN_status_t *stat,
    const FN_composite_name_t *name);

int fn_status_set_link_resolved_name(FN_status_t *stat,
    const FN_composite_name_t *name);

int fn_status_set_link_resolved_ref(FN_status_t *stat,
    const FN_ref_t *ref);
```

Links are bound to names using the normal `fn_ctx_bind( )` and unbound using the normal `fn_ctx_unbind( )` operation. The operation `fn_ref_create_link( )` is provided for constructing a link reference from a composite name. Since normal resolution always follows links, a separate operation, `fn_ctx_lookup_link( )` is provided to lookup the link itself.

In the case that an error occurred while resolving an XFN link, the status object set by the operation contains additional information about that error and sets the corresponding link status fields using `fn_status_set_link_code( )`, `fn_status_set_link_remaining_name( )`, `fn_status_set_link_resolved_name( )` and `fn_status_set_link_resolved_ref( )`. The link status fields can be retrieved using `fn_status_link_code( )`, `fn_status_link_remaining_name( )`, `fn_status_link_resolved_name( )` and `fn_status_link_resolved_ref( )`.

## ERRORS

The following status codes are of special relevance when performing operations involving XFN links:

| | |
|---|---|
| `FN_E_LINK_ERROR` | There was an error encountered resolving an XFN link encountered during resolution of the supplied name. Check the link part of the status object to determine cause of the link error. |
| `FN_E_LINK_LOOP_LIMIT` | A non-terminating loop (cycle) in the resolution can arise due to XFN links encountered during the resolution of a composite name. This code indicates either the definite detection of such a cycle, or that resolution exceeded an implementation-defined limit on the number of XFN links allowed for a single operation invoked by the caller. |
| `FN_E_MALFORMED_LINK` | A malformed link reference was encountered. For the `fn_ctx_lookup_link( )` operation, the name supplied resolved to a reference that was not a link. |

## APPLICATION USAGE

For the `fn_ctx_bind( )`, `fn_ctx_unbind( )`, `fn_ctx_rename( )`, `fn_ctx_lookup_link( )`, `fn_ctx_create_subcontext( )` and `fn_ctx_destroy_subcontext( )` operations, resolution of the given name continues to the target context — that named by all but the terminal atomic part of the given name; the terminal atomic name is not resolved. Consequently, for operations that involve unbinding the terminal atomic part such as `fn_ctx_unbind( ) ,` if the terminal atomic name is bound to a link, the link is not followed and the link itself is unbound from the terminal atomic name.

Many naming systems support a native notion of link that may be used within the naming system itself. XFN does not determine whether there is any relationship between such native links and XFN links.

**SEE ALSO**

`FN_composite_name_t`(3XFN), `FN_ref_t`(3XFN), `FN_status_t`(3XFN), `fn_ctx_bind`(3XFN), `fn_ctx_destroy_subcontext`(3XFN), `fn_ctx_lookup`(3XFN), `fn_ctx_lookup_link`(3XFN), `fn_ctx_rename`(3XFN), `fn_ctx_unbind`(3XFN), `xfn_status_codes`(3XFN), `xfn`(3XFN)

## NAME

xfn_status_codes – descriptions of XFN status codes

## SYNOPSIS

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

## DESCRIPTION

The result status of operations in the context interface and the attribute interface is encapsulated in an `FN_status_t` object. This object contains information about how the operation completed: whether an error occurred in performing the operation; if so, what kind of error; and information localizing where the error occurred. In the case that the error occurred while resolving an XFN link, the status object contains additional information about that error.

The context status object consists of several items of information. One of them is the primary status code, describing the disposition of the operation. In the case that an error occurred while resolving an XFN link, the primary status code has the value `FN_E_LINK_ERROR`, and the link status code describes the error that occurred while resolving the XFN link.

### XFN Status Codes

Both the primary status code and the link status code are values of type `unsigned int` that are drawn from the same set of meaningful values. XFN reserves the values `0` through `127` for standard meanings. Currently, values and interpretations for the following codes are determined by XFN.

| | |
|---|---|
| `FN_SUCCESS` | The operation succeeded. |
| `FN_E_ATTR_NO_PERMISSION` | The caller did not have permission to perform the attempted attribute operation. |
| `FN_E_ATTR_VALUE_REQUIRED` | The operation attempted to create an attribute without a value, and the specific naming system does not allow this. |
| `FN_E_AUTHENTICATION_FAILURE` | The identity of the client principal could not be verified. |
| `FN_E_COMMUNICATION_FAILURE` | An error occurred in communicating with one of the contexts involved in the operation. |
| `FN_E_CONFIGURATION_ERROR` | A problem was detected that indicated an error in the installation of the XFN implementation. |
| `FN_E_CONTINUE` | The operation should be continued using the remaining name and the resolved reference returned in the status. |

| | |
|---|---|
| `FN_E_CTX_NO_PERMISSION` | The client did not have permission to perform the operation. |
| `FN_E_CTX_NOT_EMPTY` | (Applies only to `fn_ctx_destroy_subcontext( )`.) The naming system required that the context be empty before its destruction, and it was not empty. |
| `FN_E_CTX_UNAVAILABLE` | Service could not be obtained from one of the contexts involved in the operation. This may be because the naming system is busy, or is not providing service. In some implementations this may not be distinguished from a communication failure. |
| `FN_E_ILLEGAL_NAME` | The name supplied to the operation was not a well- formed XFN composite name, or one of the component names was not well-formed according to the syntax of the naming system(s) involved in its resolution. |
| `FN_E_E_INCOMPATIBLE_CODE_SETS` | The operation involved character strings of incompatible code sets, or the supplied code set is not supported by the implementation. |
| `FN_E_INSUFFICIENT_RESOURCES` | Either the client or one of the involved contexts could not obtain sufficient resources (for example, memory, file descriptors, communication ports, stable media space, and so on) to complete the operation successfully. |
| `FN_E_INVALID_ATTR_IDENTIFIER` | The attribute identifier was not in a format acceptable to the naming system, or its content was not valid for the format specified for the identifier. |

| | |
|---|---|
| `FN_E_INVALID_ATTR_VALUE` | One of the values supplied was not in the appropriate form for the given attribute. |
| `FN_E_INVALID_ENUM_HANDLE` | The enumeration handle supplied was invalid, either because it was from another enumeration, or because an update operation occurred during the enumeration, or because of some other reason. |
| `FN_E_INVALID_SYNTAX_ATTRS` | The syntax attributes supplied are invalid or insufficient to fully specify the syntax. |
| `FN_E_LINK_ERROR` | There was an error in resolving an XFN link encountered during resolution of the supplied name. |
| `FN_E_LINK_LOOP_LIMIT` | A non-terminating loop (cycle) in the resolution can arise due to XFN links encountered during the resolution of a composite name. This code indicates either the definite detection of such a cycle, or that resolution exceeded an implementation-defined limit on the number of XFN links allowed for a single operation invoked by the caller. |
| `FN_E_MALFORMED_LINK` | A malformed link reference was encountered. For `fn_ctx_lookup_link()`, the name supplied resolved to a reference that was not a link. |
| `FN_E_MALFORMED_REFERENCE` | A context object could not be constructed from the supplied reference, because the reference was not properly formed. |
| `FN_E_NAME_IN_USE` | (Only for operations that bind names.) The supplied name was already in use. |
| `FN_E_NAME_NOT_FOUND` | Resolution of the supplied composite name proceeded to a context in |

| | |
|---|---|
| | which the next atomic component of the name was not bound. |
| `FN_E_NO_SUCH_ATTRIBUTE` | The object did not have an attribute with the given identifier. |
| `FN_E_NO_SUPPORTED_ADDRESS` | A context object could not be constructed from a particular reference. The reference contained no address type over which the context interface was supported. |
| `FN_E_NOT_A_CONTEXT` | Either one of the intermediate atomic names did not name a context, and resolution could not proceed beyond this point, or the operation required that the caller supply the name of a context, and the name did not resolve to a reference for a context. |
| `FN_E_OPERATION_NOT_SUPPORTED` | The operation attempted is not supported. |
| `FN_E_PARTIAL_RESULT` | The operation attempted is returning a partial result. |
| `FN_E_SYNTAX_NOT_SUPPORTED` | The syntax type specified is not supported. |
| `FN_E_TOO_MANY_ATTR_VALUES` | The operation attempted to associate more values with an attribute than the naming system supported. |
| `FN_E_UNSPECIFIED_ERROR` | An error occurred that could not be classified by any of the other error codes. |

**FILES**

| | |
|---|---|
| `#include <xfn/xfn.h>` | XFN status codes header file |

**SEE ALSO** `FN_status_t`(3XFN), `xfn`(3XFN)

**NOTES** The implementation of XFN in this Solaris release is based on the X/Open preliminary specification. It is likely that there will be minor changes to these interfaces to reflect changes in the final version of this specification. The next minor release of Solaris will offer binary compatibility for applications developed using the current interfaces. As the interfaces evolve toward standardization, it is possible that future releases of Solaris will require minor source code changes to applications that have been developed against the preliminary specification.

## 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>

## 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 recommended enterprise level information service is NIS+ or NIS version 3, see `nis+`(1) . Moreover, this version of SunOS supports only the client interface to NIS version 2. It is expected that this client interface will be served either by an existing `ypserv` process running on another machine on the network that has an earlier version of SunOS or by an NIS+ server, see `rpc.nisd`(1M) , running in "YP-compatibility mode". Refer to the NOTES section in `ypfiles`(4) for implications of being an NIS client of an NIS+ server in "YP-compatibility mode", and to `ypbind`(1M) , `ypwhich`(1) , `ypmatch`(1) , and `ypcat`(1) for commands to access NIS from a client machine. 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 may 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* . *indomain* and *inmap* strings must be non-null and null-terminated. String parameters which are accompanied by a count parameter may not be null, but 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, and a failure code (YPERR_ *xxxx* ) otherwise. Failure codes are described in the `ERRORS` section.

### Routines

yp_bind (char **indomain* );

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; this is done automatically whenever an NIS lookup function is called. `yp_bind( )` 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. If a process calls `yp_bind( )` , it should call `yp_unbind( )` when it is done using NIS in order to free up resources.

void yp_unbind(char **indomain* );

Each binding allocates (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. `yp_unbind( )` 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.

If an error is not RPC-related, or if `rpcbind` is not running, or if `ypbind` is not running, or if a bound `ypserv` process returns any answer (success or failure), the `ypclnt` layer will return control to the user code, either with an error code, or a success code and any results.

yp_get_default_domain (char ***outdomain* );

The NIS lookup calls require a map name and a domain name, at minimum. It is assumed that the client process knows the name of the map of interest. Client processes should 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 thus 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(char **indomain* , char **inmap* , char **inkey* , int *inkeylen* , char ***outval* , int **outvallen* );

`yp_match( )` returns the value associated with a passed key. This key must be exact; no pattern matching is available. `yp_match( )` requires a full YP map name; for example, `hosts.byname` instead of the nickname `hosts` .

yp_first(char **indomain* , char **inmap* , char ***outkey* , int **outkeylen* , char ***outval* , int **outvallen* );

`yp_first( )` returns the first key-value pair from the named map in the named domain.

yp_next(char **indomain* , char **inmap* , char **inkey* , int *inkeylen* , char ***outkey* , int **outkeylen* , char ***outval* , int **outvallen* );

`yp_next( )` 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, for that matter, of next) is particular to the structure of the NIS map being processing; there is no relation in retrieval order 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 made 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 will be seen exactly once. Further, if the same sequence of operations is performed on the same map at the same server, the entries will be seen in the same order.

Under conditions of heavy server load or server failure, it is possible for the domain to become unbound, then bound once again (perhaps to a different server) while a client is running. This 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. The next paragraph describes a better solution to enumerating all entries in a map.

yp_all(char **indomain* , char **inmap* , struct ypall_callback **incallback* );

The function `yp_all( )` 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 take place as a single RPC request and response. `yp_all( )` can be used just like any other NIS name service procedure, identify the map in the normal manner, and supply the name of a function which 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 will hold one of the return status values defined in `<rpcsvc/yp_prot.h` – either `YP_TRUE` or an error code. (See `ypprot_err( )` , below, for a function which 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. It is the responsibility of the `foreach( )` function to 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 will 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 may 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 a 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(char **indomain* , char **inmap* , unsigned long **outorder* );
`yp_order( )` returns the order number for a map. This function is not supported if the `ypbind` process on the client's system is bound to an NIS+ server running in "YP-compatibility mode".

yp_master(char **indomain* , char **inmap* , char ***outname* );
`yp_master( )` returns the machine name of the master NIS server for a map.

char *yperr_string(int *incode* );
`yperr_string( )` returns a pointer to an error message string that is null-terminated but contains no period or NEWLINE .

ypprot_err (unsigned int *incode* );

`ypprot_err()` takes an NIS name service protocol error code as input, and returns a ypclnt layer error code, which may be used in turn 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**

`nis+`(1) , `ypcat`(1) , `ypmatch`(1) , `ypwhich`(1) , `rpc.nisd`(1M) , `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, 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.

# Index