

AIX Version 7.3

*Technical Reference: Communication  
Subroutines*



**Note**

Before using this information and the product it supports, read the information in [“Notices” on page 383.](#)

This edition applies to AIX Version 7.3 and to all subsequent releases and modifications until otherwise indicated in new editions.

© **Copyright International Business Machines Corporation 2021.**

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

---

# Contents

<b>About this document.....</b>	<b>vii</b>
Highlighting.....	vii
Case-sensitivity in AIX.....	vii
ISO 9000.....	vii
<b>Communication subroutines.....</b>	<b>1</b>
eXternal Data Representation.....	1
xdr_accepted_reply Subroutine.....	1
xdr_array Subroutine.....	2
xdr_bool Subroutine.....	2
xdr_bytes Subroutine.....	3
xdr_callhdr Subroutine.....	4
xdr_callmsg Subroutine.....	5
xdr_char Subroutine.....	5
xdr_destroy Macro.....	6
xdr_enum Subroutine.....	7
xdr_float Subroutine.....	7
xdr_free Subroutine.....	8
xdr_getpos Macro.....	9
xdr_hyper Subroutine.....	10
xdr_inline Macro.....	10
xdr_int Subroutine.....	11
xdr_long Subroutine.....	12
xdr_opaque Subroutine.....	12
xdr_opaque_auth Subroutine.....	13
xdr_pmap Subroutine.....	14
xdr_pmaplist Subroutine.....	15
xdr_pointer Subroutine.....	15
xdr_reference Subroutine.....	16
xdr_rejected_reply Subroutine.....	17
xdr_replymsg Subroutine.....	18
xdr_setpos Macro.....	18
xdr_short Subroutine.....	19
xdr_string Subroutine.....	20
xdr_u_char Subroutine.....	21
xdr_u_int Subroutine.....	21
xdr_u_long Subroutine.....	22
xdr_u_short Subroutine.....	23
xdr_union Subroutine.....	24
xdr_vector Subroutine.....	25
xdr_void Subroutine.....	25
xdr_wrapstring Subroutine.....	26
xdr_authunix_parms Subroutine.....	27
xdr_double Subroutine.....	27
xdrmem_create Subroutine.....	28
xdrrec_create Subroutine.....	29
xdrrec_endofrecord Subroutine.....	30
xdrrec_eof Subroutine.....	31
xdrrec_skiprecord Subroutine.....	31
xdrstdio_create Subroutine.....	32

Network Information Services.....	33
yp_all Subroutine.....	33
yp_bind Subroutine.....	35
yp_first Subroutine.....	36
yp_get_default_domain Subroutine.....	37
yp_master Subroutine.....	37
yp_match Subroutine.....	38
yp_next Subroutine.....	39
yp_order Subroutine.....	41
yp_unbind Subroutine.....	41
yp_update Subroutine.....	42
yperr_string Subroutine.....	43
ypprot_err Subroutine.....	44
Simple Network Management Protocol (SNMP).....	45
getsmuxEntrybyname or getsmuxEntrybyidentity Subroutine.....	45
isodetailor Subroutine.....	46
ll_hdinit, ll_dbinit, ll_log, or ll_log Subroutine.....	47
o_number, o_integer, o_string, o_igeneric, o_generic, o_specific, or o_ipaddr Subroutine.....	50
oid_cmp, oid_cpy, oid_free, sprintoid, str2oid, ode2oid, oid2ode, oid2ode_aux, prim2oid, or oid2prim Subroutine.....	52
oid_extend or oid_normalize Subroutine.....	54
readobjects Subroutine.....	55
s_generic Subroutine.....	56
smux_close Subroutine.....	57
smux_error Subroutine.....	58
smux_free_tree Subroutine.....	59
smux_init Subroutine.....	60
smux_register Subroutine.....	61
smux_response Subroutine.....	62
smux_simple_open Subroutine.....	63
smux_trap Subroutine.....	64
smux_wait Subroutine.....	66
text2inst, name2inst, next2inst, or nexttot2inst Subroutine.....	67
text2oid or text2obj Subroutine.....	68
Sockets.....	69
_.....	69
a.....	72
b.....	77
c.....	80
d.....	84
e.....	87
f.....	97
g.....	110
h.....	157
i.....	160
kvalid_user Subroutine.....	200
listen Subroutine.....	201
n.....	202
PostQueuedCompletionStatus Subroutine.....	205
r.....	206
s.....	247
WriteFile Subroutine.....	317
Packet Capture.....	319
ioctl BPF Control Operations.....	319
Librdmacm Library.....	321
Returned error rules.....	321
Supported verbs.....	322
Device Management.....	354

Memory region management.....	355
Libibverbs Library.....	358
Returned error rules.....	358
Supported Verbs.....	358
Verbs not supported by the libibverbs library.....	381
<b>Notices.....</b>	<b>383</b>
Privacy policy considerations.....	384
Trademarks.....	385
<b>Index.....</b>	<b>387</b>



## About this document

---

This topic collection provides experienced C programmers with complete detailed information about eXternal Data Representation, Network Information Services (NIS), and Simple Network Management Protocol (SNMP), sockets, packet capture, Librdman library, and Libibverbs library for the AIX operating system. To use the topic collection effectively, you should be familiar with commands, system calls, subroutines, file formats, and special files. This publication is also available on the documentation CD that is shipped with the operating system.

## Highlighting

---

The following highlighting conventions are used in this document:

Item	Description
<b>Bo1d</b>	Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects.
<i>Italics</i>	Identifies parameters whose actual names or values are to be supplied by the user.
Monospace	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

## Case-sensitivity in AIX

---

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

## ISO 9000

---

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



---

# Technical Reference: Communication subroutines

Review the communication subroutines for eXternal data representation, Network Information Services (NIS), Simple Network Management Protocol (SNMP), and sockets.

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

---

## eXternal Data Representation

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

### xdr\_accepted\_reply Subroutine

#### Purpose

Encodes RPC reply messages.

#### Library

C Library (**libc.a**)

#### Syntax

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

```
int xdr_accepted_reply ( xdrs, ar )  
XDR *xdrs;  
struct accepted_reply *ar;
```

#### Description

The **xdr\_accepted\_reply** subroutine encodes Remote Procedure Call (RPC) reply messages. The routine generates message replies similar to RPC message replies without using the RPC program.

#### Parameters

Item	Description
------	-------------

<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
-------------	---

<i>ar</i>	Specifies the address of the structure that contains the RPC reply.
-----------	---

#### Return Values

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

#### Related information

[List of RPC Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Remote Procedure Call \(RPC\) Overview for Programming](#)

## xdr\_array Subroutine

### Purpose

Translates between variable-length arrays and their corresponding external representations.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_array (xdrs, arrp, sizep, maxsize, elsize, elproc)  
XDR * xdrs;  
char ** arrp;  
u_int * sizep;  
u_int maxsize;  
u_int elsize;  
xdrproc_t elproc;
```

### Description

The **xdr\_array** subroutine is a filter primitive that translates between variable-length arrays and their corresponding external representations. This subroutine is called to encode or decode each element of the array.

### Parameters

Item	Description
<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
<i>arrp</i>	Specifies the address of the pointer to the array. If the <i>arrp</i> parameter is null when the array is being deserialized, the XDR program allocates an array of the appropriate size and sets the parameter to that array.
<i>sizep</i>	Specifies the address of the element count of the array. The element count cannot exceed the value for the <i>maxsize</i> parameter.
<i>maxsize</i>	Specifies the maximum number of array elements.
<i>elsize</i>	Specifies the byte size of each of the array elements.
<i>elproc</i>	Translates between the C form of the array elements and their external representations. This parameter is an XDR filter.

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_bool Subroutine

### Purpose

Translates between Booleans and their external representations.

## Library

C Library (**libc.a**)

## Syntax

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

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

## Description

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

## Parameters

### Item Description

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

*bp* Specifies the address of the Boolean data.

## Return Values

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

## Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_bytes Subroutine

### Purpose

Translates between internal counted byte arrays and their external representations.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_bytes ( xdrs, sp, sizep, maxsize)  
XDR *xdrs;  
char **sp;  
u_int *sizep;  
u_int maxsize;
```

### Description

The **xdr\_bytes** subroutine is a filter primitive that translates between counted byte arrays and their external representations. This subroutine treats a subset of generic arrays, in which the size of array

elements is known to be 1 and the external description of each element is built-in. The length of the byte array is explicitly located in an unsigned integer. The byte sequence is not terminated by a null character. The external representation of the bytes is the same as their internal representation.

## Parameters

Item	Description
<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
<i>sp</i>	Specifies the address of the pointer to the byte array.
<i>sizep</i>	Points to the length of the byte area. The value of this parameter cannot exceed the value of the <i>maxsize</i> parameter.
<i>maxsize</i>	Specifies the maximum number of bytes allowed when XDR encodes or decodes messages.

## Return Values

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

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_callhdr Subroutine

### Purpose

Describes RPC call header messages.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_callhdr ( xdrs, chdr )  
XDR *xdrs;  
struct rpc_msg *chdr;
```

### Description

The **xdr\_callhdr** subroutine describes Remote Procedure Call (RPC) call header messages. This subroutine generates call headers that are similar to RPC call headers without using the RPC program.

### Parameters

Item	Description
<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
<i>chdr</i>	Points to the structure that contains the header for the call message.

## Return Values

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

## Related information

[List of RPC Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Remote Procedure Call \(RPC\) Overview for Programming](#)

## xdr\_callmsg Subroutine

### Purpose

Describes RPC call messages.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_callmsg ( xdrs, msg )  
XDR *xdrs;  
struct rpc_msg *msg;
```

### Description

The **xdr\_callmsg** subroutine describes Remote Procedure Call (RPC) call messages. This subroutine generates messages similar to RPC messages without using the RPC program.

### Parameters

Item	Description
------	-------------

<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
-------------	---

<i>msg</i>	Points to the structure that contains the text of the call message.
------------	---

## Return Values

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

## Related information

[List of RPC Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Remote Procedure Call \(RPC\) Overview for Programming](#)

## xdr\_char Subroutine

### Purpose

Translates between C language characters and their external representations.

### Library

C Library (**libc.a**)

## Syntax

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

```
xdr_char ( xdrs, cp )  
XDR *xdrs;  
char *cp;
```

## Description

The **xdr\_char** subroutine is a filter primitive that translates between C language characters and their external representations.

**Note:** Encoded characters are not packed and occupy 4 bytes each. For arrays of characters, the programmer should consider using the **xdr\_bytes**, **xdr\_opaque**, or **xdr\_string** routine.

## Parameters

### Item Description

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

*cp* Points to the character.

## Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_destroy Macro

### Purpose

Destroys the XDR stream pointed to by the *xdrs* parameter.

### Library

C Library (**libc.a**)

## Syntax

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

```
void xdr_destroy ( xdrs )  
XDR *xdrs;
```

## Description

The **xdr\_destroy** macro invokes the destroy routine associated with the eXternal Data Representation (XDR) stream pointed to by the *xdrs* parameter and frees the private data structures allocated to the stream. The use of the XDR stream handle is undefined after it is destroyed.

## Parameters

### Item Description

*xdrs* Points to the XDR stream handle.

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Non-Filter Primitives](#)

## xdr\_enum Subroutine

### Purpose

Translates between a C language enumeration (enum) and its external representation.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_enum ( xdrs, ep )  
XDR *xdrs;  
enum_t *ep;
```

### Description

The **xdr\_enum** subroutine is a filter primitive that translates between a C language enumeration (enum) and its external representation.

### Parameters

Item	Description
------	-------------

<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
-------------	---

<i>ep</i>	Specifies the address of the enumeration data.
-----------	--

### Return Values

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

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_float Subroutine

### Purpose

Translates between C language floats and their external representations.

### Library

C Library (**libc.a**)

## Syntax

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

```
xdr_float ( xdrs, fp)  
XDR *xdrs;  
float *fp;
```

## Description

The **xdr\_float** subroutine is a filter primitive that translates between C language floats (normalized single-precision floating-point numbers) and their external representations.

## Parameters

### Item Description

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

*fp* Specifies the address of the float.

## Return Values

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

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_free Subroutine

### Purpose

Deallocates, or frees, memory.

### Library

C Library (**libc.a**)

## Syntax

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

```
void xdr_free ( proc, objp)  
xdrproc_t proc;  
char *objp;
```

## Description

The **xdr\_free** subroutine is a generic freeing routine that deallocates memory. The *proc* parameter specifies the eXternal Data Representation (XDR) routine for the object being freed. The *objp* parameter is a pointer to the object itself.

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

## Parameters

### Item Description

*proc* Points to the XDR stream handle.

*objp* Points to the object being freed.

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Non-Filter Primitives](#)

## xdr\_getpos Macro

### Purpose

Returns an unsigned integer that describes the current position in the data stream.

### Library

C Library (**libc.a**)

### Syntax

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

```
u_int xdr_getpos ( xdrs )
```

```
XDR *xdrs;
```

### Description

The **xdr\_getpos** macro invokes the get-position routine associated with the eXternal Data Representation (XDR) stream pointed to by the *xdrs* parameter. This routine returns an unsigned integer that describes the current position in the data stream.

### Parameters

#### Item Description

*xdrs* Points to the XDR stream handle.

### Return Values

This macro returns an unsigned integer describing the current position in the stream. In some XDR streams, it returns a value of -1, even though the value has no meaning.

### Related reference

[xdr\\_setpos Macro](#)

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Non-Filter Primitives](#)

## xdr\_hyper Subroutine

### Purpose

Translates long integers from C language to their external representations.

### Library

C Library (**libc.a**)

### Syntax

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

### Description

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

### Parameters

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

### Return Values

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

## xdr\_inline Macro

### Purpose

Returns a pointer to the buffer of a stream pointed to by the *xdrs* parameter.

### Library

C Library (**libc.a**)

### Syntax

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

```
long *x_inline ( xdrs, len )  
XDR *xdrs;  
int len;
```

### Description

The **xdr\_inline** macro invokes the inline subroutine associated with the eXternal Data Representation (XDR) stream pointed to by the *xdrs* parameter. The subroutine returns a pointer to a contiguous piece of the stream's buffer, whose size is specified by the *len* parameter. The buffer can be used for any purpose, but it is not data-portable. The **xdr\_inline** macro may return a value of null if it cannot return a buffer segment of the requested size.

## Parameters

Item	Description
------	-------------

<i>xdrs</i>	Points to the XDR stream handle.
<i>len</i>	Specifies the size, in bytes, of the internal buffer.

## Return Values

This macro returns a pointer to a piece of the stream's buffer.

## Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Non-Filter Primitives](#)

## xdr\_int Subroutine

### Purpose

Translates between C language integers and their external representations.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_int ( xdrs, ip )  
XDR *xdrs;  
int *ip;
```

### Description

The **xdr\_int** subroutine is a filter primitive that translates between C language integers and their external representations.

## Parameters

Item	Description
------	-------------

<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
<i>ip</i>	Specifies the address of the integer.

## Return Values

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

## Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_long Subroutine

### Purpose

Translates between C language long integers and their external representations.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_long  
( xdrs, lp )  
XDR *xdrs;  
long *lp;
```

### Description

The **xdr\_long** filter primitive translates between C language long integers and their external representations. This primitive is characteristic of most eXternal Data Representation (XDR) library primitives and all client XDR routines.

### Parameters

Item	Description
------	-------------

- |             |   |
|-------------|---|
| <i>xdrs</i> | Points to the XDR stream handle. This parameter can be treated as an opaque handler and passed to the primitive routines. |
| <i>lp</i>   | Specifies the address of the number.  |

### Return Values

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

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

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_opaque Subroutine

### Purpose

Translates between fixed-size opaque data and its external representation.

### Library

C Library (**libc.a**)

## Syntax

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

```
xdr_opaque ( xdrs, cp, cnt )  
XDR *xdrs;  
char *cp;  
u_int cnt;
```

## Description

The **xdr\_opaque** subroutine is a filter primitive that translates between fixed-size opaque data and its external representation.

## Parameters

### Item Description

- |             |   |
|-------------|---|
| <i>xdrs</i> | Points to the eXternal Data Representation (XDR) stream handle.   |
| <i>cp</i>   | Specifies the address of the opaque object.   |
| <i>cnt</i>  | Specifies the size, in bytes, of the object. By definition, the actual data contained in the opaque object is not machine-portable. |

## Return Values

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

## Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_opaque\_auth Subroutine

### Purpose

Describes RPC authentication messages.

### Library

C Library (**libc.a**)

## Syntax

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

```
xdr_opaque_auth ( xdrs, ap )  
XDR *xdrs;  
struct opaque_auth *ap;
```

## Description

The **xdr\_opaque\_auth** subroutine describes Remote Procedure Call (RPC) authentication information messages. It generates RPC authentication message data without using the RPC program.

## Parameters

### Item Description

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

*ap* Points to the structure that contains the authentication information.

## Return Values

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

## Related information

[List of RPC Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Remote Procedure Call \(RPC\) Overview for Programming](#)

## xdr\_pmap Subroutine

### Purpose

Describes parameters for **portmap** procedures.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_pmap ( xdrs, regs )
```

```
XDR *xdrs;
```

```
struct pmap *regs;
```

### Description

The **xdr\_pmap** subroutine describes parameters for **portmap** procedures. This subroutine generates **portmap** parameters without using the **portmap** interface.

## Parameters

### Item Description

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

*regs* Points to the buffer or register where the **portmap** daemon stores information.

## Return Values

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

## Related information

[portmap subroutine](#)

[List of RPC Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

## xdr\_pmaplist Subroutine

### Purpose

Describes a list of port mappings externally.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_pmaplist ( xdrs, rp )  
XDR *xdrs;  
struct pmaplist **rp;
```

### Description

The **xdr\_pmaplist** subroutine describes a list of port mappings externally. This subroutine generates the port mappings to Remote Procedure Call (RPC) ports without using the **portmap** interface.

### Parameters

Item	Description
------	-------------

<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
-------------	---

<i>rp</i>	Points to the structure that contains the <b>portmap</b> listings.
-----------	--

### Return Values

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

### Related information

[portmap subroutine](#)

[List of RPC Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

## xdr\_pointer Subroutine

### Purpose

Provides pointer chasing within structures and serializes null pointers.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_pointer ( xdrs, objpp, objsize, xdrobj )  
XDR * xdrs;  
char ** objpp;
```

```
u_int  objsize;
xdrproc_t  xdrobj;
```

## Description

The **xdr\_pointer** subroutine provides pointer chasing within structures and serializes null pointers. This subroutine can represent recursive data structures, such as binary trees or linked lists.

## Parameters

Item	Description
<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
<i>objpp</i>	Points to the character pointer of the data structure.
<i>objsize</i>	Specifies the size of the structure.
<i>xdrobj</i>	Specifies the XDR filter for the object.

## Return Values

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

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Non-Filter Primitives](#)

## xdr\_reference Subroutine

### Purpose

Provides pointer chasing within structures.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_reference ( xdrs, pp, size, proc)
XDR *xdrs;
char **pp;
u_int size;
xdrproc_t proc;
```

### Description

The **xdr\_reference** subroutine is a filter primitive that provides pointer chasing within structures. This primitive allows the serializing, deserializing, and freeing of any pointers within one structure that are referenced by another structure.

The **xdr\_reference** subroutine does not attach special meaning to a null pointer during serialization. Attempting to pass the address of a null pointer can cause a memory error. The programmer must describe data with a two-armed discriminated union. One arm is used when the pointer is valid; the other arm, when the pointer is null.

## Parameters

### Item Description

<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
<i>pp</i>	Specifies the address of the pointer to the structure. When decoding data, XDR allocates storage if the pointer is null.
<i>size</i>	Specifies the byte size of the structure pointed to by the <i>pp</i> parameter.
<i>proc</i>	Translates the structure between its C form and its external representation. This parameter is the XDR procedure that describes the structure.

## Return Values

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

## Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_rejected\_reply Subroutine

### Purpose

Describes RPC message rejection replies.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_rejected_reply ( xdrs, rr )  
XDR *xdrs;  
struct rejected_reply *rr;
```

### Description

The **xdr\_rejected\_reply** subroutine describes Remote Procedure Call (RPC) message rejection replies. This subroutine can be used to generate rejection replies similar to RPC rejection replies without using the RPC program.

## Parameters

### Item Description

<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
<i>rr</i>	Points to the structure that contains the rejected reply.

## Return Values

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

### Related information

[List of RPC Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Remote Procedure Call \(RPC\) Overview for Programming](#)

## xdr\_replymsg Subroutine

### Purpose

Describes RPC message replies.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_replymsg ( xdrs, rmsg )  
XDR *xdrs;  
struct rpc_msg *rmsg;
```

### Description

The **xdr\_replymsg** subroutine describes Remote Procedure Call (RPC) message replies. Use this subroutine to generate message replies similar to RPC message replies without using the RPC program.

### Parameters

Item	Description
------	-------------

<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
-------------	---

<i>rmsg</i>	Points to the structure containing the parameters of the reply message.
-------------	---

### Return Values

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

### Related information

[List of RPC Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Remote Procedure Call \(RPC\) Overview for Programming](#)

## xdr\_setpos Macro

### Purpose

Changes the current position in the XDR stream.

### Library

C Library (**libc.a**)

## Syntax

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

```
xdr_setpos ( xdrs, pos )  
XDR *xdrs;  
u_int pos;
```

## Description

The **xdr\_setpos** macro invokes the set-position routine associated with the eXternal Data Representation (XDR) stream pointed to by the *xdrs* parameter. The new position setting is obtained from the **xdr\_getpos** macro. The **xdr\_setpos** macro returns a value of false if the set position is not valid or if the requested position is out of bounds.

A position cannot be set in some XDR streams. Trying to set a position in such streams causes the macro to fail. This macro also fails if the programmer requests a position that is not in the stream's boundaries.

## Parameters

### Item Description

*xdrs* Points to the XDR stream handle.

*pos* Specifies a position value obtained from the **xdr\_getpos** macro.

## Return Values

Upon successful completion (if the stream is positioned successfully), this macro returns a value of 1. If unsuccessful, it returns a value of 0.

### Related reference

[xdr\\_getpos Macro](#)

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

## xdr\_short Subroutine

### Purpose

Translates between C language short integers and their external representations.

### Library

C Library (**libc.a**)

### Syntax

```
#include <rpc/xdr.h>  
xdr_short ( xdrs, sp )  
XDR *xdrs;  
short *sp;
```

### Description

The **xdr\_short** subroutine is a filter primitive that translates between C language short integers and their external representations.

## Parameters

Item	Description
------	-------------

<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
-------------	---

<i>sp</i>	Specifies the address of the short integer.
-----------	---

## Return Values

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

## Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_string Subroutine

### Purpose

Translates between C language strings and their external representations.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_string ( xdrs, sp, maxsize )  
XDR *xdrs;  
char **sp;  
u_int maxsize;
```

### Description

The **xdr\_string** subroutine is a filter primitive that translates between C language strings and their corresponding external representations. Externally, strings are represented as sequences of ASCII characters, while internally, they are represented with character pointers.

## Parameters

Item	Description
------	-------------

<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
-------------	---

<i>sp</i>	Specifies the address of the pointer to the string.
-----------	---

<i>maxsize</i>	Specifies the maximum length of the string allowed during encoding or decoding. This value is set in a protocol. For example, if a protocol specifies that a file name cannot be longer than 255 characters, then a string cannot exceed 255 characters.
----------------	--

## Return Values

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

### Related reference

[xdr\\_wrapstring Subroutine](#)

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_u\_char Subroutine

### Purpose

Translates between unsigned C language characters and their external representations.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_u_char ( xdrs, ucp)  
XDR *xdrs;  
char *ucp;
```

### Description

The **xdr\_u\_char** subroutine is a filter primitive that translates between unsigned C language characters and their external representations.

### Parameters

Item	Description
------	-------------

<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
-------------	---

<i>ucp</i>	Points to an unsigned integer.
------------	--------------------------------

### Return Values

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

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_u\_int Subroutine

### Purpose

Translates between C language unsigned integers and their external representations.

### Library

C Library (**libc.a**)

## Syntax

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

```
xdr_u_int ( xdrs, up)  
XDR *xdrs;  
u_int *up;
```

## Description

The **xdr\_u\_int** subroutine is a filter primitive that translates between C language unsigned integers and their external representations.

## Parameters

### Item Description

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

*up* Specifies the address of the unsigned long integer.

## Return Values

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

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_u\_long Subroutine

### Purpose

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

### Library

C Library (**libc.a**)

## Syntax

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

```
xdr_u_long ( xdrs, ulp)  
XDR *xdrs;  
u_long *ulp;
```

## Description

The **xdr\_u\_long** subroutine is a filter primitive that translates the unsigned long integers from the C language to their external representations.

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

## Parameters

### Item Description

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

*ulp* Specifies the address of the unsigned long integer.

## Return Values

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

## Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_u\_short Subroutine

### Purpose

Translates between C language unsigned short integers and their external representations.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_u_short ( xdrs, usp )
```

```
XDR *xdrs;
```

```
u_short *usp;
```

### Description

The **xdr\_u\_short** subroutine is a filter primitive that translates between C language unsigned short integers and their external representations.

## Parameters

### Item Description

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

*usp* Specifies the address of the unsigned short integer.

## Return Values

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

## Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

# xdr\_union Subroutine

## Purpose

Translates between discriminated unions and their external representations.

## Library

C Library (**libc.a**)

## Syntax

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

```
xdr_union (xdrs, dscmp, unp, armchoices, defaultarm)  
XDR * xdrs;  
enum_t * dscmp;  
char * unp;  
struct xdr_discrim * armchoices;  
xdrproc_t (* defaultarm);
```

## Description

The **xdr\_union** subroutine is a filter primitive that translates between discriminated C unions and their corresponding external representations. It first translates the discriminant of the union located at the address pointed to by the *dscmp* parameter. This discriminant is always an **enum\_t** value. Next, this subroutine translates the union located at the address pointed to by the *unp* parameter.

The *armchoices* parameter is a pointer to an array of **xdr\_discrim** structures. Each structure contains an ordered pair of parameters [*value*, *proc*]. If the union's discriminant is equal to the associated value, then the specified process is called to translate the union. The end of the **xdr\_discrim** structure array is denoted by a routine having a null value. If the discriminant is not found in the choices array, then the *defaultarm* structure is called (if it is not null).

## Parameters

Item	Description
<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
<i>dscmp</i>	Specifies the address of the union's discriminant. The discriminant is an enumeration ( <b>enum_t</b> ) value.
<i>unp</i>	Specifies the address of the union.
<i>armchoices</i>	Points to an array of <b>xdr_discrim</b> structures.
<i>defaultarm</i>	A structure provided in case no discriminants are found. This parameter can have a null value.

## Return Values

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

## Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_vector Subroutine

### Purpose

Translates between fixed-length arrays and their corresponding external representations.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_vector (xdrs, arrp, size, elsize, elproc)  
XDR * xdrs;  
char * arrp;  
u_int size, elsize;  
xdrproc_t elproc;
```

### Description

The **xdr\_vector** subroutine is a filter primitive that translates between fixed-length arrays and their corresponding external representations.

### Parameters

Item	Description
<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
<i>arrp</i>	Specifies the pointer to the array.
<i>size</i>	Specifies the element count of the array.
<i>elsize</i>	Specifies the size of each of the array elements.
<i>elproc</i>	Translates between the C form of the array elements and their external representation. This is an XDR filter.

### Return Values

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

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_void Subroutine

### Purpose

Supplies an XDR subroutine to the RPC system without transmitting data.

### Library

C Library (**libc.a**)

## Syntax

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

```
xdr_void ()
```

## Description

The **xdr\_void** subroutine has no function parameters. It is passed to other Remote Procedure Call (RPC) subroutines that require a function parameter, but does not transmit data.

## Return Values

This subroutine always returns a value of 1.

## Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdr\_wrapstring Subroutine

### Purpose

Calls the **xdr\_string** subroutine.

### Library

C Library (**libc.a**)

## Syntax

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

```
xdr_wrapstring ( xdrs, sp )  
XDR *xdrs;  
char **sp;
```

## Description

The **xdr\_wrapstring** subroutine is a primitive that calls the **xdr\_string** subroutine (*xdrs*, *sp*, *MAXUN.UNSIGNED*), where the *MAXUN.UNSIGNED* value is the maximum value of an unsigned integer. The **xdr\_wrapstring** subroutine is useful because the Remote Procedure Call (RPC) package passes a maximum of two eXternal Data Representation (XDR) subroutines as parameters, and the **xdr\_string** subroutine requires three.

## Parameters

### Item Description

*xdrs* Points to the XDR stream handle.

*sp* Specifies the address of the pointer to the string.

## Return Values

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

### Related reference

[xdr\\_string Subroutine](#)

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

## xdr\_authunix\_parms Subroutine

### Purpose

Describes UNIX-style credentials.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdr_authunix_parms ( xdrs, app )  
XDR *xdrs;  
struct authunix_parms *app;
```

### Description

The **xdr\_authunix\_parms** subroutine describes UNIX-style credentials. This subroutine generates credentials without using the Remote Procedure Call (RPC) authentication program.

### Parameters

Item	Description
------	-------------

<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
-------------	---

<i>app</i>	Points to the structure that contains the UNIX-style authentication credentials.
------------	--

### Return Values

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

### Related information

[List of RPC Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Remote Procedure Call \(RPC\) Overview for Programming](#)

## xdr\_double Subroutine

### Purpose

Translates between C language double-precision numbers and their external representations.

### Library

C Library (**libc.a**)

## Syntax

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

```
xdr_double ( xdrs, dp)  
XDR *xdrs;  
double *dp;
```

## Description

The **xdr\_double** subroutine is a filter primitive that translates between C language double-precision numbers and their external representations.

## Parameters

### Item Description

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

*dp* Specifies the address of the double-precision number.

## Return Values

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

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Library Filter Primitives](#)

## xdrmem\_create Subroutine

### Purpose

Initializes in local memory the XDR stream pointed to by the *xdrs* parameter.

### Library

C Library (**libc.a**)

### Syntax

```
#include <rpc/xdr.h>  
void  
xdrmem_create ( xdrs, addr, size, op)  
XDR *xdrs;  
char *addr;  
u_int size;  
enum xdr_op op;
```

### Description

The **xdrmem\_create** subroutine initializes in local memory the eXternal Data Representation (XDR) stream pointed to by the *xdrs* parameter. The XDR stream data is written to or read from a chunk of memory at the location specified by the *addr* parameter.

## Parameters

Item	Description
------	-------------

<i>xdrs</i>	Points to the XDR stream handle.
<i>addr</i>	Points to the memory where the XDR stream data is written to or read from.
<i>size</i>	Specifies the length of the memory in bytes.
<i>op</i>	Specifies the XDR direction. The possible choices are <b>XDR_ENCODE</b> , <b>XDR_DECODE</b> , or <b>XDR_FREE</b> .

## Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Non-Filter Primitives](#)

## xdrrec\_create Subroutine

### Purpose

Provides an XDR stream that can contain long sequences of records.

### Library

C Library (**libc.a**)

### Syntax

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

**void**

**xdrrec\_create** (*xdrs*, *sendsize*, *recvsize*, *handle*, *readit*, *writeit*)

**XDR \*** *xdrs*;

**u\_int** *sendsize*;

**u\_int** *recvsize*;

**char \*** *handle*;

**int (\*** *readit* **)** (), **(\*** *writeit* **)** ();

### Description

The **xdrrec\_create** subroutine provides an eXternal Data Representation (XDR) stream that can contain long sequences of records and handle them in both the encoding and decoding directions. The record contents contain data in XDR form. The routine initializes the XDR stream object pointed to by the *xdrs* parameter.

**Note:** This XDR stream implements an intermediate record stream. As a result, additional bytes are in the stream to provide record boundary information.

## Parameters

Item	Description
------	-------------

<i>xdrs</i>	Points to the XDR stream handle.
<i>sendsize</i>	Sets the size of the input buffer to which data is written. If 0 is specified, the buffers are set to the system defaults.

Item	Description
<i>recvsize</i>	Sets the size of the output buffer from which data is read. If 0 is specified, the buffers are set to the system defaults.
<i>handle</i>	Points to the input/output buffer's handle, which is opaque.
<i>readit</i>	Points to the subroutine to call when a buffer needs to be filled. Similar to the <b>read</b> system call.
<i>writeit</i>	Points to the subroutine to call when a buffer needs to be flushed. Similar to the <b>write</b> system call.

### Related reference

[xdrrec\\_endofrecord Subroutine](#)

[xdrrec\\_eof Subroutine](#)

[xdrrec\\_skiprecord Subroutine](#)

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Non-Filter Primitives](#)

## xdrrec\_endofrecord Subroutine

### Purpose

Causes the current outgoing data to be marked as a record.

### Library

C Library (**libc.a**)

### Syntax

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

```
xdrrec_endofrecord ( xdrs, sendnow )
XDR *xdrs;
bool_t sendnow;
```

### Description

The **xdrrec\_endofrecord** subroutine causes the current outgoing data to be marked as a record and can only be invoked on streams created by the **xdrrec\_create** subroutine. If the value of the *sendnow* parameter is nonzero, the data in the output buffer is marked as a completed record and the output buffer is optionally written out.

### Parameters

Item	Description
<i>xdrs</i>	Points to the eXternal Data Representation (XDR) stream handle.
<i>sendnow</i>	Specifies whether the record should be flushed to the output <b>tcp</b> stream.

### Return Values

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

**Related reference**

[xdrrec\\_create Subroutine](#)

**Related information**

[List of XDR Programming References](#)

[Understanding XDR Non-Filter Primitives](#)

## xdrrec\_eof Subroutine

**Purpose**

Checks the buffer for an input stream that indicates the end of file (EOF).

**Library**

C Library (**libc.a**)

**Syntax**

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

```
xdrrec_eof ( xdrs )  
XDR *xdrs;
```

**Description**

The **xdrrec\_eof** subroutine checks the buffer for an input stream to see if the stream reached the end of the file. This subroutine can only be invoked on streams created by the **xdrrec\_create** subroutine.

**Parameters****Item Description**

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

**Return Values**

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

**Related reference**

[xdrrec\\_create Subroutine](#)

**Related information**

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

## xdrrec\_skiprecord Subroutine

**Purpose**

Causes the position of an input stream to move to the beginning of the next record.

**Library**

C Library (**libc.a**)

## Syntax

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

```
xdrrec_skiprecord ( xdrs)  
XDR *xdrs;
```

## Description

The **xdrrec\_skiprecord** subroutine causes the position of an input stream to move past the current record boundary and onto the beginning of the next record of the stream. This subroutine can only be invoked on streams created by the **xdrrec\_create** subroutine. The **xdrrec\_skiprecord** subroutine tells the eXternal Data Representation (XDR) implementation that the rest of the current record in the stream's input buffer should be discarded.

## Parameters

### Item Description

*xdrs* Points to the XDR stream handle.

## Return Values

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

### Related reference

[xdrrec\\_create Subroutine](#)

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Non-Filter Primitives](#)

## xdrstdio\_create Subroutine

### Purpose

Initializes the XDR data stream pointed to by the *xdrs* parameter.

### Library

C Library (**libc.a**)

### Syntax

```
#include <stdio.h>  
#include <rpc/xdr.h>  
void xdrstdio_create ( xdrs, file, op)  
XDR *xdrs;  
FILE *file;  
enum xdr_op op;
```

### Description

The **xdrstdio\_create** subroutine initializes the eXternal Data Representation (XDR) data stream pointed to by the *xdrs* parameter. The XDR stream data is written to or read from the standard input/output stream pointed to by the *file* parameter.

**Note:** The destroy routine associated with such an XDR stream calls the **fflush** function on the *file* stream, but never calls the **fclose** function.

## Parameters

### Item Description

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

### Related information

[List of XDR Programming References](#)

[eXternal Data Representation \(XDR\) Overview for Programming](#)

[Understanding XDR Non-Filter Primitives](#)

## Network Information Services

---

This topic collection includes subroutines that derive information from the Network Information Services lookup.

### yp\_all Subroutine

#### Purpose

Transfers all of the key-value pairs from the Network Information Services (NIS) server to the client as the entire map.

#### Library

C Library (**libc.a**)

#### Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

```
yp_all ( indomain, inmap, incallback )
char *indomain;
char *inmap;
struct ypall_Callback *incallback {
int (* foreach) ();
char * data;
};
```

```
foreach ( instatus, inkey, inkeylen, inval, invallen, indata )
int instatus;
char * inkey;
int inkeylen;
char * inval;
int invallen;
char * indata;
```

## Description

The **yp\_all** subroutine provides a way to transfer an entire map from the server to the client in a single request. The routine uses Transmission Control Protocol (TCP) rather than User Datagram Protocol (UDP) used by other NIS subroutines. This entire transaction takes place as a single Remote Procedure Call (RPC) request and response. The **yp\_all** subroutine is used like any other NIS procedure, identifying a subroutine and map in the normal manner, and supplying a subroutine to process each key-value pair within the map.

The memory pointed to by the *inkey* and *inval* parameters is private to the **yp\_all** subroutine. This memory is overwritten with each new key-value pair processed. The **foreach** function uses the contents of the memory but does not own the memory itself. Key and value objects presented to the **foreach** function look exactly as they do in the server's map. Objects not terminated by a new-line or null character in the server's map are not terminated by a new-line or null character in the client's map.

**Note:** The remote procedure call is returned to the **yp\_all** subroutine only after the transaction is completed (successfully or unsuccessfully) or after the **foreach** function rejects any more key-value pairs.

## Parameters

Item	Description
<i>data</i>	Specifies state information between the <b>foreach</b> function and the mainline code (see also the <i>indata</i> parameter).
<i>indomain</i>	Points to the name of the domain used as input to the subroutine.
<i>inmap</i>	Points to the name of the map used as input to the subroutine.
<i>incallback</i>	Specifies the structure containing the user-defined <b>foreach</b> function, which is called for each key-value pair transferred.
<i>instatus</i>	Specifies either a return status value of the form <b>NIS_TRUE</b> or an error code. The error codes are defined in the <b>rpcsvc/yp_prot.h</b> file.
<i>inkey</i>	Points to the current key of the key-value pair as returned from the server's database.
<i>inkeylen</i>	Returns the length, in bytes, of the <i>inkey</i> parameter.
<i>inval</i>	Points to the current value of the key-value pair as returned from the server's database.
<i>invallen</i>	Specifies the size of the value in bytes.
<i>indata</i>	Specifies the contents of the <b>incallback-&gt;data</b> element passed to the <b>yp_all</b> subroutine. The <b>data</b> element shares state information between the <b>foreach</b> function and the mainline code. The <i>indata</i> parameter is optional because no part of the NIS client package inspects its contents.

## Return Values

The **foreach** subroutine returns a value of 0 when it is ready to be called again for additional received key-value pairs. It returns a nonzero value to stop the flow of key-value pairs. If the **foreach** function returns a nonzero value, it is not called again, and the **yp\_all** subroutine returns a value of 0.

## Related information

[Network Information Service \(NIS\) Overview for System Management](#)  
[Remote Procedure Call \(RPC\) Overview for Programming](#)

# yp\_bind Subroutine

## Purpose

Used in programs to call the **ypbind** daemon directly for processes that use backup strategies when Network Information Services (NIS) is not available.

## Library

C Library (**libc.a**)

## Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

```
yp_bind ( indomain)
char *indomain;
```

## Description

In order to use NIS, the client process must be bound to an NIS server that serves the appropriate domain. That is, the client must be associated with a specific NIS server that services the client's requests for NIS information. The NIS lookup processes automatically use the **ypbind** daemon to bind the client, but the **yp\_bind** subroutine can be used in programs to call the daemon directly for processes that use backup strategies (for example, a local file) when NIS is not available.

Each NIS binding allocates, or uses up, one client process socket descriptor, and each bound domain uses one socket descriptor. Multiple requests to the same domain use the same descriptor.

**Note:** If a Remote Procedure Call (RPC) failure status returns from the use of the **yp\_bind** subroutine, the domain is unbound automatically. When this occurs, the NIS client tries to complete the operation if the **ypbind** daemon is running and either of the following is true:

- The client process cannot bind a server for the proper domain.
- RPCs to the server fail.

## Parameters

Item	Description
<i>indomain</i>	Points to the name of the domain for which to attempt the bind.

## Return Values

The NIS client returns control to the user with either an error or a success code if any of the following occurs:

- The error is not related to RPC.
- The **ypbind** daemon is not running.
- The **ypserv** daemon returns the answer.

## Related information

[ypbind command](#)

[Network Information Service \(NIS\) Overview for System Management](#)

[Remote Procedure Call \(RPC\) Overview for Programming](#)

## yp\_first Subroutine

### Purpose

Returns the first key-value pair from the named Network Information Services (NIS) map in the named domain.

### Library

C Library (**libc.a**)

### Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

```
yp_first (indomain, inmap, outkey, outkeylen, outval, outvallen)
char * indomain;
char * inmap;
char ** outkey;
int * outkeylen;
char ** outval;
int * outvallen;
```

### Description

The **yp\_first** routine returns the first key-value pair from the named NIS map in the named domain.

### Parameters

Item	Description
<i>indomain</i>	Points to the name of the domain used as input to the subroutine.
<i>inmap</i>	Points to the name of the map used as input to the subroutine.
<i>outkey</i>	Specifies the address of the uninitialized string pointer where the first key is returned. Memory is allocated by the NIS client using the <b>malloc</b> subroutine, and may be freed by the application.
<i>outkeylen</i>	Returns the length, in bytes, of the <i>outkey</i> parameter.
<i>outval</i>	Specifies the address of the uninitialized string pointer where the value associated with the key is returned. Memory is allocated by the NIS client using the <b>malloc</b> subroutine, and may be freed by the application.
<i>outvallen</i>	Returns the length, in bytes, of the <i>outval</i> parameter.

### Return Values

Upon successful completion, this subroutine returns a value of 0. If unsuccessful, it returns an error as described in the **rpcsvc/yp\_prot.h** file.

### Related information

[malloc subroutine](#)

[Network Information Service \(NIS\) Overview for System Management](#)

[Remote Procedure Call \(RPC\) Overview for Programming](#)

## yp\_get\_default\_domain Subroutine

### Purpose

Gets the default domain of the node.

### Library

C Library (**libc.a**)

### Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

```
yp_get_default_domain ( outdomain )
char **outdomain;
```

### Description

Network Information Services (NIS) lookup calls require both a map name and a domain name. Client processes can get the default domain of the node by calling the **yp\_get\_default\_domain** routine and using the value returned in the *outdomain* parameter as the input domain (*indomain*) parameter for NIS remote procedure calls.

### Parameters

Item	Description
<i>outdomain</i>	Specifies the address of the uninitialized string pointer where the default domain is returned. Memory is allocated by the NIS client using the <b>malloc</b> subroutine and should not be freed by the application.

### Return Values

Upon successful completion, this routine returns a value of 0. If unsuccessful, it returns an error as described in the **rpcsvc/ypclnt.h** file.

### Related information

[malloc subroutine](#)

[Network Information Service \(NIS\) Overview for System Management](#)

[Remote Procedure Call \(RPC\) Overview for Programming](#)

## yp\_master Subroutine

### Purpose

Returns the machine name of the Network Information Services (NIS) master server for a map.

### Library

C Library (**libc.a**)

### Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

```
yp_master ( indomain, inmap, outname)
char *indomain;
char *inmap;
char **outname;
```

## Description

The **yp\_master** subroutine returns the machine name of the NIS master server for a map.

## Parameters

Item	Description
<i>indomain</i>	Points to the name of the domain used as input to the subroutine.
<i>inmap</i>	Points to the name of the map used as input to the subroutine.
<i>outname</i>	Specifies the address of the uninitialized string pointer where the name of the domain's <b>yp_master</b> server is returned. Memory is allocated by the NIS client using the <b>malloc</b> subroutine, and may be freed by the application.

## Return Values

Upon successful completion, this subroutine returns a value of 0. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp\_prot.h** file.

### Related information

[malloc subroutine](#)

[Network Information Service \(NIS\) Overview for System Management](#)

[Remote Procedure Call \(RPC\) Overview for Programming](#)

## yp\_match Subroutine

### Purpose

Searches for the value associated with a key.

### Library

C Library (**libc.a**)

### Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

```
yp_match (indomain, inmap, inkey, inkeylen, outval, outvallen)
char * indomain;
char * inmap;
char * inkey;
int inkeylen;
char ** outval;
int * outvallen;
```

## Description

The **yp\_match** subroutine searches for the value associated with a key. The input character string entered as the key must match a key in the Network Information Services (NIS) map exactly because pattern matching is not available in NIS.

## Parameters

Item	Description
<i>indomain</i>	Points to the name of the domain used as input to the subroutine.
<i>inmap</i>	Points to the name of the map used as input to the subroutine.
<i>inkey</i>	Points to the name of the key used as input to the subroutine.
<i>inkeylen</i>	Specifies the length, in bytes, of the key.
<i>outval</i>	Specifies the address of the uninitialized string pointer where the values associated with the key are returned. Memory is allocated by the NIS client using the <b>malloc</b> subroutine, and may be freed by the application.
<i>outvallen</i>	Returns the length, in bytes, of the <i>outval</i> parameter.

## Return Values

Upon successful completion, this subroutine returns a value of 0. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp\_prot.h** file.

## Related information

[malloc subroutine](#)

[Network Information Service \(NIS\) Overview for System Management](#)

[Remote Procedure Call \(RPC\) Overview for Programming](#)

## yp\_next Subroutine

### Purpose

Returns each subsequent value it finds in the named Network Information Services (NIS) map until it reaches the end of the list.

### Library

C Library (**libc.a**)

### Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

```
yp_next (indomain, inmap, inkey, inkeylen, outkey, outkeylen, outval,
outvallen)
char * indomain;
char * inmap;
char * inkey;
int inkeylen;
char ** outkey;
int * outkeylen;
char ** outval;
int * outvallen;
```

## Description

The **yp\_next** subroutine returns each subsequent value it finds in the named NIS map until it reaches the end of the list.

The **yp\_next** subroutine must be preceded by an initial **yp\_first** subroutine. Use the *outkey* parameter value returned from the initial **yp\_first** subroutine as the value of the *inkey* parameter for the **yp\_next** subroutine. This will return the second key-value pair associated with the map. To show every entry in the NIS map, the **yp\_first** subroutine is called with the **yp\_next** subroutine called repeatedly. Each time the **yp\_next** subroutine returns a key-value, use it as the *inkey* parameter for the next call.

The concepts of *first* and *next* depend on the structure of the NIS map being processed. The routines do not retrieve the information in a specific order, such as the lexical order from the original, non-NIS database information files or the numerical sorting order of the keys, values, or key-value pairs. If the **yp\_first** subroutine is called on a specific map with the **yp\_next** subroutine called repeatedly until the process returns a **YPERR\_NOMORE** message, every entry in the NIS map is seen once. If the same sequence of operations is performed on the same map at the same server, the entries are seen in the same order.

**Note:** If a server operates under a heavy load or fails, the domain can become unbound and then bound again while a client is running. If it binds itself to a different server, entries may be seen twice or not at all. The domain rebinds itself to protect the enumeration process from being interrupted before it completes. Avoid this situation by returning all of the keys and values with the **yp\_all** subroutine.

## Parameters

Item	Description
<i>indomain</i>	Points to the name of the domain used as input to the subroutine.
<i>inmap</i>	Points to the name of the map used as input to the subroutine.
<i>inkey</i>	Points to the key that is used as input to the subroutine.
<i>inkeylen</i>	Returns the length, in bytes, of the <i>inkey</i> parameter.
<i>outkey</i>	Specifies the address of the uninitialized string pointer where the first key is returned. Memory is allocated by the NIS client using the <b>malloc</b> subroutine, and may be freed by the application.
<i>outkeylen</i>	Returns the length, in bytes, of the <i>outkey</i> parameter.
<i>outval</i>	Specifies the address of the uninitialized string pointer where the values associated with the key are returned. Memory is allocated by the NIS client using the <b>malloc</b> subroutine, and may be freed by the application.
<i>outvallen</i>	Returns the length, in bytes, of the <i>outval</i> parameter.

## Return Values

Upon successful completion, this routine returns a value of 0. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp\_prot.h** file.

## Related information

[malloc subroutine](#)

[Network Information Service \(NIS\) Overview for System Management](#)

[Remote Procedure Call \(RPC\) Overview for Programming](#)

## yp\_order Subroutine

### Purpose

Returns the order number for an Network Information Services (NIS) map that identifies when the map was built.

### Library

C Library (**libc.a**)

### Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

```
yp_order (indomain, inmap, outorder)
char * indomain;
char * inmap;
int * outorder;
```

### Description

The **yp\_order** subroutine returns the order number for a NIS map that identifies when the map was built. The number determines whether the local NIS map is more current than the master NIS database.

### Parameters

Item	Description
<i>indomain</i>	Points to the name of the domain used as input to the subroutine.
<i>inmap</i>	Points to the name of the map used as input to the subroutine.
<i>outorder</i>	Points to the returned order number, which is a 10-digit ASCII integer that represents the operating system time, in seconds, when the map was built.

### Return Values

Upon successful completion, this routine returns a value of 0. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp\_prot.h** file.

### Related information

[Network Information Service \(NIS\) Overview for System Management](#)  
[Remote Procedure Call \(RPC\) Overview for Programming](#)

## yp\_unbind Subroutine

### Purpose

Manages socket descriptors for processes that access multiple domains.

### Library

C Library (**libc.a**)

## Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

```
void yp_unbind ( indomain)
char *indomain;
```

## Description

The **yp\_unbind** subroutine is available to manage socket descriptors for processes that access multiple domains. When the **yp\_unbind** subroutine is used to free a domain, all per-process and per-node resources that were used to bind the domain are also freed.

## Parameters

Item	Description
<i>indomain</i>	Points to the name of the domain used as input to the subroutine.

## Return Values

Upon successful completion, this routine returns a value of 0. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp\_prot.h** file.

## Related information

[ypbind command](#)

[Remote Procedure Call \(RPC\) Overview for Programming](#)

[Sockets Overview](#)

## yp\_update Subroutine

### Purpose

Makes changes to an Network Information Services (NIS) map.

### Library

C Library (**libc.a**)

## Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

```
yp_update (indomain, inmap, yprop, inkey, inkeylen, indata, indatalen)
char * indomain;
char * inmap;
unsigned yprop;
char * inkey;
int inkeylen;
char * indata;
int indatalen;
```

## Description

**Note:** This routine depends upon the secure Remote Procedure Call (RPC) protocol, and will not work unless the network is running it.

The **yp\_update** subroutine is used to make changes to a NIS map. The syntax is the same as that of the **yp\_match** subroutine except for the additional *ypop* parameter, which may take on one of the following four values:

Value	Description
<b>ypop_INSERT</b>	Inserts the key-value pair into the map. If the key already exists in the map, the <b>yp_update</b> subroutine returns a value of <b>YPERR_KEY</b> .
<b>ypop_CHANGE</b>	Changes the data associated with the key to the new value. If the key is not found in the map, the <b>yp_update</b> subroutine returns a value of <b>YPERR_KEY</b> .
<b>ypop_STORE</b>	Stores an item in the map regardless of whether the item already exists. No error is returned in either case.
<b>ypop_DELETE</b>	Deletes an entry from the map.

## Parameters

Item	Description
<i>indomain</i>	Points to the name of the domain used as input to the subroutine.
<i>inmap</i>	Points to the name of the map used as input to the subroutine.
<i>ypop</i>	Specifies the update operation to be used as input to the subroutine.
<i>inkey</i>	Points to the input key to be used as input to the subroutine.
<i>inkeylen</i>	Specifies the length, in bytes, of the <i>inkey</i> parameter.
<i>indata</i>	Points to the data used as input to the subroutine.
<i>indatalen</i>	Specifies the length, in bytes, of the data used as input to the subroutine.

## Return Values

Upon successful completion, this routine returns a value of 0. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp\_prot.h** file.

## Files

Item	Description
<u><a href="#">/var/yp/updaters</a></u>	A makefile for updating NIS maps.

## Related information

[Network Information Service \(NIS\) Overview for System Management](#)  
[Remote Procedure Call \(RPC\) Overview for Programming](#)

## yperr\_string Subroutine

### Purpose

Returns a pointer to an error message string.

### Library

C Library (**libc.a**)

## Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

```
char *yperr_string ( incode)
int incode;
```

## Description

The **yperr\_string** routine returns a pointer to an error message string. The error message string is null-terminated but contains no period or new-line escape characters.

## Parameters

Item	Description
<i>incode</i>	Contains Network Information Services (NIS) error codes as described in the <b>rpcsvc/yp_prot.h</b> file.

## Return Values

This subroutine returns a pointer to an error message string corresponding to the *incode* parameter.

### Related reference

[ypprot\\_err Subroutine](#)

### Related information

[Network Information Service \(NIS\) Overview for System Management](#)

## ypprot\_err Subroutine

### Purpose

Takes an Network Information Services NIS protocol error code as input and returns an error code to be used as input to a **yperr\_string** subroutine.

### Library

C Library (**libc.a**)

## Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

```
ypprot_err ( incode)
u_int incode;
```

## Description

The **ypprot\_err** subroutine takes a NIS protocol error code as input and returns an error code to be used as input to a **yperr\_string** subroutine.

## Parameters

Item	Description
<i>incode</i>	Specifies the NIS protocol error code used as input to the subroutine.

## Return Values

This subroutine returns a corresponding error code to be passed to the **yperr\_string** subroutine.

## Related reference

[yperr\\_string Subroutine](#)

## Related information

[Network Information Service \(NIS\) Overview for System Management](#)

[Remote Procedure Call \(RPC\) Overview for Programming](#)

# Simple Network Management Protocol (SNMP)

---

The Simple Network Management Protocol (SNMP) is used by network hosts to exchange information in the management of networks. SNMP network management is based on the familiar client-server model that is widely used in Transmission Control Protocol/Internet Protocol (TCP/IP)-based network applications. Each managed host runs a process called an agent. The agent is a server process that maintains the MIB database for the host. Hosts that are involved in network management decision-making may run a process called a manager. A manager is a client application that generates requests for MIB information and processes responses. In addition, a manager may send requests to agent servers to modify MIB information.

## getsmuxEntrybyname or getsmuxEntrybyidentity Subroutine

### Purpose

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

### Library

SNMP Library (**libsnmp.a**)

### Syntax

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

```
struct smuxEntry *getsmuxEntrybyname ( name)  
char *name;
```

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

### Description

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

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

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

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

## Parameters

Item	Description
<i>name</i>	Points to a character string that names the SMUX peer.
<i>identity</i>	Specifies the object identifier for a SMUX peer.

## Return Values

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

## Files

Item	Description
<i>/etc/snmpd.peers</i>	Contains the SMUX peer definitions for the network.

## Related information

[List of Network Manager Programming References](#)  
[SNMP Overview for Programmers](#)

## isodetailor Subroutine

### Purpose

Initializes variables for various logging facilities.

### Library

ISODE Library (**libisode.a**)

### Syntax

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

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

### Description

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

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

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

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

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

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

## Parameters

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

## Files

Item	Description
<b>/etc/isodetailor</b>	Location of user's copy of the <b>/usr/lpp/snmpd/smux/isodetailor</b> file.
<b>/usr/lpp/snmpd/smux/isodetailor</b>	Contains a complete list of all the logging parameters.

### Related reference

[ll\\_hdinit, ll\\_dbinit, \\_ll\\_log, or ll\\_log Subroutine](#)

### Related information

[List of Network Manager Programming References](#)

[SNMP Overview for Programmers](#)

## ll\_hdinit, ll\_dbinit, \_ll\_log, or ll\_log Subroutine

### Purpose

Reports errors to log files.

### Library

ISODE Library (**libisode.a**)

### Syntax

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

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

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

```
int _ll_log (lp, event, ap)
```

```

register LLog *lp;
int event;
va_list ap;

int ll_log ( va_alist)
va_dcl

```

## Description

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

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

```

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

```

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

```

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

```

The possible values for the `ll_stat` field are:

```

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

```

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

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

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

The expected parameter format for the **\_ll\_log** and **ll\_log** subroutines is:

- **\_ll\_log**(*lp, event, what, string\_format, ...*);

- `ll_log(lp, event, what, string_format, ...);`

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

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

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

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

where:

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

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

## Parameters

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

## Return Values

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

## Related reference

[isodetailor Subroutine](#)

## Related information

[List of Network Manager Programming References](#)

[Examples of SMUX Error Logging Routines](#)

[SNMP Overview for Programmers](#)

# **o\_number, o\_integer, o\_string, o\_igeneric, o\_generic, o\_specific, or o\_ipaddr Subroutine**

## Purpose

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

## Library

SNMP Library (**libsnmp.a**)

## Syntax

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

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

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

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

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

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

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

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

## Description

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

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

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

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

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

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

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

## Parameters

Item	Description
<i>oi</i>	Contains the <b>OI</b> data structure for the variable whose value is to be recorded into the binding structure.
<i>v</i>	Specifies the variable binding parameter, which is of type <b>type_SNMP_VarBind</b> . The <i>v</i> parameter contains a name and a value field. The value field contents are supplied by the <b>o_</b> subroutines.
<i>number</i>	Contains an integer to store in the value field of the <i>v</i> (variable bind) parameter.
<i>base</i>	Points to the character string to store in the value field of the <i>v</i> parameter.
<i>len</i>	Designates the length of the integer character string to copy. The character string is described by the <i>base</i> parameter.

Item	Description
<i>offset</i>	Contains an integer value of the current type of request, for example: <pre>type_SNMP_PDUs_get__next__request</pre>
<i>value</i>	Contains a character pointer to a value.
<i>netaddr</i>	Points to a <b>sockaddr_in</b> structure. The subroutine only uses the <code>sin_addr</code> field of this structure.

## Return Values

The return values for these subroutines are:

Value	Description
<b>int_SNMP_error__status_genErr</b>	Indicates an error occurred when setting the <i>v</i> parameter value.
<b>int_SNMP_error__status_noErr</b>	Indicates no errors found.

### Related reference

[s\\_generic Subroutine](#)

### Related information

[List of Network Manager Programming References](#)

[SNMP Overview for Programmers](#)

[Working with Management Information Base \(MIB\) Variables](#)

## oid\_cmp, oid\_cpy, oid\_free, sprintoid, str2oid, ode2oid, oid2ode, oid2ode\_aux, prim2oid, or oid2prim Subroutine

### Purpose

Manipulates the object identifier data structure.

### Library

ISODE Library (**libisode.a**)

### Syntax

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

```
int oid_cmp (p, q)
```

```
OID p, q;
```

```
OID oid_cpy (oid)
```

```
OID oid;
```

```
void oid_free (oid)
```

```
OID oid;
```

```
char *sprintoid (oid)
```

```
OID oid;
```

```
OID str2oid (s)
```

```
char * s;
```

**OID ode2oid** (*descriptor*)  
**char \*** *descriptor*;

```
char *oid2ode (oid)
OID oid;
```

**OID \*oid2ode\_aux** (*descriptor, quote*)  
**char \*** *descriptor*;  
**int** *quote*;

**OID prim2oid** (*pe*)  
**PE** *pe*;

```
PE oid2prim (oid)
OID oid;
```

## Description

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

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

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

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

The **oid2ode** subroutine is identical to the **sprintoid** subroutine except that the **oid2ode** subroutine checks whether the *oid* parameter is in the **isobjects** database. The **oid2ode** subroutine is implemented as a macro call to the **oid2ode\_aux** subroutine. The **oid2ode\_aux** subroutine is similar to the **oid2ode** subroutine except for an additional integer parameter that specifies whether the string should be enclosed by quotes. The **oid2ode** subroutine always encloses the string in quotes.

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

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

## Parameters

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

## Return Values

The **oid\_cmp** subroutine returns a 0 if the structures are identical, -1 if the first object is less than the second, and a 1 if any other conditions are found. The **oid\_cpy** subroutine returns a pointer to the designated object identifier when the subroutine is successful.

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

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

### Related reference

[oid\\_extend](#) or [oid\\_normalize](#) Subroutine

[text2oid](#) or [text2obj](#) Subroutine

### Related information

[List of Network Manager Programming References](#)

[SNMP Overview for Programmers](#)

## oid\_extend or oid\_normalize Subroutine

### Purpose

Extends the base ISODE library subroutines.

### Library

SNMP Library (**libsnmp.a**)

### Syntax

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

```
OID oid_extend (q, howmuch)
```

```
OID q;
```

```
integer howmuch;
```

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

```
OID q;
```

```
integer howmuch, initial;
```

### Description

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

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

These subroutines do not free the *q* parameter.

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

## Parameters

Item	Description
<i>q</i>	Specifies the size of the original array.
<i>howmuch</i>	Specifies the size of the array for the new <b>OID</b> structure.
<i>initial</i>	Indicates the initialized value of the <b>OID</b> structure extensions.

## Return Values

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

### Related reference

[oid\\_cmp](#), [oid\\_cpy](#), [oid\\_free](#), [sprintoid](#), [str2oid](#), [ode2oid](#), [oid2ode](#), [oid2ode\\_aux](#), [prim2oid](#), or [oid2prim](#)  
Subroutine

### Related information

[List of Network Manager Programming References](#)  
[SNMP Overview for Programmers](#)

## readobjects Subroutine

### Purpose

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

### Library

SNMP Library (**libsnmp.a**)

### Syntax

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

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

### Description

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

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

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

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

## Parameters

### Item Description

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

## Return Values

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

### Related reference

[text2oid or text2obj Subroutine](#)

[smux\\_free\\_tree Subroutine](#)

### Related information

[List of Network Manager Programming References](#)

[SNMP Overview for Programmers](#)

## s\_generic Subroutine

### Purpose

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

### Library

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

### Syntax

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

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

### Description

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

The *offset* parameter is used to determine the stage of the set process. If the *offset* parameter value is **type\_SNMP\_PDUs\_set\_\_request**, the value is checked for validity and the value in the *ot\_save* field in the **OI** structure is set. If the *offset* parameter value is **type\_SNMP\_PDUs\_commit**, the value in the *ot\_save* field is freed and moved to the MIB *ot\_info* field. If the *offset* parameter value is **type\_SNMP\_PDUs\_rollback**, the value in the *ot\_save* field is freed and no new value is written.

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

## Parameters

### Item Description

*oi* Designates the **OI** structure representing the MIB variable to be set.

Item	Description
<i>v</i>	Specifies the variable binding that contains the value to be set.
<i>offset</i>	Contains the stage of the set. The possible values for the <i>offset</i> parameter are <b>type_SNMP_PDUs_commit</b> , <b>type_SNMP_PDUs_rollback</b> , or <b>type_SNMP_PDUs_set__request</b> .

## Return Values

If the subroutine is successful, a value of **int\_SNMP\_error\_\_status\_noError** is returned. Otherwise, a value of **int\_SNMP\_error\_\_status\_badValue** is returned.

### Related reference

[o\\_number, o\\_integer, o\\_string, o\\_igeneric, o\\_generic, o\\_specific, or o\\_ipaddr Subroutine](#)

### Related information

[List of Network Manager Programming References](#)

[SNMP Overview for Programmers](#)

[SNMP daemon processing](#)

## smux\_close Subroutine

### Purpose

Ends communications with the SNMP agent.

### Library

SNMP Library (**libsnmp.a**)

### Syntax

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

```
int smux_close ( reason )
int reason;
```

### Description

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

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

### Parameters

Item	Description
<i>reason</i>	Indicates an integer value denoting the reason the close PDU message is being sent.

### Return Values

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

## Error Codes

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

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

### Related information

[List of Network Manager Programming References](#)

[SNMP Overview for Programmers](#)

## smux\_error Subroutine

### Purpose

Creates a readable string from the **smux\_errno** global variable value.

### Library

SNMP Library (**libsnmp.a**)

### Syntax

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

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

### Description

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

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

### Parameters

Item	Description
<i>error</i>	Contains the error to interpret. Usually called with the value of the <b>smux_errno</b> variable, but can be called with any error that is an integer.

### Return Values

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

## Related information

[List of Network Manager Programming References](#)  
[SNMP Overview for Programmers](#)

# smux\_free\_tree Subroutine

## Purpose

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

## Library

SNMP Library (**libsnmp.a**)

## Syntax

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

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

## Description

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

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

This subroutine does not return values or error codes.

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

## Parameters

Item	Description
<i>parent</i>	Contains a character string holding the immediate parent of the tree to be deleted.
<i>child</i>	Contains a character string holding the beginning of the tree to be deleted.

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

## Related reference

[readobjects Subroutine](#)

## Related information

[snmpd subroutine](#)

[List of Network Manager Programming References](#)  
[SNMP Overview for Programmers](#)

# smux\_init Subroutine

## Purpose

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

## Library

SNMP Library (**libsnmp.a**)

## Syntax

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

```
int smux_init ( debug)  
int debug;
```

## Description

The **smux\_init** subroutine initializes the TCP socket that is used by the SMUX agent to communicate with the SNMP daemon. The subroutine assumes that loopback is used to define the path to the SNMP daemon. Name resolution attempts to find an IPv6 address mapping for loopback. If it cannot find an IPv6 address, it tries to find an IPv4 address for loopback. The subroutine also clears the base structures that the SMUX code uses. The **smux\_init** subroutine also sets the debug level that is used when it runs the SMUX subroutines.

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

## Parameters

Item	Description
------	-------------

<i>debug</i>	Indicates the level of debug to be printed during SMUX subroutines.
--------------	---

## Return Values

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

## Error Codes

Possible values for the **smux\_errno** global variable are:

Value	Description
<b>congestion</b>	Indicates memory allocation problems
<b>youLoseBig</b>	Signifies problem with SNMP library code
<b>systemError</b>	Indicates TCP connection failure.

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

## Related information

[List of Network Manager Programming References](#)  
[SNMP Overview for Programmers](#)

# smux\_register Subroutine

## Purpose

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

## Library

SNMP Library (**libsnmp.a**)

## Syntax

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

```
int smux_register ( subtree, priority, operation )
```

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

## Description

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

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

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

## Parameters

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

## Return Values

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

## Error Codes

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

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

### Related information

[List of Network Manager Programming References](#)

[SNMP Overview for Programmers](#)

## smux\_response Subroutine

### Purpose

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

### Library

SNMP Library (**libsnmp.a**)

### Syntax

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

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

### Description

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

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

### Parameters

Item	Description
<i>event</i>	Specifies a <b>type_SNMP_GetResponse__PDU</b> variable that contains the response PDU to send to the SNMP agent.

### Return Values

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

## Error Codes

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

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

### Related information

[List of Network Manager Programming References](#)

[SNMP Overview for Programmers](#)

## smux\_simple\_open Subroutine

### Purpose

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

### Library

SNMP Library (**libsnmp.a**)

### Syntax

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

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

### Description

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

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

### Parameters

Item	Description
<i>identity</i>	Specifies an object identifier that describes the SMUX peer.

Item	Description
<i>description</i>	Contains a string of characters that describes the SMUX peer. The <i>description</i> parameter value cannot be longer than 254 characters.
<i>commname</i>	Contains the password to be sent to the SNMP agent. Can be a null value.
<i>commlen</i>	Indicates the length of the community name ( <i>commname</i> parameter) to be sent to the SNMP agent. The value for this parameter must be at least 0.

## Return Values

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

## Error Codes

If the subroutine is unsuccessful, the **smux\_errno** global variable is set one of the following values:

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

## Related information

[List of Network Manager Programming References](#)  
[SNMP Overview for Programmers](#)

## smux\_trap Subroutine

### Purpose

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

### Library

SNMP Library (**libsnp.a**)

### Syntax

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

```
int smux_trap ( generic, specific, bindings)
```

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

## Description

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

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

## Parameters

Item	Description
<i>generic</i>	Contains an integer specifying the generic trap type. The value must be one of the following: <b>0</b> Specifies a cold start. <b>1</b> Specifies a warm start. <b>2</b> Specifies a link down. <b>3</b> Specifies a link up. <b>4</b> Specifies an authentication failure. <b>5</b> Specifies an EGP neighbor loss. <b>6</b> Specifies an enterprise-specific trap type.
<i>specific</i>	Contains an integer that uniquely identifies the trap. The unique identity is typically assigned by the registration authority for the enterprise owning the SMUX peer.
<i>bindings</i>	Indicates the variable bindings to assign to the trap protocol data unit (PDU).

## Return Values

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

## Error Codes

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

Value	Description
<b>invalidOperation</b>	Indicates the Transmission Control Protocol (TCP) connection was not completed.
<b>congestion</b>	Indicates memory is not available. The TCP connection was closed.

Value	Description
<b>youLoseBig</b>	Indicates an error occurred in the SNMP code. The TCP connection was closed.

### Related information

[List of Network Manager Programming References](#)  
[SNMP Overview for Programmers](#)

## smux\_wait Subroutine

### Purpose

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

### Library

SNMP Library (**libsnmp.a**)

### Syntax

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

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

### Description

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

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

### Parameters

Item	Description
<i>event</i>	Points to a pointer of <b>type_SMUX_PDUs</b> . This holds the PDUs received by the <b>smux_wait</b> subroutine.
<i>isecs</i>	Specifies an integer value equal to the number of seconds to wait for a message.

### Return Values

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

### Error Codes

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

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

### Related information

[List of Network Manager Programming References](#)  
[SNMP Overview for Programmers](#)

## text2inst, name2inst, next2inst, or nexttot2inst Subroutine

### Purpose

Retrieves instances of variables from various forms of data.

### Library

SNMP Library (**libsnmp.a**)

### Syntax

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

```
OI text2inst ( text)  
char *text;
```

```
OI name2inst ( oid)  
OID oid;
```

```
OI next2inst (oid)  
OID oid;
```

```
OI nexttot2inst (oid, ot)  
OID oid;  
OT ot;
```

### Description

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

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

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

The **next2inst** and **nexttot2inst** subroutines find the next object in the database given an object identifier. The **next2inst** subroutine starts at the root of the tree, while the **nexttot2inst** subroutine starts at the object given in the *ot* parameter. If another object cannot be found, the **NULLOI** value will be returned.

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

## Parameters

### Item Description

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

## Return Values

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

### Related reference

[text2oid or text2obj Subroutine](#)

### Related information

[List of Network Manager Programming References](#)

[SNMP Overview for Programmers](#)

## text2oid or text2obj Subroutine

### Purpose

Converts a text string into some other value.

### Library

SNMP Library (**libsnmp.a**)

### Syntax

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

```
OID text2oid ( text )  
char *text;
```

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

### Description

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

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

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

## Parameters

### Item Description

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

## Return Values

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

## Related reference

[readobjects Subroutine](#)

[text2inst, name2inst, next2inst, or nextot2inst Subroutine](#)

[oid\\_cmp, oid\\_cpy, oid\\_free, sprintoid, str2oid, ode2oid, oid2ode, oid2ode\\_aux, prim2oid, or oid2prim Subroutine](#)

## Sockets

---

The operating system includes the Berkeley Software Distribution (BSD) interprocess communication (IPC) facility known as sockets. Sockets are communication channels that enable unrelated processes to exchange data locally and across networks. A single socket is one end point of a two-way communication channel. Socket subroutines enable interprocess and network interprocess communications (IPC).

AIX® runtime services beginning with the character `_`.

## `_getlong` Subroutine

### Purpose

Retrieves long byte quantities.

### Library

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

### Syntax

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

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

### Description

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

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

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

### Parameters

Item	Description
<i>MessagePtr</i>	Specifies a pointer into the byte stream.

## Return Values

The `_getlong` subroutine returns an unsigned long (32-bit) value.

## Files

Item	Description
<a href="#"><u>/etc/resolv.conf</u></a>	Lists name server and domain names.

## Related information

[Sockets Overview](#)

[Understanding Domain Name Resolution](#)

## `_getshort` Subroutine

### Purpose

Retrieves short byte quantities.

### Library

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

### Syntax

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

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

### Description

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

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

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

### Parameters

Item	Description
<i>MessagePtr</i>	Specifies a pointer into the byte stream.

## Return Values

The `_getshort` subroutine returns an unsigned short (16-bit) value.

## Files

Item	Description
------	-------------

<a href="#"><u>/etc/resolv.conf</u></a>	Defines name server and domain names.
---	---------------------------------------

## Related information

[Sockets Overview](#)

[Understanding Domain Name Resolution](#)

## **\_putlong Subroutine**

### Purpose

Places long byte quantities into the byte stream.

### Library

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

### Syntax

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

```
void _putlong ( Long, MessagePtr)
unsigned long Long;
u_char *MessagePtr;
```

### Description

The **\_putlong** subroutine places long byte quantities into the byte stream or arbitrary byte boundaries.

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

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

### Parameters

Item	Description
------	-------------

<i>Long</i>	Represents a 32-bit integer.
-------------	------------------------------

<i>MessagePtr</i>	Represents a pointer into the byte stream.
-------------------	--

## Files

Item	Description
------	-------------

<a href="#"><u>/etc/ resolv.conf</u></a>	Lists the name server and domain name.
--	--

## Related information

[Sockets Overview](#)

## **\_putshort Subroutine**

### **Purpose**

Places short byte quantities into the byte stream.

### **Library**

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

### **Syntax**

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

```
void _putshort ( Short, MessagePtr)
unsigned short Short;
u_char *MessagePtr;
```

### **Description**

The **\_putshort** subroutine puts short byte quantities into the byte stream or arbitrary byte boundaries.

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

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

### **Parameters**

<b>Item</b>	<b>Description</b>
<i>Short</i>	Represents a 16-bit integer.
<i>MessagePtr</i>	Represents a pointer into the byte stream.

### **Files**

<b>Item</b>	<b>Description</b>
<u><a href="#">/etc/resolv.conf</a></u>	Lists the name server and domain name.

### **Related information**

[Sockets Overview](#)

[Understanding Domain Name Resolution](#)

## **a**

AIX runtime services beginning with the letter *a*.

## **accept Subroutine**

## Purpose

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

## Library

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

## Syntax

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

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

## Description

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

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

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

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

The **accept** subroutine is used with **SOCK\_STREAM** and **SOCK\_CONN\_DGRAM** socket types.

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

The socket applications can be compiled with **COMPAT\_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

## Parameters

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

## Return Values

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

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

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

## Error Codes

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

Item	Description
<b>EBADF</b>	The <i>Socket</i> parameter is not valid.
<b>ECONNRESET</b>	The connection has been reset by the partner.
<b>EINTR</b>	The <b>accept</b> function was interrupted by a signal that was caught before a valid connection arrived.
<b>EINVAL</b>	The socket referenced by <i>s</i> is not currently a listen socket or has been shutdown with <b>shutdown</b> . A <b>listen</b> must be done before an <b>accept</b> is allowed.
<b>EMFILE</b>	The system limit for open file descriptors per process has already been reached ( <b>OPEN_MAX</b> ).
<b>ENFILE</b>	The maximum number of files allowed are currently open.
<b>ENOTSOCK</b>	The <i>Socket</i> parameter refers to a file, not a socket.
<b>EOPNOTSUPP</b>	The referenced socket is not of type <b>SOCK_STREAM</b> .
<b>EFAULT</b>	The <i>Address</i> parameter is not in a writable part of the user address space.
<b>EWOULDBLOCK</b>	The socket is marked as nonblocking, and no connections are present to be accepted.
<b>ENETDOWN</b>	The network with which the socket is associated is down.
<b>ENOTCONN</b>	The socket is not in the connected state.
<b>ECONNABORTED</b>	The client aborted the connection.

## Examples

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

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

### Related reference

[listen Subroutine](#)

[socket Subroutine](#)

### Related information

[Accepting Stream Connections Example Program](#)

[Binding Names to Sockets](#)

## arpresolve\_common Subroutine

## Purpose

Reads or creates new arp entries so that hardware addresses can be resolved.

## Syntax

```
int arpresolve_common ( ac, m, arpwhoas, dst, hwaddr, szhwaddr, extra, if_dependent)

    register struct arpcom *ac;
    struct mbuf *m;
    int (*arpwhoas)(register struct arpcom *ac,
                   struct in_addr *addr, int skipbestif, void *extra),
    struct sockaddr_in *dst;
    u_char * hwaddr;
    int szhwaddr;
    void *extra;
    union if_dependent *if_dependent;
```

## Description

The **arpresolve\_common** subroutine reads or creates new arp entries so that hardware addresses can be resolved. It is called by **arpresolve** from the IF layer of the interface. If the arp entry is complete, then **arpresolve\_common** returns the address pointed to by *hwaddr* and the data pointed to by *if\_dependent* if *if\_dependent* is true. If the arp entry is not complete, then this subroutine adds the memory buffer pointed to by *mbuf* to **at\_hold**. **at\_hold** holds one or more packets that are waiting for the arp entry to complete so they can be transmitted.

If an arp entry does not exist, **arpresolve\_common** creates a new entry by calling **arptnew** and then adds the memory buffer pointed to by *mbuf* to **at\_hold**. This subroutine calls **arpwhoas** when it creates a new arp entry or when the timer for the incomplete arp entry (with the IP address that is pointed to by *dst*) has expired.

## Parameters

Item	Description
<i>ac</i>	Points to the <b>arpcom</b> structure.
<i>m</i>	Points to the memory buffer ( <b>mbuf</b> ), which will be added to the list awaiting completion of the arp table entry.
<i>arpwhoas</i>	Points to the <b>arpwhoas</b> subroutine.
<i>addr</i>	Points to the <b>in_addr</b> structure's address.
<i>extra</i>	A void pointer that can be used in the future so that IF layers can pass extra structures to <b>arpwhoas</b> .
<i>dst</i>	Points to the <b>sockaddr_in</b> structure. This structure has the destination IP address.
<i>hwaddr</i>	Points to the buffer. This buffer contains the hardware address if it finds a completed entry.
<i>szhwaddr</i>	Size of the buffer pointed to by <i>hwaddr</i> .
<i>if_dependent</i>	Pointer to the <b>if_dependent</b> structure. <b>arpresolve_common</b> uses this to pass the <b>if_dependent</b> data, which is part of the <b>arptab</b> entry, to the calling function.

## Return Values

Item	Description
<b>ARP_MBUF</b>	The arp entry is not complete.

Item	Description
<b>ARP_HWADDR</b>	The <i>hwaddr</i> buffer is filled with the hardware address.
<b>ARP_FLG_NOARP</b>	The arp entry does not exist, and the <b>IFF_NOARP</b> flag is set only if the value of <i>if_type</i> is <b>IFT_ETHER</b> .

## arpupdate Subroutine

### Purpose

Updates arp entries for a given IP address.

### Syntax

```
int arpupdate (ac, m, hp, action, prm)
    register struct arpcom *ac;
    struct mbuf *m;
    caddr_t hp;
    int action;
    struct arpupdate_parm *prm;
```

### Description

The **arpupdate** subroutine updates arp entries for a given IP address. It is called by **arpinput** from the IF layer of the interface. This subroutine searches the arp table for an entry that matches the IP address. It then updates the arp entry for the given IP address. The **arpupdate** subroutine also performs reverse arp lookups.

The **arpupdate** subroutine enters a new address in **arptab**, pushing out the oldest entry from the bucket if there is no room. This subroutine always succeeds because no bucket can be completely filled with permanent entries (except when **arpioctl** tests whether another permanent entry can fit).

Depending on the action specified, the *prm* IP addresses **isaddr**, **itaddr**, and **myaddr** are used by the **arpupdate** subroutine.

### Parameters

Item	Description
<i>ac</i>	Points to the <b>arpcom</b> structure.
<i>m</i>	Points to the memory buffer ( <b>mbuf</b> ), that contains the arp response packet received by the interface.
<i>hp</i>	Points to the buffer that is passed by the interrupt handler.

<b>Item</b>	<b>Description</b>
<i>action</i>	<p>Returns a value that indicates which action is taken:</p> <p><b>LOOK</b> Looks for the <b>isaddr</b> IP address in the arp table and returns the hardware address and <b>if_dependent</b> structure.</p> <p><b>LKPUB</b> Looks for the <b>isaddr</b> IP address in the arp table and returns the hardware address and <b>if_dependent</b> structure only if the <b>ATF_PUBL</b> is set.</p> <p><b>UPDT</b> Updates the arp entry for an IP address (<b>isaddr</b>). If no arp entry is there, creates a new one and updates the <b>if_dependent</b> structure using the <b>ptr</b> function passed in the <b>prm</b> structure.</p> <p><b>REVARP</b> Reverses the arp request. <i>hwaddr</i> contains the hardware address, <i>szhwaddr</i> indicates its size, and <i>saddr</i> returns the IP address if an entry is found.</p>
<i>prm</i>	<p>Points to the <b>arpudpate_parm</b> structure. The values are:</p> <p><b>LOOK or LKPUB</b> <b>itaddr</b> and <b>myaddr</b> are ignored. <b>isaddr</b> is used for arp table lookup.</p> <p><b>UPDTE</b> <b>isaddr</b> points to the sender protocol address. <b>itaddr</b> points to the target protocol address. <b>myaddr</b> points to the protocol address of the interface that received the packet.</p>

## Return Values

<b>Item</b>	<b>Description</b>
<b>ARP_OK</b>	Lookup or update was successful.
<b>ARP_FAIL</b>	Lookup or update failed.
<b>ARP_NEWF</b>	New arp entry could not be created.

## b

AIX runtime services beginning with the letter *b*.

## bind Subroutine

### Purpose

Binds a name to a socket.

### Library

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

### Syntax

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

```
int bind ( Socket, Name, NameLength)
int Socket;
```

```
const struct sockaddr *Name;  
socklen_t NameLength;
```

## Description

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

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

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

The socket applications can be compiled with **COMPAT\_43** defined. This makes the **sockaddr** structure BSD 4.3 compatible. For more details refer to the **socket.h** file.

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.

**Note:** When you enable IPv6 for an application, IPv4 addresses are also supported. You can use an AF\_INET6 socket to send and receive both IPv4 and IPv6 packets because AF\_INET6 sockets are capable of handling communication with both IPv4 and IPv6 hosts. However, you must convert the address format of the IPv4 addresses that were previously passed to the socket calls to the IPv4-mapped IPv6 address format. For example, you must convert 10.1.1.1 in the *sockaddr\_in* structure to ::ffff:10.1.1.1 in the *sockaddr\_in6* structure.

## Parameters

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

## Return Values

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

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

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

## Error Codes

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

Value	Description
<b>EACCES</b>	The requested address is protected, and the current user does not have permission to access it.
<b>EADDRINUSE</b>	The specified address is already in use.
<b>EADDRNOTAVAIL</b>	The specified address is not available from the local machine.

Value	Description
<b>EAFNOSUPPORT</b>	The specified address is not a valid address for the address family of the specified socket.
<b>EAGAIN</b>	The transient ports are already in use and are not available.
<b>EBADF</b>	The <i>Socket</i> parameter is not valid.
<b>EDESTADDRREQ</b>	The <i>address</i> argument is a null pointer.
<b>EFAULT</b>	The <i>Address</i> parameter is not in a writable part of the <i>UserAddress</i> space.
<b>EINVAL</b>	The socket is already bound to an address.
<b>ENOBUF</b>	Insufficient buffer space available.
<b>ENODEV</b>	The specified device does not exist.
<b>ENOTSOCK</b>	The <i>Socket</i> parameter refers to a file, not a socket.
<b>EOPNOTSUPP</b>	The socket referenced by <i>Socket</i> parameter does not support address binding.

## Examples

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

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

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

### Related reference

[connect Subroutine](#)

[socket Subroutine](#)

## bind2addrsel Subroutine

### Purpose

Binds a socket to an address according to address selection preferences.

### Library

Library (**libc.a**)

### Syntax

```
#include <netinet/in.h>
int bind2addrsel(int socket, const struct sockaddr *dstaddr, socklen_t dstaddrlen)
```

## Description

When establishing a communication with a distant address, AIX uses a address selection algorithm to define what local address will be used to communicate with a distant address. This algorithm uses a set of ordered rules (RFC 3484) to choose this local address. Some of these rules use the type of address for this selection. By default, public addresses are preferred over temporary addresses; CGA addresses are preferred over non CGA addresses; home addresses are preferred over care-of addresses. An application may prefer the use other preference choices (for example use a temporary address rather that a public address) for the rules using the type of address. If these rules are applied, these preferences will be used. The application can express these preferences using a **setsockopt** call with the IPV6\_ADDR\_PREFERENCES option and a combination of the following flags:

- IPV6\_PREFER\_SRC\_HOME: prefer addresses reachable from a Home source address
- IPV6\_PREFER\_SRC\_COA: prefer addresses reachable from a Care-of source address
- IPV6\_PREFER\_SRC\_TMP: prefer addresses reachable from a temporary address
- IPV6\_PREFER\_SRC\_PUBLIC: the prefer addresses reachable from a public source address
- IPV6\_PREFER\_SRC\_CGA: the prefer addresses reachable from a Cryptographically Generated Address (CGA) source address
- IPV6\_PREFER\_SRC\_NONCGA: the prefer addresses reachable from a non-CGA source address

The application will then call **bind2addrsel**. **bind2addrsel** binds a socket to a local address selected to communicate with the given destination address according to the address selection preferences.

## Parameters

Item	Description
<b>socket</b>	Specifies the unique socket name
<b>dstaddr</b>	Points to a sockaddr structure containing the destination address. The sin6_family field of this sockaddr structure must be set to AF_INET6.
<b>dstaddrlen</b>	Specifies the size of the sockaddr structure pointed by dstaddr.

## Return Values

Upon successful completion, the subroutine returns 0

If unsuccessful, the subroutine returns -1 and errno is set accordingly:

## C

AIX runtime services beginning with the letter c.

## connect Subroutine

### Purpose

Connects two sockets.

### Library

Standard C Library (libc.a)

### Syntax

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

int connect ( Socket, Name, NameLength )
```

```
int Socket;  
const struct sockaddr *Name;  
socklen_t NameLength;
```

## Description

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

If a **connect** subroutine is issued on an unbound socket or a partially bound socket, the system automatically binds the socket. A partially bound socket is a socket that is assigned a port number but no IP address. The **connect** subroutine can be used to connect a socket to itself. A socket can be connected to itself, by binding a socket to a local port (by using **bind** subroutine) and then connecting it to the same port with a local IP address (by using **connect** subroutine).

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

- If the initiating socket is `SOCK_DGRAM`, the **connect** subroutine establishes the peer address. The peer address identifies the socket where all datagrams are sent on subsequent **send** subroutines. No connections are made by this **connect** subroutine. If the UDP socket is receiving datagrams when the **connect** subroutine is called, the subroutine changes the IP address, preventing the socket from receiving datagram packets based on the previous address.
- If the initiating socket is `SOCK_STREAM` or `SOCK_CONN_DGRAM`, the **connect** subroutine attempts to make a connection to the socket specified by the *Name* parameter. Each communication space interprets the *Name* parameter differently. For `SOCK_CONN_DGRAM` socket type and ATM protocol, the remote station might modify some of the ATM parameters. If the remote station modifies the ATM parameters, applications might query new values of ATM parameters by using the appropriate socket options.
- For a UNIX domain socket, a **connect** call succeeds only if the process that calls **connect** subroutine has read and write permissions on the socket file that is created by the **bind** call. Permissions are determined by the `umask` value, which is the value of the process that created the file.

## Parameters

Item	Description
<i>Socket</i>	Specifies the unique name of the socket.
<i>Name</i>	Specifies the address of the target socket that forms the other end of the communication line
<i>NameLength</i>	Specifies the length of the address structure.

## Return Values

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

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

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

## Error Codes

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

Value	Description
<b>EADDRINUSE</b>	The specified address is already in use. This error also occurs if the <code>SO_REUSEADDR</code> socket option was set and the local address (whether specified or selected by the system) is already in use.
<b>EADDRNOTAVAIL</b>	The specified address is not available from the local machine.
<b>EAFNOSUPPORT</b>	The addresses in the specified address family cannot be used with this socket.
<b>EALREADY</b>	The socket is specified with the <b>O_NONBLOCK</b> or <b>O_NDELAY</b> flag, and a previous connection attempt is not yet completed.
<b>EINTR</b>	The attempt to establish a connection was interrupted by delivery of a signal that was caught. The connection is established asynchronously.
<b>EACCES</b>	Search permission is denied on a component of the path prefix or write access to the named socket is denied.
<b>ENOBUFS</b>	The system ran out of memory for an internal data structure.
<b>EOPNOTSUPP</b>	The socket that is referenced by <i>Socket</i> parameter does not support <b>connect</b> .
<b>EWouldBLOCK</b>	The range that is allocated for TCP/UDP ephemeral ports is exhausted.
<b>EBADF</b>	The <i>Socket</i> parameter is not valid.
<b>ECONNREFUSED</b>	The attempt to connect was rejected.
<b>EFAULT</b>	The <i>Address</i> parameter is not in a writable part of the user address space.
<b>EINPROGRESS</b>	The socket is marked as nonblocking. The connection cannot be immediately completed. The application program can select the socket for writing during the connection process.
<b>EINVAL</b>	This error occurs in the following cases: <ul style="list-style-type: none"> <li>• The specified path name contains a character with the high-order bit set.</li> <li>• If the previous call to the <b>connect</b> subroutine fails. In such case, close and re-open the socket that is in error state and reattempt the call to the <b>connect</b> subroutine.</li> </ul>
<b>EISCONN</b>	The socket is already connected.
<b>ENETDOWN</b>	The specified physical network is down.
<b>ENETUNREACH</b>	No route to the network or host is present.
<b>ENOSPC</b>	There is no space that is left on a device or system table.
<b>ENOTCONN</b>	The socket might not be connected.
<b>ENOTSOCK</b>	The <i>Socket</i> parameter refers to a file, not a socket.
<b>ETIMEDOUT</b>	The establishment of a connection timed out before a connection was made.
<b>EPROTOTYPE</b>	The specified address has a different type from the socket that is bound to the specified peer address.
<b>ELOOP</b>	Too many symbolic links were encountered in converting the path name in the address.
<b>ENOENT</b>	A component of the path name does not name an existing file or the path name is an empty string.
<b>ENOTDIR</b>	A component of the path prefix of the path name in the address is not a directory.

## Examples

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

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

### Related reference

[bind Subroutine](#)

[/etc/socks5c.conf File](#)

## CreateIoCompletionPort Subroutine

### Purpose

Creates an I/O completion port with no associated file descriptor or associates an opened socket or file with an existing or newly created I/O completion port.

### Syntax

```
#include <iocp.h>
int CreateIoCompletionPort (FileDescriptor, CompletionPort, CompletionKey, ConcurrentThreads)
HANDLE FileDescriptor, CompletionPort;
DWORD CompletionKey, ConcurrentThreads;
```

### Description

The **CreateIoCompletionPort** subroutine creates an I/O completion port or associates an open file descriptor with an existing or newly created I/O completion port. When creating a new I/O completion port, the *CompletionPort* parameter is set to NULL, the *FileDescriptor* parameter is set to INVALID\_HANDLE\_VALUE (-1), and the *CompletionKey* parameter is ignored.

The **CreateIoCompletionPort** subroutine returns a descriptor (an integer) to the I/O completion port created or modified.

The **CreateIoCompletionPort** subroutine is part of the I/O Completion Port (IOCP) kernel extension.

**Note:** This subroutine only works with file descriptors of sockets, or regular files for use with the Asynchronous I/O (AIO) subsystem. It does not work with file descriptors of other types.

### Parameters

Item	Description
<i>FileDescriptor</i>	Specifies a valid file descriptor obtained from a call to the <b>socket</b> or <b>accept</b> subroutines.
<i>CompletionPort</i>	Specifies a valid I/O completion port descriptor. Specifying a <i>CompletionPort</i> parameter value of NULL causes the <b>CreateIoCompletionPort</b> subroutine to create a new I/O completion port.
<i>CompletionKey</i>	Specifies an integer to serve as the identifier for completion packets generated from a particular file-completion port set.
<i>ConcurrentThreads</i>	This parameter is not implemented and is present only for compatibility purposes.

## Return Values

Upon successful completion, the **CreateIoCompletionPort** subroutine returns an integer (the I/O completion port descriptor).

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

- Returns a value of NULL to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable, see the link in the Related Information section of this document.

## Error Codes

The **CreateIoCompletionPort** subroutine is unsuccessful if either of the following errors occur:

Item	Description
<b>EBADF</b>	The I/O completion port descriptor is invalid.
<b>EINVAL</b>	The file descriptor is invalid.
<b>EALREADY</b>	The file descriptor is already associated.

## Examples

The following program fragment illustrates the use of the **CreateIoCompletionPort** subroutine to create a new I/O completion port with no associated file descriptor:

```
c = CreateIoCompletionPort (INVALID_HANDLE_VALUE, NULL, 0, 0);
```

The following program fragment illustrates the use of the **CreateIoCompletionPort** subroutine to associate file descriptor 34 (which has a newly created I/O completion port) with completion key 25:

```
c = CreateIoCompletionPort (34, NULL, 25, 0);
```

The following program fragment illustrates the use of the **CreateIoCompletionPort** subroutine to associate file descriptor 54 (which has an existing I/O completion port) with completion key 15:

```
c = CreateIoCompletionPort (54, 12, 15, 0);
```

## Related information

[Error Notification Object Class](#)

## d

AIX runtime services beginning with the letter *d*.

## dn\_comp Subroutine

### Purpose

Compresses a domain name.

### Library

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

### Syntax

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

```
#include <arpa/nameser.h>
#include <resolv.h>
```

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

## Description

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

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

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

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

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

## Parameters

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

## Return Values

Upon successful completion, the **dn\_comp** subroutine returns the size of the compressed domain name. If unsuccessful, the **dn\_comp** subroutine returns a value of -1 to the calling program.

## Files

Item	Description
<b>/usr/include/resolv.h</b>	Contains global information used by the resolver subroutines.

## Related reference

[dn\\_expand Subroutine](#)

## Related information

[TCP/IP name resolution](#)

[Sockets Overview](#)

[Understanding Domain Name Resolution](#)

# dn\_expand Subroutine

## Purpose

Expands a compressed domain name.

## Library

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

## Syntax

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

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

## Description

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

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

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

## Parameters

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

## Return Values

Upon successful completion, the **dn\_expand** subroutine returns the size of the expanded domain name. If unsuccessful, the **dn\_expand** subroutine returns a value of -1 to the calling program.

## Files

Item	Description
<a href="#">/etc/resolv.conf</a>	Defines name server and domain name constants, structures, and values.

## Related reference

[dn\\_comp Subroutine](#)

## Related information

[exit subroutine](#)

[TCP/IP name resolution](#)

## e

AIX runtime services beginning with the letter e.

## ebind Subroutine

### Purpose

Binds a name to a socket. Also binds a socket to the specific Sensitivity Level (SL) that is passed as a parameter.

### Library

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

### Syntax

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

```
int ebind ( Socket, Name, NameLength, Label)
int Socket;
const struct sockaddr *Name;
socklen_t NameLength;
sec_labels_t *Label;
```

### Description

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

When a NULL pointer is passed to the *Label* parameter, then a normal multi-level port is created. However, when a valid label is passed to the *Label* parameter, a port at the specified Sensitivity Level (SL) is created. This means that only those incoming connections at the specified SL are able to connect. This also means that multiple sockets can be bound to the same port at different SLs. It is possible to create a multi-level port as well as several specific-level ports. If none of the specific SLs matches the incoming packet, then the packet port is a default multi-level port.

## Parameters

Item	Description
<i>Socket</i>	Specifies the socket descriptor of the socket to be bound. The socket descriptor is an integer,
<i>Name</i>	Points to an address structure that specifies the address to which the socket should be bound. The <b>/usr/include/sys/socket.h</b> file defines the <b>sockaddr</b> address structure. The <b>sockaddr</b> structure contains an identifier specific to the address format and protocol provided in the <b>socket</b> subroutine.
<i>NameLength</i>	Specifies the length of the socket address structure.
<i>Label</i>	Specifies the Sensitivity Label associated with the socket.

## Return Values

Item	Description
Successful	0
Unsuccessful	-1

## Error Codes

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

Value	Description
<b>EACCES</b>	The requested address is protected, and the current user does not have permission to access it.
<b>EADDRINUSE</b>	The specified address is already in use.
<b>EADDRNOTAVAIL</b>	The specified address is not available from the local machine.
<b>EAFNOSUPPORT</b>	The specified address is not a valid address for the address family of the specified socket.
<b>EBADF</b>	The <i>Socket</i> parameter is not valid.
<b>EDESTADDRREQ</b>	The <i>address</i> argument is a null pointer.
<b>EFAULT</b>	The <i>Address</i> parameter is not in a writable part of the user address space.
<b>EINVAL</b>	The socket is already bound to an address.
<b>ENOBUF</b>	Insufficient buffer space available.
<b>ENODEV</b>	The specified device does not exist.
<b>ENOTSOCK</b>	The <i>Socket</i> parameter refers to a file, not a socket.
<b>EOPNOTSUPP</b>	The socket referenced by the <i>Socket</i> parameter does not support address binding.

## endhostent Subroutine

### Purpose

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

## Library

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

## Syntax

```
#include <netdb.h>  
endhostent ()
```

## Description

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

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

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

## Files

Item	Description
<a href="#"><u>/etc/hosts</u></a>	Contains the host name database.
<a href="#"><u>/etc/netsvc.conf</u></a>	Contains the name service ordering.
<a href="#"><u>/usr/include/netdb.h</u></a>	Contains the network database structure.

### Related reference

[sethostent Subroutine](#)

### Related information

[exit subroutine](#)

[Sockets Overview](#)

## endhostent\_r Subroutine

### Purpose

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

## Library

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

## Syntax

```
#include <netdb.h>  
  
void endhostent_r (struct hostent_data *ht_data);
```

## Description

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

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

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

## Parameters

Item	Description
<i>ht_data</i>	Points to the <b>hostent_data</b> structure

## Files

Item	Description
<b>/etc/hosts</b>	Contains the host name database.
<b>/etc/netsvc.conf</b>	Contains the name service ordering.
<b>/usr/include/netdb.h</b>	Contains the network database structure.

## endnetent Subroutine

### Purpose

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

### Library

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

### Syntax

```
#include <netdb.h>
```

```
void endnetent ( )
```

## Description

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

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

## Return Values

If a previous **setnetent** subroutine has been performed and the *StayOpen* parameter does not equal 0, then the **endnetent** subroutine will not close the **/etc/networks** file. Also, the **setnetent** subroutine does not indicate that it closed the file. A second **setnetent** subroutine has to be issued with the *StayOpen*

parameter equal to 0 in order for a following **endnetent** subroutine to succeed. If this is not done, the **/etc/networks** file must be closed with the **exit** subroutine.

## Examples

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

```
endnetent();
```

## Files

Item	Description
------	-------------

<a href="#">/etc/networks</a>	Contains official network names.
-------------------------------	----------------------------------

### Related reference

[setnetent Subroutine](#)

### Related information

[exit subroutine](#)

## endnetent\_r Subroutine

### Purpose

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

### Library

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

### Syntax

```
#include <netdb.h>

void endnetent_r (net_data)
struct netent_data *net_data;
```

### Description

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

### Parameters

Item	Description
------	-------------

<i>net_data</i>	Points to the <b>netent_data</b> structure.
-----------------	---

## Files

Item	Description
------	-------------

<a href="#">/etc/networks</a>	Contains official network names.
-------------------------------	----------------------------------

## endnetgrent\_r Subroutine

### Purpose

Handles the group network entries.

### Library

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

### Syntax

```
#include <netdb.h>
void endnetgrent_r (void **ptr)
```

### Description

The **setnetgrent\_r** subroutine establishes the network group from which the **getnetgrent\_r** subroutine will obtain members, and also restarts calls to the **getnetgrent\_r** subroutine from the beginning of the list. If the previous **setnetgrent\_r** call was to a different network group, an **endnetgrent\_r** call is implied.

The **endnetgrent\_r** subroutine frees the space allocated during the **getnetgrent\_r** calls.

### Parameters

Item	Description
<i>ptr</i>	Keeps the function threadsafe.

### Files

Item	Description
<i>/etc/netgroup</i>	Contains network groups recognized by the system.
<i>/usr/include/netdb.h</i>	Contains the network database structures.

## endprotoent Subroutine

### Purpose

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

### Library

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

### Syntax

```
void endprotoent (void)
```

### Description

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

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

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

## Return Values

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

## Examples

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

```
endprotoent();
```

## Files

Item	Description
<b>/etc/protocols</b>	Contains protocol names.

### Related reference

[setprotoent Subroutine](#)  
[getprotobyname Subroutine](#)  
[getprotobynumber Subroutine](#)  
[getprotoent Subroutine](#)  
[getservbyport Subroutine](#)  
[getservent Subroutine](#)  
[setservent Subroutine](#)

### Related information

[exit subroutine](#)  
[Sockets Overview](#)  
[Understanding Network Address Translation](#)

## endprotoent\_r Subroutine

### Purpose

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

### Library

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

### Syntax

```
void endprotoent_r(proto_data);  
struct protoent_data *proto_data;
```

## Description

The **endprotoent\_r** subroutine closes the **/etc/protocols** file, which is opened by the calls made to the **getprotoent\_r** subroutine, **getprotobyname\_r** subroutine, or **getprotobynumber\_r** subroutine.

## Parameters

Item	Description
<i>proto_data</i>	Points to the <b>protoent_data</b> structure

## Files

Item	Description
<b>/etc/protocols</b>	Contains protocol names.

## endservent Subroutine

### Purpose

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

### Library

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

### Syntax

```
#include <netdb.h>
```

```
void endservent ( )
```

### Description

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

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

### Return Values

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

### Examples

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

```
endservent ( );
```

## Files

Item	Description
------	-------------

<a href="#">/etc/services</a>	Contains service names.
-------------------------------	-------------------------

### Related reference

[getservbyname Subroutine](#)

[getservbyport Subroutine](#)

### Related information

[exit subroutine](#)

[Sockets Overview](#)

## endservent\_r Subroutine

### Purpose

Closes the [/etc/services](#) file.

### Library

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

### Syntax

```
#include <netdb.h>
void endservent_r(serv_data)
struct servent_data *serv_data;
```

### Description

The **endservent\_r** subroutine closes the [/etc/services](#) file, which is opened by a call made to the **getservent\_r** subroutine, **getservbyname\_r** subroutine, or **getservbyport\_r** subroutine opens the [/etc/services](#) file.

### Parameters

Item	Description
------	-------------

<i>serv_data</i>	Points to the <b>servent_data</b> structure
------------------	---

### Examples

To close the [/etc/services](#) file, type:

```
endservent_r(serv_data);
```

## Files

Item	Description
------	-------------

<a href="#">/etc/services</a>	Contains service names.
-------------------------------	-------------------------

## ether\_ntoa, ether\_aton, ether\_ntohost, ether\_hostton, or ether\_line Subroutine

### Purpose

Maps 48-bit Ethernet numbers.

### Library

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

### Syntax

```
#include <arpa/inet.h>
```

```
char *ether_ntoa (EthernetNumber)
struct ether_addr * EthernetNumber;
```

```
struct ether_addr *ether_aton( String);
char *string
```

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

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

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

### Description

**Attention:** Do not use the **ether\_ntoa** or **ether\_aton** subroutine in a multithreaded environment.

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

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

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

### Parameters

Item	Description
<i>EthernetNumber</i>	Points to an Ethernet number.

Item	Description
<i>String</i>	Points to an ASCII string.
<i>HostName</i>	Points to a host name.
<i>Line</i>	Points to a line.

## Return Values

Item	Description
<b>0</b>	Indicates that the subroutine was successful.
<b>non-zero</b>	Indicates that the subroutine was not successful.

## Files

Item	Description
<i>/etc/ethers</i>	Contains information about the known (48-bit) Ethernet addresses of hosts on the Internet.

## Related information

[Subroutines Overview](#)

[List of Multithread Subroutines](#)

## f

AIX runtime services beginning with the letter *f*.

## FrcaCacheCreate Subroutine

### Purpose

Creates a cache instance within the scope of a Fast Response Cache Accelerator (FRCA) instance.

### Library

FRCA Library (**libfrca.a**)

### Syntax

```
#include <frca.h>
int32_t FrcaCacheCreate ( CacheHandle, FrcaHandle, CacheSpec);
int32_t * CacheHandle;
int32_t FrcaHandle;
frca_cache_create_t * CacheSpec;
```

### Description

The **FrcaCacheCreate** subroutine creates a cache instance for an FRCA instance that has already been configured. Multiple caches can be created for an FRCA instance. Cache handles are unique only within the scope of the FRCA instance.

## Parameters

Item	Description
<i>CacheHandle</i>	Returns a handle that is required by the other cache-related subroutines of the FRCA API to refer to the newly created FRCA cache instance.
<i>FrcaHandle</i>	Identifies the FRCA instance for which the cache is created.
<i>CacheSpec</i>	Points to a <b>frca_ctrl_create_t</b> structure, which specifies the characteristics of the cache to be created. The structure contains the following members:  <b>uint32_t</b> <i>cacheType</i> ; <b>uint32_t</b> <i>nMaxEntries</i> ;  <b>Note:</b> Structure members do not necessarily appear in this order.  <b>cacheType</b> Specifies the type of the cache instance. This field must be set to <code>FCTRL_SERVERTYPE_HTTP</code> .  <b>nMaxEntries</b> Specifies the maximum number of entries allowed for the cache instance.

## Return Values

Item	Description
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable <i>errno</i> is set to indicate the specific type of error.

## Error Codes

Item	Description
<b>EINVAL</b>	The <i>CacheHandle</i> or the <i>CacheSpec</i> parameter is zero or the <i>CacheSpec</i> parameter is not of the correct type <b>FCTRL_CACHETYPE_HTTP</b> .
<b>EFAULT</b>	The <i>CacheHandle</i> or the <i>CacheSpec</i> point to an invalid address.
<b>ENOENT</b>	The <i>FrcaHandle</i> parameter is invalid.

## FrcaCacheDelete Subroutine

### Purpose

Deletes a cache instance within the scope of a Fast Response Cache Accelerator (FRCA) instance.

### Library

FRCA Library (**libfrca.a**)

### Syntax

```
#include <frca.h>  
int32_t FrcaCacheDelete ( CacheHandle, FrcaHandle );
```

```
int32_t CacheHandle;  
int32_t FrcaHandle;
```

## Description

The **FrcaCacheDelete** subroutine deletes a cache instance and releases any associated resources.

## Parameters

Item	Description
<i>CacheHandle</i>	Identifies the cache instance that is to be deleted.
<i>FrcaHandle</i>	Identifies the FRCA instance to which the cache instance belongs.

## Return Values

Item	Description
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable <i>errno</i> is set to indicate the specific type of error.

## Error Codes

Item	Description
<b>ENOENT</b>	The <i>CacheHandle</i> or the <i>FrcaHandle</i> parameter is invalid.

## FrcaCacheLoadFile Subroutine

### Purpose

Loads a file into a cache associated with a Fast Response Cache Accelerator (FRCA) instance.

### Library

FRCA Library (**libfrca.a**)

### Syntax

```
#include <frca.h>  
int32_t FrcaCacheLoadFile ( CacheHandle, FrcaHandle, FileSpec, AssocData);  
int32_t CacheHandle;  
int32_t FrcaHandle;  
frca_filespec_t * FileSpec;  
frca_assocdata_t * AssocData;
```

### Description

The **FrcaCacheLoadFile** subroutine loads a file into an existing cache instance for an previously configured FRCA instance.

## Parameters

Item	Description
<i>CacheHandle</i>	Identifies the cache instance to which the new entry should be added.
<i>FrcaHandle</i>	Identifies the FRCA instance to which the cache instance belongs.
<i>FileSpec</i>	<p>Points to a <b>frca_loadfile_t</b> structure, which specifies characteristics used to identify the cache entry that is to be loaded into the given cache. The structure contains the following members:</p> <pre><b>uint32_t</b> <i>cacheEntryType</i>; <b>char *</b> <i>fileName</i>; <b>char *</b> <i>virtualHost</i>; <b>char *</b> <i>searchKey</i>;</pre> <p><b>Note:</b> Structure members do not necessarily appear in this order.</p> <p><b>cacheEntryType</b> Specifies the type of the cache entry. This field must be set to FCTRL_CET_HTTPFILE.</p> <p><b>fileName</b> Specifies the absolute path to the file that is providing the contents for the new cache entry.</p> <p><b>virtualHost</b> Specifies a virtual host name that is being served by the FRCA instance.</p> <p><b>searchKey</b> Specifies the key that the cache entry can be found under by the FRCA instance when it processes an intercepted request. For the HTTP GET engine, the search key is identical to the <i>abs_path</i> part of the HTTP URL according to section 3.2.2 of RFC 2616. For example, the search key corresponding to the URL <code>http://www.mydomain/welcome.html</code> is <code>/welcome.html</code>.</p> <p><b>Note:</b> If a cache entry with the same type, file name, virtual host, and search key already exists and the file has not been modified since the existing entry was created, the load request succeeds without any effect. If the entry exists and the file's contents have been modified since being loaded into the cache, the cache entry is updated. If the entry exists and the file's contents have not changed, but any of the settings of the HTTP header fields change, the existing entry must be unloaded first.</p>

<b>Item</b>	<b>Description</b>
<i>AssocData</i>	<p>Points to a <b>frca_assocdata_t</b> structure, which specifies additional information to be associated with the contents of the given cache entry. The structure contains the following members:</p> <pre> <b>uint32_t</b> <i>assocDataType</i>; <b>char *</b> <i>cacheControl</i>; <b>char *</b> <i>contentType</i>; <b>char *</b> <i>contentEncoding</i>; <b>char *</b> <i>contentLanguage</i>; <b>char *</b> <i>contentCharset</i>; </pre> <p><b>Note:</b> Structure members do not necessarily appear in this order.</p> <p><b><i>assocDataType</i></b> Specifies the type of data that is associated with the given cache entry.</p> <p><b><i>cacheControl</i></b> Specifies the settings of the corresponding HTTP header field according to RFC 2616.</p> <p><b><i>contentType</i></b> Specifies the settings of the corresponding HTTP header field according to RFC 2616.</p> <p><b><i>contentEncoding</i></b> Specifies the settings of the corresponding HTTP header field according to RFC 2616.</p> <p><b><i>contentLanguage</i></b> Specifies the settings of the corresponding HTTP header field according to RFC 2616.</p> <p><b><i>contentCharset</i></b> Specifies the settings of the corresponding HTTP header field according to RFC 2616.</p>

## Return Values

<b>Item</b>	<b>Description</b>
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable <i>errno</i> is set to indicate the specific type of error.

## Error Codes

<b>Item</b>	<b>Description</b>
<b>EINVAL</b>	The <i>FileSpec</i> or the <i>AssocData</i> parameter is zero or are not of the correct type or any of the <i>fileName</i> or the <i>searchKey</i> components are zero or the size of the file is zero.
<b>EFAULT</b>	The <i>FileSpec</i> or the <i>AssocData</i> parameter or one of their components points to an invalid address.
<b>ENOMEM</b>	The FRCA or NBC subsystem is out of memory.
<b>EFBIG</b>	The content of the cache entry failed to load into the NBC. Check network options <b>nbc_limit</b> , <b>nbc_min_cache</b> , and <b>nbc_max_cache</b> .

Item	Description
<b>ENOTREADY</b>	The kernel extension is currently being loaded or unloaded.
<b>ENOENT</b>	The <i>CacheHandle</i> or the <i>FrcaHandle</i> parameter is invalid.

## FrcaCacheUnloadFile Subroutine

### Purpose

Removes a cache entry from a cache that is associated with a Fast Response Cache Accelerator (FRCA) instance.

### Library

FRCA Library (**libfrca.a**)

### Syntax

```
#include <frca.h>
int32_t FrcaCacheUnoadFile ( CacheHandle, FrcaHandle, FileSpec);
int32_t CacheHandle;
int32_t FrcaHandle;
frca_filespec_t * FileSpec;
```

### Description

The **FrcaCacheUnoadFile** subroutine removes a cache entry from an existing cache instance for an previously configured FRCA instance.

### Parameters

Item	Description
<i>CacheHandle</i>	Identifies the cache instance from which the entry should be removed.
<i>FrcaHandle</i>	Identifies the FRCA instance to which the cache instance belongs.

Item	Description
<i>FileSpec</i>	Points to a <b>frca_loadfile_t</b> structure, which specifies characteristics used to identify the cache entry that is to be removed from the given cache. The structure contains the following members:

```

uint32_t cacheEntryType;
char * fileName;
char * virtualHost;
char * searchKey;

```

**Note:** Structure members do not necessarily appear in this order.

***cacheEntryType***

Specifies the type of the cache entry. This field must be set to **FCTRL\_CET\_HTTPFILE**.

***fileName***

Specifies the absolute path to the file that is to be removed from the cache.

***virtualHost***

Specifies a virtual host name that is being served by the FRCA instance.

***searchKey***

Specifies the key under which the cache entry can be found.

**Note:** The **FrcaCacheUnoadFile** subroutine succeeds if a cache entry with the same type, file name, virtual host, and search key does not exist. This subroutine fails if the file associated with *fileName* does not exist or if the calling process does not have sufficient access permissions.

## Return Values

Item	Description
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable <i>errno</i> is set to indicate the specific type of error.

## Error Codes

Item	Description
<b>EINVAL</b>	The <i>FileSpec</i> parameter is zero or the <i>cacheEntryType</i> component is not set to <b>FCTRL_CET_HTTPFILE</b> or the <i>searchKey</i> component is zero or the <i>fileName</i> is '/' or the <i>fileName</i> is not an absolute path.
<b>EFAULT</b>	The <i>FileSpec</i> parameter or one of the components points to an invalid address.
<b>EACCES</b>	Access permission is denied on the <i>fileName</i> .

## FrcaCtrlCreate Subroutine

### Purpose

Creates a Fast Response Cache Accelerator (FRCA) control instance.

## Library

FRCA Library (**libfrca.a**)

## Syntax

```
#include <frca.h>
int32_t FrcaCtrlCreate ( FrcaHandle, InstanceSpec);
int32_t * FrcaHandle;
frca_ctrl_create_t * InstanceSpec;
```

## Description

The **FrcaCtrlCreate** subroutine creates and configures an FRCA instance that is associated with a previously configured TCP listen socket. TCP connections derived from the TCP listen socket are intercepted by the FRCA instance and, if applicable, adequate responses are generated by the in-kernel code on behalf of the user-level application.

The only FRCA instance type that is currently supported handles static GET requests as part of the Hypertext Transfer Protocol (HTTP).

## Parameters

Item	Description
<i>FrcaHandle</i>	Returns a handle that is required by the other FRCA API subroutines to refer to the newly configured FRCA instance.

Item	Description
<i>InstanceSpec</i>	Points to a <b>frca_ctrl_create_t</b> structure, which specifies the parameters used to configure the newly created FRCA instance. The structure contains the following members:

```

uint32_t serverType;
char * serverName;
uint32_t nListenSockets;
uint32_t * ListenSockets;
uint32_t flags;
uint32_t nMaxConnections;
uint32_t nLogBufs;
char * LogFile;

```

**Note:** Structure members do not necessarily appear in this order.

***serverType***

Specifies the type for the FRCA instance. This field must be set to **FCTRL\_SERVERTYPE\_HTTP**.

***serverName***

Specifies the value to which the HTTP header field is set.

***nListenSocket***

Specifies the number of listen socket descriptors pointed to by *listenSockets*.

***listenSocket***

Specifies the TCP listen socket that the FRCA instance should be configured to intercept.

**Note:** The TCP listen socket must exist and the **SO\_KERNACCEPT** socket option must be set at the time of calling the **FrcaCtrlCreate** subroutine.

***flags***

Specifies the logging format, the initial state of the logging subsystem, and whether responses generated by the FRCA instance should include the **Server:** HTTP header field. The valid flags are as follows:

- FCTRL\_KEEPLIVE**
- FCTRL\_LOGFORMAT**
- FCTRL\_LOGFORMAT\_ECLF**
- FCTRL\_LOGFORMAT\_VHOST**
- FCTRL\_LOGMODE**
- FCTRL\_LOGMODE\_ON**
- FCTRL\_SENDSERVERHEADER**

***nMaxConnections***

Specifies the maximum number of intercepted connections that are allowed at any given point in time.

***nLogBufs***

Specifies the number of preallocated logging buffers used for logging information about HTTP GET requests that have been served successfully.

Item	Description
------	-------------

**logFile**

Specifies the absolute path to a file used for appending logging information. The HTTP GET engine uses *logFile* as a base name and appends a sequence number to it to generate the actual file name. Whenever the size of the current log file exceeds the threshold of approximately 1 gigabyte, the sequence number is incremented by 1 and the logging subsystem starts appending to the new log file.

**Note:** The FRCA instance creates the log file, but not the path to it. If the path does not exist or is not accessible, the FRCA instance reverts to the default log file **/tmp/frca.log**.

### Return Values

Item	Description
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable <i>errno</i> is set to indicate the specific type of error.

### Error Codes

Item	Description
<b>EINVAL</b>	The <i>FrcaHandle</i> or the <i>InstanceSpec</i> parameter is zero or is not of the correct type or the <i>listenSockets</i> components do not specify any socket descriptors.
<b>EFAULT</b>	The <i>FrcaHandle</i> or the <i>InstanceSpec</i> or a component of the <i>InstanceSpec</i> points to an invalid address.
<b>ENOTREADY</b>	The kernel extension is currently being loaded or unloaded.
<b>ENOTSOCK</b>	A TCP listen socket does not exist.

## FrcaCtrlDelete Subroutine

### Purpose

Deletes a Fast Response Cache Accelerator (FRCA) control instance.

### Library

FRCA Library (**libfrca.a**)

### Syntax

```
#include <frca.h>
int32_t FrcaCtrlDelete ( FrcaHandle);
int32_t * FrcaHandle;
```

### Description

The **FrcaCtrlDelete** subroutine deletes an FRCA instance and releases any associated resources.

The only FRCA instance type that is currently supported handles static GET requests as part of the Hypertext Transfer Protocol (HTTP).

## Parameters

Item	Description
<i>FrcaHandle</i>	Identifies the FRCA instance on which this operation is performed.

## Return Values

Item	Description
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable <i>errno</i> is set to indicate the specific type of error.

## Error Codes

Item	Description
<b>ENOENT</b>	The <i>FrcaHandle</i> parameter is invalid.
<b>ENOTREADY</b>	The FRCA control instance is in an undefined state.

## FrcaCtrlLog Subroutine

### Purpose

Modifies the behavior of the logging subsystem.

### Library

FRCA Library (**libfrca.a**)

### Syntax

```
#include <frca.h>
int32_t FrcaCtrlLog ( FrcaHandle, Flags);
int32_t FrcaHandle;
uint32_t Flags;
```

### Description

The **FrcaCtrlLog** subroutine modifies the behavior of the logging subsystem for the Fast Response Cache Accelerator (FRCA) instance specified. Modifiable attributes are the logging mode, which can be turned on or off, and the logging format, which defaults to the HTTP Common Log Format (CLF). The logging format can be changed to Extended Common Log Format (ECLF) and can be set to include virtual host information.

The only FRCA instance type that is currently supported handles static GET requests as part of the Hypertext Transfer Protocol (HTTP).

## Parameters

Item	Description
<i>FrcaHandle</i>	Returns a handle that is required by the other FRCA API subroutines to refer to the newly configured FRCA instance.
<i>Flags</i>	Specifies the behavior of the logging subsystem. The parameter value is constructed by logically ORing single flags. The valid flags are as follows: <b>FCTRL_LOGFORMAT</b> <b>FCTRL_LOGFORMAT_ECLF</b> <b>FCTRL_LOGFORMAT_VHOST</b> <b>FCTRL_LOGMODE</b> <b>FCTRL_LOGMODE_ON</b>

## Return Values

Item	Description
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable <i>errno</i> is set to indicate the specific type of error.

## Error Codes

Item	Description
<b>ENOTREAD</b> Y	The kernel extension is currently being loaded or unloaded.

## FrcaCtrlStart Subroutine

### Purpose

Starts the interception of TCP data connections for a previously configured Fast Response Cache Accelerator (FRCA) instance.

### Library

FRCA Library (**libfrca.a**)

### Syntax

```
#include <frca.h>  
int32_t FrcaCtrlStart ( FrcaHandle );  
int32_t * FrcaHandle;
```

### Description

The **FrcaCtrlStart** subroutine starts the interception of TCP data connections for an FRCA instance. If the FRCA instance cannot handle the data on that connection, it passes the data to the user-level application that has established the listen socket.

The only FRCA instance type that is currently supported handles static GET requests as part of the Hypertext Transfer Protocol (HTTP).

## Parameters

Item	Description
<i>FrcaHandle</i>	Identifies the FRCA instance on which this operation is performed.

## Return Values

Item	Description
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable <i>errno</i> is set to indicate the specific type of error.

## Error Codes

Item	Description
<b>ENOENT</b>	The <i>FrcaHandle</i> parameter is invalid.
<b>ENOTREADY</b>	The FRCA control instance is in an undefined state.
<b>ENOTSOCK</b>	A TCP listen socket that was passed in with the <i>FrcaCtrlCreate</i> cannot be intercepted because it does not exist.

## FrcaCtrlStop Subroutine

### Purpose

Stops the interception of TCP data connections for a Fast Response Cache Accelerator (FRCA) instance.

### Library

FRCA Library (**libfrca.a**)

### Syntax

```
#include <frca.h>
int32_t FrcaCtrlStop ( FrcaHandle);
int32_t * FrcaHandle;
```

### Description

The **FrcaCtrlStop** subroutine stops the interception of newly arriving TCP data connections for a previously configured FRCA instance. Connection requests are passed to the user-level application that has established the listen socket.

The only FRCA instance type that is currently supported handles static GET requests as part of the Hypertext Transfer Protocol (HTTP).

## Parameters

Item	Description
<i>FrcaHandle</i>	Identifies the FRCA instance on which this operation is performed.

## Return Values

Item	Description
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable <i>errno</i> is set to indicate the specific type of error.

## Error Codes

Item	Description
<b>ENOENT</b>	The <i>FrcaHandle</i> parameter is invalid.
<b>ENOTREADY</b>	The FRCA control instance has not been started yet.

## freeaddrinfo Subroutine

### Purpose

Frees memory allocated by the [“getaddrinfo Subroutine”](#) on page 110.

### Library

The Standard C Library (<**libc.a**>)

### Syntax

```
#include <sys/socket.h>
#include <netdb.h>
void freeaddrinfo (struct addrinfo *ai)
```

### Description

The **freeaddrinfo** subroutine frees one or more **addrinfo** structures returned by the **getaddrinfo** subroutine, along with any additional storage associated with those structures. If the **ai\_next** field of the structure is not NULL, the entire list of structures is freed.

### Parameters

Item	Description
<i>ai</i>	Points to dynamic storage allocated by the <b>getaddrinfo</b> subroutine

### Related information

[gai\\_strerror Subroutine](#)

## g

AIX runtime services beginning with the letter *g*.

## getaddrinfo Subroutine

### Purpose

Protocol-independent hostname-to-address translation.

## Library

Library (**libc.a**)

## Syntax

```
#include <sys/socket.h>
#include <netdb.h>
int getaddrinfo (hostname, servname, hints, res)
const char *hostname;
const char *servname;
const struct addrinfo *hints;
struct addrinfo **res;
```

## Description

The *hostname* and *servname* parameters describe the hostname and/or service name to be referenced. Zero or one of these arguments may be NULL. A non-NULL hostname may be either a hostname or a numeric host address string (a dotted-decimal for IPv4 or hex for IPv6). A non-NULL servname may be either a service name or a decimal port number.

The *hints* parameter specifies hints concerning the desired return information. The *hostname* and *servname* parameters are pointers to null-terminated strings or NULL. One or both of these arguments must be a non-NULL pointer. In a normal client scenario, both the *hostname* and *servname* parameters are specified. In the normal server scenario, only the *servname* parameter is specified. A non-NULL hostname string can be either a host name or a numeric host address string (for example, 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 *hints* parameter, to provide hints concerning the type of socket that the caller supports. In this **hints** structure, all members other than **ai\_flags**, **ai\_eflags**, **ai\_family**, **ai\_socktype**, and **ai\_protocol** must be zero or a NULL pointer. A value of PF\_UNSPEC for **ai\_family** means the caller will accept any protocol family. A value of zero for **ai\_socktype** means the caller accepts any socket type. A value of zero for **ai\_protocol** means the caller accepts any protocol. For example, if the caller handles only TCP and not UDP, the **ai\_socktype** member of the **hints** structure should be set to SOCK\_STREAM when the **getaddrinfo** subroutine is called. If the caller handles only IPv4 and not IPv6, the **ai\_family** member of the **hints** structure should be set to PF\_INET when **getaddrinfo** is called. If the *hints* parameter in **getaddrinfo** is a NULL pointer, it is the same as if the caller fills in an **addrinfo** structure initialized to zero with **ai\_family** set to PF\_UNSPEC.

Upon successful return, a pointer to a linked list of one or more **addrinfo** structures is returned through the *res* parameter. 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** subroutine. 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 the **bind** subroutine. If the *hostname* parameter 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, the returned socket address structure is ready for a call to the **connect** subroutine (for a connection-oriented protocol) or the **connect**, **sendto**, or **sendmsg** subroutine (for a connectionless protocol). If the *hostname* parameter is a NULL pointer, the IP address portion of the socket address structure is set to the loopback address.

If the AI\_CANONNAME bit is set in the **ai\_flags** member of the **hints** structure, upon successful return the **ai\_canonname** member of the first **addrinfo** structure in the linked list points to a NULL-terminated string containing the canonical name of the specified hostname.

If the AI\_NUMERICHOST flag is specified, a non-NULL nodename string supplied is a numeric host address string. Otherwise, an (EAI\_NONAME) error is returned. This flag prevents any type of name resolution service (such as, DNS) from being invoked.

If the AI\_NUMERICSERV flag is specified, a non-NULL servname string supplied is a numeric port string. Otherwise, an (EAI\_NONAME) error is returned. This flag prevents any type of name resolution service from being invoked.

If the AI\_V4MAPPED flag is specified along with an **ai\_family** value of AF\_INET6, the **getaddrinfo** subroutine returns IPv4-mapped IPv6 addresses when no matching IPv6 addresses (**ai\_addrlen** is 16) are found. For example, when using DNS, if no AAAA or A6 records are found, a query is made for A records. Any found are returned as IPv4-mapped IPv6 addresses. The AI\_V4MAPPED flag is ignored unless **ai\_family** equals AF\_INET6.

If the AI\_ALL flag is used with the AI\_V4MAPPED flag, the **getaddrinfo** subroutine returns all matching IPv6 and IPv4 addresses. For example, when using DNS, a query is first made for AAAA/A6 records. If successful, those IPv6 addresses are returned. Another query is made for A records, and any IPv4 addresses found are returned as IPv4-mapped IPv6 addresses. The AI\_ALL flag without the AI\_V4MAPPED flag is ignored.

**Note:** When **ai\_family** is not specified (AF\_UNSPEC), AI\_V4MAPPED and AI\_ALL flags are used if AF\_INET6 is supported.

If the AI\_EXTFLAGS is specified in the **ai\_flags** member of the hints structure and **ai\_eflags** is specified as a non zero value, the address selection algorithm is affected. The address selection algorithm orders the list of returned addrinfo structures using a set of ordered rules (RFC 3484) taking into account the address contained in the ai\_addr member of each addrinfo structure and the source addresses from which this address can be reached. The ai\_eflags expresses preferences meaning that the rules described below will be applied if a higher rule has not ordered the set of addresses before.

The **ai\_eflags** can be set to a combination of the following flags:

- IPV6\_PREFER\_SRC\_HOME: prefer addresses reachable from a Home source address
- IPV6\_PREFER\_SRC\_COA: prefer addresses reachable from a Care-of source address
- IPV6\_PREFER\_SRC\_TMP: prefer addresses reachable from a temporary address
- IPV6\_PREFER\_SRC\_PUBLIC: the prefer addresses reachable from a public source address
- IPV6\_PREFER\_SRC\_CGA: the prefer addresses reachable from a Cryptographically Generated Address (CGA) source address
- IPV6\_PREFER\_SRC\_NONCGA: the prefer addresses reachable from a non-CGA source address

For instance, the IPV6\_PREFER\_SRC\_TMP ai\_eflags means that the address selection algorithm will order the returned addrinfo structures with addresses reachable from a temporary address before the ones with addresses reachable from a public address whenever possible. Setting contradictory flags (e.g. IPV6\_PREFER\_SRC\_TMP and IPV6\_PREFER\_SRC\_PUBLIC) at the same time results in the error EINVAL.

If the AI\_ADDRCONFIG flag is specified, a query for AAAA or A6 records should occur only if the node has at least one IPv6 source address configured. A query for A records should occur only if the node has at least one IPv4 source address configured. The loopback address is not considered valid as a configured source address.

All of the information returned by the **getaddrinfo** subroutine is dynamically allocated: the **addrinfo** structures, the socket address structures, and canonical host name strings pointed to by the **addrinfo** structures. To return this information to the system, [freeaddrinfo](#) subroutine is called.

**The addrinfo structure is defined as:**

```
struct addrinfo {
    int          ai_flags;           /* AI_PASSIVE, AI_CANONNAME */
    int          ai_family;         /* PF_xxx */
    int          ai_socktype;       /* SOCK_xxx */
    int          ai_protocol;       /* 0 or IP=PROTO_xxx for IPv4 and IPv6 */
    size_t      ai_addrlen;        /* length of ai_addr */
    char        *ai_canonname;     /* canonical name for hostname */
    struct sockaddr *ai_addr;      /* binary address */
}
```

```

struct addrinfo  *ai_next;          /* next structure in linked list */
int ai_eflags; /* Extended flags for special usage */
}

```

## Return Values

If the query is successful, a pointer to a linked list of one or more **addrinfo** structures is returned via the *res* parameter. A zero return value indicates success. If the query fails, a non-zero error code is returned.

## Error Codes

The following names are the non-zero error codes. See *netdb.h* for further definition.

Item	Description
<b>EAI_ADDRFAMILY</b>	Address family for hostname not supported
<b>EAI_AGAIN</b>	Temporary failure in name resolution
<b>EAI_BADFLAGS</b>	Invalid value for <b>ai_flags</b>
<b>EAI_FAIL</b>	Non-recoverable failure in name resolution
<b>EAI_FAMILY</b>	<b>ai_family</b> not supported
<b>EAI_MEMORY</b>	Memory allocation failure
<b>EAI_NODATA</b>	No address associated with <i>hostname</i>
<b>EAI_NONAME</b>	No <i>hostname</i> nor <i>servname</i> provided, or not known
<b>EAI_SERVICE</b>	<i>servname</i> not supported for <b>ai_socktype</b>
<b>EAI_SOCKTYPE</b>	<b>ai_socktype</b> not supported
<b>EAI_SYSTEM</b>	System error returned in <i>errno</i>
<b>EAI_BADEXTFLAGS</b>	Invalid value for <b>ai_eflags</b> .

### Related information

[gai\\_strerror Subroutine](#)

## get\_auth\_method Subroutine

### Purpose

Returns the list of authentication methods for the secure rcmds.

### Library

Authentication Methods Library (**libauthm.a**)

### Syntax

```
int get_auth_method (uint ** authm)
```

### Description

This method returns the authentication methods currently configured in the order in which they should be attempted in the unsigned integer pointer the user passed in.

The list in the unsigned integer pointer is either NULL (on an error) or is an array of unsigned integers terminated by a zero. Each integer identifies an authentication method. The order that a client should attempt to authenticate is defined by the order of the list.

**Note:** The calling routine is responsible for freeing the memory in which the list is contained.

The flags identifying the authentication methods are defined in the `/usr/include/authm.h` file.

## Parameter

Item	Description
<i>authm</i>	Points to an array of unsigned integers. The list of authentication methods is returned in the zero terminated list.

## Return Values

Upon successful completion, the `get_auth_method` subroutine returns a zero.

Upon unsuccessful completion, the `get_auth_method` subroutine returns an **errno**.

## getdomainname Subroutine

### Purpose

Gets the name of the current domain.

### Library

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

### Syntax

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

### Description

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

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

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

**Note:** Domain names are restricted to 256 characters.

### Parameters

Item	Description
<i>Name</i>	Specifies the domain name to be returned.
<i>Na<del>m</del>elen</i>	Specifies the size of the array pointed to by the <i>Name</i> parameter.

## Return Values

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

## Error Codes

The following error may be returned by this subroutine:

Value	Description
<b>EFAULT</b>	The <i>Name</i> parameter gave an invalid address.

## Related reference

[setdomainname Subroutine](#)

## Related information

[Sockets Overview](#)

## gethostbyaddr Subroutine

### Purpose

Gets network host entry by address.

### Library

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

### Syntax

```
#include <netdb.h>
```

```
struct hostent *gethostbyaddr ( Address, Length, Type )  
const void *Address, size_t Length, int Type;
```

### Description

The **gethostbyaddr** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

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

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

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

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

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

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

## Parameters

Item	Description
<i>Address</i>	Specifies a host address. The host address is passed as a pointer to the binary format address.
<i>Length</i>	Specifies the length of host address.
<i>Type</i>	Specifies the domain type of the host address. It can be either AF_INET or AF_INET6.

## Return Values

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

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

## Error Codes

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

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

## Files

Item	Description
<u><a href="#">/etc/hosts</a></u>	Contains the host-name database.
<u><a href="#">/etc/resolv.conf</a></u>	Contains the name server and domain name information.
<u><a href="#">/etc/netsvc.conf</a></u>	Contains the name of the services ordering.
<u><a href="#">/usr/include/netdb.h</a></u>	Contains the network database structure.

### Related reference

[gethostbyname Subroutine](#)

### Related information

[Sockets Overview](#)

[Network Address Translation](#)

## gethostbyaddr\_r Subroutine

### Purpose

Gets network host entry by address.

## Library

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

## Syntax

```
#include <netdb.h>  
int gethostbyaddr_r(Addr, Len, Type, Htent, Ht_data)  
const char *Addr, size_t Len, int Type, struct hostent *Htent, struct hostent_data *Ht_data;
```

## Description

This function internally calls the **gethostbyaddr** subroutine and stores the value returned by the **gethostbyaddr** subroutine to the **hostent** structure.

## Parameters

Item	Description
<i>Addr</i>	Points to the host address that is a pointer to the binary format address.
<i>Len</i>	Specifies the length of the address.
<i>Type</i>	Specifies the domain type of the host address. It can be either <b>AF_INET</b> or <b>AF_INET6</b> .
<i>Htent</i>	Points to a <b>hostent</b> structure which is used to store the return value of the <b>gethostaddr</b> subroutine.
<i>Ht_data</i>	Points to a <b>hostent_data</b> structure.

## Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

**Note:** The return value of the **gethostbyaddr** subroutine points to static data that is overwritten by subsequent calls. This data must be copied at every call to be saved for use by subsequent calls. The **gethostbyaddr\_r** subroutine solves this problem.

If the *Name* parameter is a **hostname**, this subroutine searches for a machine with that name as an IP address. Because of this, use the **gethostbyname\_r** subroutine.

## Error Codes

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

Item	Description
<b>HOST_NOT_FOUND</b>	The host specified by the <i>Name</i> parameter was not found.
<b>TRY_AGAIN</b>	The local server did not receive a response from an authoritative server. Try again later.
<b>NO_RECOVERY</b>	Indicates an unrecoverable error occurred.
<b>NO_ADDRESS</b>	The requested <i>Name</i> parameter is valid but does not have an Internet address at the name server.
<b>SERVICE_UNAVAILABLE</b>	None of the name services specified are running or available.
<b>EINVAL</b>	The <b>hostent</b> pointer is <b>NULL</b> .

## Files

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

## gethostbyname Subroutine

### Purpose

Gets network host entry by name.

### Library

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

### Syntax

```
#include <netdb.h>
```

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

### Description

The **gethostbyname** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

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

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

When using NIS for name resolution, if the **getdomaniname** subroutine is successful and **yp\_bind** indicates yellow pages are running, then the **gethostbyname** subroutine queries NIS for the name.

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

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

### Parameters

Item	Description
<i>Name</i>	Points to the host name.

## Return Values

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

If the parameter *Name* passed to **gethostbyname** is actually an IP address, **gethostbyname** will return a non-NULL **hostent** structure with an IP address as the hostname without actually doing a lookup. Remember to call **inet\_addr** subroutine to make sure *Name* is not an IP address before calling **gethostbyname**. To resolve an IP address call **gethostbyaddr** instead.

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

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

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

## Error Codes

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

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

## Examples

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

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

## Files

Item	Description
<b><u>/etc/hosts</u></b>	Contains the host name data base.
<b><u>/etc/resolv.conf</u></b>	Contains the name server and domain name.
<b><u>/etc/netsvc.conf</u></b>	Contains the name services ordering.
<b><u>/usr/include/netdb.h</u></b>	Contains the network database structure.

### Related reference

[gethostbyaddr Subroutine](#)

[inet\\_addr Subroutine](#)

## Related information

[Sockets Overview](#)

# gethostbyname\_r Subroutine

## Purpose

Gets network host entry by name.

## Library

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

## Syntax

```
#include netdb.h  
int gethostbyname_r(Name, Htent, Ht_data)  
  
const char *Name, struct hostent *Htent, struct hostent_data *Ht_data;
```

## Description

This function internally calls the **gethostbyname** subroutine and stores the value returned by the **gethostbyname** subroutine to the **hostent** structure.

## Parameters

Item	Description
<i>Name</i>	Points to the host name (which is a constant).
<i>Htent</i>	Points to a <b>hostent</b> structure in which the return value of the <b>gethostbyname</b> subroutine is stored.
<i>Ht_data</i>	Points to a <b>hostent_data</b> structure.

## Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

### Note:

The return value of the **gethostbyname** subroutine points to static data that is overwritten by subsequent calls. This data must be copied at every call to be saved for use by subsequent calls. The **gethostbyname\_r** subroutine solves this problem.

If the *Name* parameter is an IP address, this subroutine searches for a machine with that IP address as a name. Because of this, use the **gethostbyaddr\_r** subroutine instead of the **gethostbyname\_r** subroutine if the *Name* parameter is an IP address.

## Error Codes

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

Item	Description
HOST_NOT_FOUND	The host specified by the <i>Name</i> parameter was not found.
TRY_AGAIN	The local server did not receive a response from an authoritative server. Try again later.
NO_RECOVERY	An unrecoverable error occurred.
NO_ADDRESS	The requested <i>Name</i> is valid but does not have an Internet address at the name server.
SERVICE_UNAVAILABLE	None of the name services specified are running or available.
EINVAL	The hostent pointer is NULL.

## Files

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

## gethostent Subroutine

### Purpose

Retrieves a network host entry.

### Library

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

### Syntax

```
#include <netdb.h>

struct hostent *gethostent ()
```

### Description

The **gethostent** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

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

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

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

## Return Values

Upon successful completion, the **gethostent** subroutine returns a pointer to a **hostent** structure. If an error occurs or the end of the file is reached, the **gethostent** subroutine returns a null pointer.

## Files

Item	Description
<a href="#">/etc/hosts</a>	Contains the host name database.
<a href="#">/etc/netsvc.conf</a>	Contains the name services ordering.
<a href="#">/usr/include/netdb.h</a>	Contains the network database structure.

## Related information

[Sockets Overview](#)

[Network Address Translation](#)

## gethostent\_r Subroutine

### Purpose

Retrieves a network host entry.

### Library

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

### Syntax

```
#include <netdb.h>  
  
int gethostent_r (htent, ht_data)  
struct hostent *htent;  
struct hostent_data *ht_data;
```

### Description

When using DNS/BIND name service resolution, the **gethostent\_r** subroutine is not defined.

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

The **gethostent\_r** subroutine internally calls the **gethostent** subroutine, and stores the values in the **htent** and **ht\_data** structures.

The **gethostent** subroutine overwrites the static data returned in subsequent calls. The **gethostent\_r** subroutine does not.

### Parameters

Item	Description
<i>htent</i>	Points to the <b>hostent</b> structure
<i>ht_data</i>	Points to the <b>hostent_data</b> structure

## Return Values

This subroutine returns a 0 if successful, and a -1 if unsuccessful.

## Files

Item	Description
<code>/etc/hosts</code>	Contains the host name database.
<code>/etc/netsvc.conf</code>	Contains the name services ordering.
<code>/usr/include/netdb.h</code>	Contains the network database structure.

## gethostid Subroutine

### Purpose

Gets the unique identifier of the current host.

### Library

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

### Syntax

```
#include <unistd.h>
int gethostid ( )
```

### Description

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

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

### Return Values

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

### Related reference

[sethostname Subroutine](#)

### Related information

[Sockets Overview](#)

## gethostname Subroutine

### Purpose

Gets the name of the local host.

### Library

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

## Syntax

```
#include <unistd.h>
```

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

## Description

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

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

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

## Parameters

Item	Description
<i>Name</i>	Specifies the address of an array of bytes where the host name is to be stored.
<i>NameLength</i>	Specifies the length of the <i>Name</i> array.

## Return Values

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

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

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

## Error Codes

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

Error	Description
<b>EFAULT</b>	The <i>Name</i> parameter or <i>NameLength</i> parameter gives an invalid address.

## Related reference

[sethostname Subroutine](#)

## Related information

[Sockets Overview](#)

## GetMultipleCompletionStatus Subroutine

### Purpose

Dequeues multiple completion packets from a specified I/O completion port.

## Syntax

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

```
int GetMultipleCompletionStatus (CompletionPort, Nmin, Nmax, Timeout, Results[])  
HANDLE CompletionPort;  
DWORD Nmin, Nmax, Timeout;  
struct gmcs {  
    DWORD transfer_count, completion_key, errno;  
    LPOVERLAPPED overlapped;  
} Results[];
```

## Description

The **GetMultipleCompletionStatus** subroutine attempts to dequeue a number of completion packets from the completion port that is specified by the *CompletionPort* parameter. The number of dequeued completion packets that are wanted ranges from the value of the *Nmin* parameter through the value of the *Nmax* parameter. As it collects the packets, this subroutine might wait a predetermined maximum amount of time that is specified by the *Timeout* parameter for the minimum number of completion packets to arrive. If, for example, the X<sup>th</sup> completion packet does not arrive in time, the subroutine returns with only X-1 packets completed.

Either the *Timeout* parameter or a signal might cause a return with completions fewer than the value of the *Nmin* parameter. In other words, *Nmin* completions are not guaranteed to be returned unless the *Timeout* parameter value is set to INFINITE, and a signal does not interrupt the wait. The return of zero completions is not considered an error. The **errno** value will, however, indicate the condition with either the **ETIMEDOUT** or **EINTR** error code. In extreme low-memory situations, the kernel might not be able to provide a timeout. In this case, the system call returns immediately with any available completions, up to the value of the *Nmax* parameter, and the **errno** value is set to **ENOMEM**. Be sure to set the **errno** value to zero before calling the **GetMultipleCompletionStatus** subroutine so that the change of the **errno** value that the subroutine makes can be distinguished from the existing value.

The **GetMultipleCompletionStatus** subroutine is part of the I/O Completion Port (IOCP) kernel extension.

### Note:

1. This subroutine only works with file descriptors of sockets, or regular files for use with the asynchronous I/O subsystem. It does not work with file descriptors of other types.
2. This function must be the exclusive wait mechanism on a completion port. Multiple simultaneous waits through the **GetMultipleCompletionStatus** subroutine, the **GetQueuedCompletionStatus** subroutine, or both, are not supported.
3. When the **GetMultipleCompletionStatus** subroutine is used with the **lio\_listio** subroutine, you can set the value of the *cmd* parameter of the **lio\_listio** subroutine to **LIO\_NOWAIT\_GMCS** to avoid asynchronous updating of the **aiocb** structures, thereby reducing overhead. In this case, you must use the **GetMultipleCompletionStatus** subroutine to wait for I/O completions, and retrieve completion status only from the **struct gmcs** members, not from the **aiocb** structure. When using the **LIO\_NOWAIT\_GMCS** value, do not use the *completion\_key* value in the **gmcs** structure. Do not use the **LIO\_NOWAIT\_AIOWAIT** value with the **lio\_listio** subroutine when using the **GetMultipleCompletionStatus** subroutine. The **LIO\_NOWAIT\_GMCS** value is available for that purpose.
4. Cancelling an asynchronous I/O operation will not affect the **GetMultipleCompletionStatus** subroutine. Even if the cancelling reduces the number of active asynchronous I/O operations to zero, the subroutine will continue to wait.
5. When using the **GetMultipleCompletionStatus** subroutine with sockets, do not wait for multiple completions (*Nmin* > 1) with an **INFINITE** timeout. Use a finite timeout value, and to be prepared to repeat the call if additional completions are still expected.

## Parameters

Item	Description	Attribute description
<i>CompletionPort</i>	Specifies the file descriptor for the completion port that this subroutine will access.	
<i>Nmin</i>	Specifies the minimum number of completions. Fewer might be returned if the value of the <i>timeout</i> parameter is exceeded, or a signal accepted. More might be returned, up to the number that is specified by the <i>Nmax</i> parameter, if additional completions have occurred. Setting the value of the <i>Nmin</i> parameter to zero will poll for completions and return immediately, ignoring the value of the <i>timeout</i> parameter.	
<i>Nmax</i>	Specifies the maximum number of completions to wait for, up to the value of the GMCS_NMAX macro.	
<i>Results</i>	This is the address of an array of the <b>gmcs</b> structure to receive the completion data. The array must contain space for the number of entries specified by the <i>Nmax</i> parameter.	
	<i>Results[i].transfer_count</i>	Specifies the number of bytes transferred. This parameter is set by the subroutine from the value received in the <i>i</i> <sup>th</sup> completion packet. This value is limited to 2 G.
	<i>Results[i].completion_key</i>	Specifies the completion key associated with the file descriptor that is used in the transfer request. This parameter is set by the subroutine from the value received in the <i>i</i> <sup>th</sup> completion packet. Do not use this value with the <b>LIO_NOWAIT_GMCS</b> command parameter of the <b>lio_listio</b> subroutine.
	<i>Results[i].errno</i>	Specifies the <b>errno</b> value that is associated with the <i>i</i> <sup>th</sup> completion packet. When asynchronous I/O requests are started using the <b>lio_listio</b> subroutine with the <b>LIO_NOWAIT_GMCS</b> command parameter, you must use this error value, not the <b>aio_errno</b> member in the <b>aiocb</b> structure, to retrieve the error value that is associated with an I/O request.

Item	Description	Attribute description
	<i>Results[i].overlapped</i>	Specifies the overlapped structure that is used in the transfer request. This parameter is set by the subroutine from the value received in the $i^{\text{th}}$ completion packet. For regular files, this parameter contains a pointer to the asynchronous I/O control block (AIOCB) for a completed AIO request. If an application uses the same completion port for both socket and AIO to regular files, it must use unique <i>completion_key</i> values to differentiate between sockets and regular files to properly interpret the <i>overlapped</i> parameter.
<i>Timeout</i>	Specifies the amount of time in milliseconds that the subroutine is to wait for completion packets. This value can be set to zero. If this parameter is set to INFINITE, the subroutine will never time out.	

## Return Values

Item	Description
Success	The subroutine returns an integer ranging from zero through the value of the <i>Nmax</i> parameter, indicating how many completion packets are dequeued.
Failure	The subroutine returns a value of -1.

## Error codes

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

Item	Description
<b>EINVAL</b>	The value of the <i>CompletionPort</i> or other parameter is not valid.
<b>EBUSY</b>	Another thread is already waiting on the I/O completion port.
<b>EBADF</b>	This error code might also be returned when the value of the <i>CompletionPort</i> parameter is not valid.

If an error occurs after some completions have been handled, the error notifications will be lost. An **EFAULT** error when copying out results can cause the situation.

## Examples

1. The following program fragment illustrates the use of the **GetMultipleCompletionStatus** subroutine to dequeue up to 10 completion packets within a 100-millisecond window.

```
struct gmcs results[10];
int n_results;
HANDLE iocpfd;
errno = 0;
n_results = GetMultipleCompletionStatus(iocpfd, 10, 10, 100, results);
```

## Related information

[lio\\_listio subroutine](#)

[Error Notification Object Class](#)

## getnameinfo Subroutine

### Purpose

Address-to-host name translation [given the binary address and port].

**Note:** This is the reverse functionality of the “getaddrinfo Subroutine” on [page 110](#) host-to-address translation.



**Attention:** This is not a POSIX (1003.1g) specified function.

### Library

Library (**libc.a**)

### Syntax

```
#include <sys/socket.h>
#include <netdb.h>
int
getnameinfo (sa, salen, host, hostlen, serv, servlen, flags)
const struct sockaddr *sa;
char *host;
size_t hostlen;
char *serv;
size_t servlen;
int flags;
```

### Description

The *sa* parameter 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* parameter gives the length of the **sockaddr\_in** or **sockaddr\_in6** structure.

**Note:** A reverse lookup is performed on the IP address and port number provided in *sa*.

The *host* parameter is a buffer where the hostname associated with the IP address is copied. The *hostlen* parameter provides the length of this buffer. The service name associated with the port number is copied into the buffer pointed to by the *serv* parameter. The *servlen* parameter provides the length of this buffer.

The *flags* parameter defines flags that may be used to modify the default actions of this function. By default, the fully-qualified domain name (FQDN) for the host is looked up in DNS and returned.

Item	Description
<b>NI_NOFQDN</b>	If set, return only the hostname portion of the FQDN. If cleared, return the FQDN.
<b>NI_NUMERICHOST</b>	If set, return the numeric form of the host address. If cleared, return the name.
<b>NI_NAMEREQD</b>	If set, return an error if the host's name cannot be determined. If cleared, return the numeric form of the host's address (as if NI_NUMERICHOST had been set).
<b>NI_NUMERICSERV</b>	If set, return the numeric form of the desired service. If cleared, return the service name.
<b>NI_DGRAM</b>	If set, consider the desired service to be a datagram service, (for example, call <b>getserbyport</b> with an argument of <b>udp</b> ). If clear, consider the desired service to be a stream service (for example, call <b>getserbyport</b> with an argument of <b>tcp</b> ).

## Return Values

A zero return value indicates successful completion; a non-zero value indicates failure. If successful, the strings for hostname and service name are copied into the *host* and *serv* buffers, respectively. If either the host or service name cannot be located, the numeric form is copied into the *host* and *serv* buffers, respectively.

### Related information

[gai\\_strerror Subroutine](#)

[Subroutines Overview](#)

## getnetbyaddr Subroutine

### Purpose

Gets network entry by address.

### Library

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

### Syntax

```
#include <netdb.h>
```

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

### Description

The **getnetbyaddr** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

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

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

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

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

### Parameters

Item	Description
<i>Network</i>	Specifies the number of the network to be located.
<i>Type</i>	Specifies the address family for the network. The only supported value is <b>AF_INET</b> .

### Return Values

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

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

## Files

Item	Description
<a href="#">/etc/networks</a>	Contains official network names.

### Related reference

[endnetent](#) Subroutine

[setnetent](#) Subroutine

### Related information

[Sockets Overview](#)

## getnetbyaddr\_r Subroutine

### Purpose

Gets network entry by address.

### Library

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

### Syntax

```
#include<netdb.h>
int getnetbyaddr_r(net, type, netent, net_data)

register in_addr_t net;
register int type;
struct netent *netent;
struct netent_data *net_data;
```

### Description

The **getnetbyaddr\_r** subroutine retrieves information from the **/etc/networks** file using the *Name* parameter as a search key.

The **getnetbyaddr\_r** subroutine internally calls the **getnetbyaddr** subroutine and stores the information in the structure data.

The **getnetbyaddr** subroutine overwrites the static data returned in subsequent calls. The **getnetbyaddr\_r** subroutine does not.

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

### Parameters

Item	Description
<i>Net</i>	Specifies the number of the network to be located.
<i>Type</i>	Specifies the address family for the network. The only supported values are AF_INET, and AF_INET6.
<i>netent</i>	Points to the <b>netent</b> structure.
<i>net_data</i>	Points to the <b>net_data structure</b> .

## Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

## Files

Item	Description
<code>/etc/networks</code>	Contains official network names.

## getnetbyname Subroutine

### Purpose

Gets network entry by name.

### Library

Standard C Library (`libc.a`)

### Syntax

```
#include <netdb.h>
```

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

### Description

The `getnetbyname` subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

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

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

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

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

### Parameters

Item	Description
<i>Name</i>	Points to a string containing the name of the network.

### Return Values

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

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

## Files

### Item

[/etc/networks](#)

### Description

Contains official network names.

### Related reference

[endnetent Subroutine](#)

### Related information

[Sockets Overview](#)

## getnetbyname\_r Subroutine

### Purpose

Gets network entry by name.

### Library

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

### Syntax

```
#include <netdb.h>
int getnetbyname_r(Name, netent, net_data)
register const char *Name;
struct netent *netent;
struct netent_data *net_data;
```

### Description

The **getnetbyname\_r** subroutine retrieves information from the **/etc/networks** file using the *Name* parameter as a search key.

The **getnetbyname\_r** subroutine internally calls the **getnetbyname** subroutine and stores the information in the structure data.

The **getnetbyname** subroutine overwrites the static data returned in subsequent calls. The **getnetbyname\_r** subroutine does not.

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

### Parameters

Item	Description
<i>Name</i>	Points to a string containing the name of the network.
<i>netent</i>	Points to the <b>netent</b> structure.
<i>net_data</i>	Points to the <b>net_data</b> structure.

### Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

**Note:** If an error occurs or the end of the file is reached, the **getnetbyname\_r** subroutine returns a -1 to indicate error.

## Files

Item	Description
<a href="#"><u>/etc/networks</u></a>	Contains official network names.

## getnetent Subroutine

### Purpose

Gets network entry.

### Library

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

### Syntax

```
#include <netdb.h>
```

```
struct netent *getnetent ( )
```

### Description

The **getnetent** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **getnetent** subroutine retrieves network information by opening and sequentially reading the [/etc/networks](#) file.

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

Use the [endnetent](#) subroutine to close the [/etc/networks](#) file.

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

### Return Values

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

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

## Files

Item	Description
<a href="#"><u>/etc/networks</u></a>	Contains official network names.

### Related reference

[setnetent Subroutine](#)

### Related information

[Sockets Overview](#)

## getnetent\_r Subroutine

### Purpose

Gets network entry.

### Library

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

### Syntax

```
#include <netdb.h>
int getnetent_r(netent, net_data)

struct netent *netent;
struct netent_data *net_data;
```

### Description

The **getnetent\_r** subroutine retrieves network information by opening and sequentially reading the **/etc/networks** file. This subroutine internally calls the **getnetent** subroutine and stores the values in the hostent structure.

The **getnetent** subroutine overwrites the static data returned in subsequent calls. The **getnetent\_r** subroutine does not. Use the **endnetent\_r** subroutine to close the **/etc/networks** file.

### Parameters

Item	Description
<i>netent</i>	Points to the <b>netent</b> structure.
<i>net_data</i>	Points to the <b>net_data</b> structure.

### Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

**Note:** If an error occurs or the end of the file is reached, the **getnetent\_r** subroutine returns a -1 to indicate error.

### Files

Item	Description
<b>/etc/networks</b>	Contains official network names.

## getnetgrent\_r Subroutine

### Purpose

Handles the group network entries.

### Library

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

## Syntax

```
#include <netdb.h>
int getnetgrent_r(machinep, namep, domainp, ptr)
char **machinep, **namep, **domainp;
void **ptr;
```

## Description

The **getnetgrent\_r** subroutine internally calls the **getnetgrent** subroutine and stores the information in the structure data. This subroutine returns 1 or 0, depending if netgroup contains the machine, user, and domain triple as a member. Any of these three strings can be NULL, in which case it signifies a wildcard.

The **getnetgrent\_r** subroutine returns the next member of a network group. After the call, the *machinep* parameter contains a pointer to a string containing the name of the machine part of the network group member. The *namep* and *domainp* parameters contain similar pointers. If *machinep*, *namep*, or *domainp* is returned as a NULL pointer, it signifies a wildcard.

The **getnetgrent** subroutine overwrites the static data returned in subsequent calls. The **getnetgrent\_r** subroutine does not.

## Parameters

Item	Description
<i>machinep</i>	Points to the string containing the machine part of the network group.
<i>namep</i>	Points to the string containing the user part of the network group.
<i>domainp</i>	Points to the string containing the domain name.
<i>ptr</i>	Keeps the function threadsafe.

## Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

## Files

Item	Description
<b>/etc/netgroup</b>	Contains network groups recognized by the system.
<b>/usr/include/netdb.h</b>	Contains the network database structures.

## getpeername Subroutine

### Purpose

Gets the name of the peer socket.

### Library

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

## Syntax

```
#include <sys/socket.h>
int getpeername ( Socket, Name, NameLength )
```

```
int Socket;
struct sockaddr *Name;
socklen_t *NameLength;
```

## Description

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

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

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

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

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

## Parameters

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

## Return Values

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

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

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

## Error Codes

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

Error	Description
<b>EBADF</b>	The <i>Socket</i> parameter is not valid.
<b>EINVAL</b>	The socket has been shut down.
<b>ENOTSOCK</b>	The <i>Socket</i> parameter refers to a file, not a socket.
<b>ENOTCONN</b>	The socket is not connected.
<b>ENOBUFS</b>	Insufficient resources were available in the system to complete the call.
<b>EFAULT</b>	The <i>Address</i> parameter is not in a writable part of the user address space.

## Examples

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

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

### Related reference

[getsockname Subroutine](#)

### Related information

[Sockets Overview](#)

## getprotobyname Subroutine

### Purpose

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

### Library

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

### Syntax

```
#include <netdb.h>
```

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

### Description

The **getprotobyname** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

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

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

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

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

## Parameters

Item	Description
<i>Name</i>	Specifies the protocol name.

## Return Values

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

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

### Related reference

[endprotoent Subroutine](#)

[setprotoent Subroutine](#)

[setservent Subroutine](#)

### Related information

[Sockets Overview](#)

## getprotobyname\_r Subroutine

### Purpose

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

### Library

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

### Syntax

```
#include <netdb.h>

int getprotobyname_r(Name, protoent, proto_data)
register const char *Name;
struct protoent *protoent;
struct protoent_data *proto_data;
```

### Description

The **getprotobyname\_r** subroutine retrieves protocol information from the **/etc/protocols** file by protocol name.

An application program can use the **getprotobyname\_r** subroutine to access a protocol name, aliases, and protocol number.

The **getprotobyname\_r** subroutine searches the protocols file sequentially from the start of the file until it finds a matching protocol name or until it reaches the end of the file. The subroutine writes the **protoent** structure, which contains fields for a line of information in the **/etc/protocols** file.

The **netdb.h** file defines the **protoent** structure.

The **getprotobyname** subroutine overwrites any static data returned in subsequent calls. The **getprotobyname\_r** subroutine does not.

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

## Parameters

Item	Description
<i>Name</i>	Specifies the protocol name.
<i>protoent</i>	Points to the <b>protoent</b> structure.
<i>proto_data</i>	Points to the <b>proto_data</b> structure.

## Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

**Note:** If an error occurs or the end of the file is reached, the **getprotobyname\_r** subroutine returns a -1 to indicate error.

## getprotobynumber Subroutine

### Purpose

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

### Library

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

### Syntax

```
#include <netdb.h>
```

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

### Description

The **getprotobynumber** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

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

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

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

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

## Parameters

Item	Description
<i>Protocol</i>	Specifies the protocol number.

## Return Values

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

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

## Files

Item	Description
<a href="#"><u>/etc/protocols</u></a>	Contains protocol information.

### Related reference

[endprotoent Subroutine](#)

### Related information

[Sockets Overview](#)

## getprotobynumber\_r Subroutine

### Purpose

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

### Library

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

### Syntax

```
#include <netdb.h>
int getprotobynumber_r(proto, protoent, proto_data)
register int proto;
struct protoent *protoent;
struct protoent_data *proto_data;
```

### Description

The **getprotobynumber\_r** subroutine retrieves protocol information from the **/etc/protocols** file using a specified protocol number as a search key.

An application program can use the **getprotobynumber\_r** subroutine to access a protocol name, aliases, and number.

The **getprotobynumber\_r** subroutine searches the **/etc/protocols** file sequentially from the start of the file until it finds a matching protocol name, protocol number, or until it reaches the end of the file.

The subroutine writes the **protoent** structure, which contains fields for a line of information in the **/etc/protocols** file.

The **netdb.h** file defines the **protoent** structure.

The **getprotobynumber** subroutine overwrites static data returned in subsequent calls. The **getprotobynumber\_r** subroutine does not.

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

## Parameters

Item	Description
<i>proto</i>	Specifies the protocol number.
<i>protoent</i>	Points to the <b>protoent</b> structure.
<i>proto_data</i>	Points to the <b>proto_data</b> structure.

## Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

**Note:** If an error occurs or the end of the file is reached, the **getprotobynumber\_r** subroutine sets the *protoent* parameter to NULL and returns a -1 to indicate error.

## Files

Item	Description
<i>/etc/protocols</i>	Contains protocol information.

## getprotoent Subroutine

### Purpose

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

### Library

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

### Syntax

```
#include <netdb.h>
```

```
struct protoent *getprotoent ( )
```

### Description

The **getprotoent** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

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

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

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

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

### Return Values

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

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

## Files

Item	Description
<a href="#">/etc/protocols</a>	Contains protocol information.

## Related reference

[endprotoent Subroutine](#)

## Related information

[Sockets Overview](#)

## getprotoent\_r Subroutine

### Purpose

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

### Library

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

### Syntax

```
#include <netdb.h>

int getprotoent_r(protoent, proto_data)
struct protoent *protoent;
struct protoent_data *proto_data;
```

### Description

The **getprotoent\_r** subroutine retrieves protocol information from the **/etc/protocols** file. An application program can use the **getprotoent\_r** subroutine to access a protocol name, its aliases, and protocol number. The **getprotoent\_r** subroutine opens and performs a sequential read of the **/etc/protocols** file. This subroutine writes to the **protoent** structure, which contains the fields for a line of information in the **/etc/protocols** file.

The **netdb.h** file defines the **protoent** structure.

Use the **endprotoent\_r** subroutine to close the **/etc/protocols** file. Static data is overwritten in subsequent calls when using the **getprotoent** subroutine. The **getprotoent\_r** subroutine does not overwrite.

### Parameters

Item	Description
<i>protoent</i>	Points to the <b>protoent</b> structure.
<i>proto_data</i>	Points to the <b>proto_data</b> structure.

### Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

**Note:** If an error occurs or the end of the file is reached, the **getprotoent\_r** subroutine sets the *protoent* parameter to NULL.

## Files

Item	Description
<code>/etc/protocols</code>	Contains protocol information.

## GetQueuedCompletionStatus Subroutine

### Purpose

Dequeues a completion packet from a specified I/O completion port.

### Syntax

```
#include <iocp.h>
boolean_t GetQueuedCompletionStatus (CompletionPort, TransferCount, CompletionKey, Overlapped,
Timeout)
HANDLE CompletionPort;
LPDWORD TransferCount, CompletionKey;
LPOVERLAPPED Overlapped; DWORD Timeout;
```

### Description

The **GetQueuedCompletionStatus** subroutine attempts to dequeue a completion packet from the *CompletionPort* parameter. If there is no completion packet to be dequeued, this subroutine waits a predetermined amount of time as indicated by the *Timeout* parameter for a completion packet to arrive.

The **GetQueuedCompletionStatus** subroutine returns a boolean indicating whether or not a completion packet has been dequeued.

The **GetQueuedCompletionStatus** subroutine is part of the I/O Completion Port (IOCP) kernel extension.

**Note:** This subroutine only works with file descriptors of sockets, or regular files for use with the Asynchronous I/O (AIO) subsystem. It does not work with file descriptors of other types.

### Parameters

Item	Description
<i>CompletionPort</i>	Specifies the completion port that this subroutine will attempt to access.
<i>TransferCount</i>	Specifies the number of bytes transferred. This parameter is set by the subroutine from the value received in the completion packet.
<i>CompletionKey</i>	Specifies the completion key associated with the file descriptor used in the transfer request. This parameter is set by the subroutine from the value received in the completion packet.
<i>Overlapped</i>	Specifies the overlapped structure used in the transfer request. This parameter is set by the subroutine from the value received in the completion packet. For regular files, this parameter contains a pointer to the AIOCB for a completed AIO request. If an application uses the same completion port for both socket and AIO to regular files, it must use unique <i>CompletionKey</i> values to differentiate between sockets and regular files in order to properly interpret the <i>Overlapped</i> parameter.
<i>Timeout</i>	Specifies the amount of time in milliseconds the subroutine is to wait for a completion packet. If this parameter is set to INFINITE, the subroutine will never timeout.

## Return Values

Upon successful completion, the **GetQueuedCompletionStatus** subroutine returns a boolean indicating its success.

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

- Returns a value of 0 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable, see the link in the Related Information section of this document.

## Error Codes

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

Item	Description
<b>ETIMEDOUT</b>	No completion packet arrived to be dequeued and the <i>Timeout</i> parameter has elapsed.
<b>EINVAL</b>	The value of the <i>CompletionPort</i> or other parameter is not valid.
<b>EAGAIN</b>	Resource temporarily unavailable. If a sleep is interrupted by a signal, <b>EAGAIN</b> may be returned.
<b>ENOTCONN</b>	Socket is not connected. The <b>ENOTCONN</b> return can happen for two reasons. One is if a request is made, the <b>fd</b> is then closed, then the request is returned back to the process. The error will be <b>ENOTCONN</b> . The other is if the socket drops while the <b>fd</b> is still open, the requests after the socket drops (disconnects) will return <b>ENOTCONN</b> .
<b>EBADF</b>	This error code might also be returned when the value of the <i>CompletionPort</i> parameter is not valid.

## Examples

The following program fragment illustrates the use of the **GetQueuedCompletionStatus** subroutine to dequeue a completion packet.

```
int transfer_count, completion_key
LPOVERLAPPED overlapped;
c = GetQueuedCompletionStatus (34, &transfer_count, &completion_key, &overlapped, 1000);
```

### Related information

[Error Notification Object Class](#)

## getservbyname Subroutine

### Purpose

Gets service entry by name.

### Library

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

### Syntax

```
#include <netdb.h>
```

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

## Description

The **getservbyname** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

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

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

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

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

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

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

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

## Parameters

Item	Description
<i>Name</i>	Specifies the name of a service.
<i>Protocol</i>	Specifies a protocol for use with the specified service.

## Return Values

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

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

## Files

Item	Description
<b><u>/etc/services</u></b>	Contains service names.

### Related reference

[endservent Subroutine](#)

### Related information

[Sockets Overview](#)

[Understanding Network Address Translation](#)

## getservbyname\_r Subroutine

### Purpose

Gets service entry by name.

### Library

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

### Syntax

```
#include <netdb.h>
int getservbyname_r(name, proto, servent, serv_data)
const char *Name, proto;
struct servent *servent;
struct servent_data *serv_data;
```

### Description

**Requirement:** Use the **getservbyname** subroutine instead of the **getservbyname\_r** subroutine. The **getservbyname\_r** subroutine is compatible only with earlier versions of AIX.

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

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

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

Upon locating a matching name and protocol, the **getservbyname\_r** subroutine stores the values to the **servent** structure. The **getservbyname** subroutine overwrites the static data it returns in subsequent calls. The **getservbyname\_r** subroutine does not.

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

You must fill the **servent\_data** structure with zeros before its first access by either the **setservent\_r** or the **getservbyname\_r** subroutine.

### Parameters

Item	Description
<i>name</i>	Specifies the name of a service.
<i>proto</i>	Specifies a protocol for use with the specified service.
<i>servent</i>	Points to the <b>servent</b> structure.
<i>serv_data</i>	Points to the <b>serv_data</b> structure.

### Return Values

The function returns a 0 if successful and a -1 if unsuccessful. The **getservbyname** subroutine returns a pointer to a **servent** structure when a successful match occurs. Entries in this structure are in network byte order.

**Note:** If an error occurs or the end of the file is reached, the **getservbyname\_r** returns a -1.

## Files

Item	Description
<code>/etc/services</code>	Contains service names.

## getservbyport Subroutine

### Purpose

Gets service entry by port.

### Library

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

### Syntax

```
#include <netdb.h>
```

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

### Description

The **getservbyport** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

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

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

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

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

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

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

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

### Parameters

Item	Description
<i>Port</i>	Specifies the port where a service resides.
<i>Protocol</i>	Specifies a protocol for use with the service.

## Return Values

Upon successful completion, the **getservbyport** subroutine returns a pointer to a **servent** structure. If an error occurs or the end of the file is reached, the **getservbyport** subroutine returns a null pointer.

## Files

Item	Description
<a href="#"><u>/etc/services</u></a>	Contains service names.

### Related reference

[endservent Subroutine](#)

[endprotoent Subroutine](#)

### Related information

[Sockets Overview](#)

## getservbyport\_r Subroutine

### Purpose

Gets service entry by port.

### Library

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

### Syntax

```
#include <netdb.h>

int getservbyport_r(Port, Proto, servent, serv_data)
int Port;
const char *Proto;
struct servent *servent;
struct servent_data *serv_data;
```

### Description

The **getservbyport\_r** subroutine retrieves an entry from the **/etc/services** file using a port number as a search key. An application program can use the **getservbyport\_r** subroutine to access a service, service aliases, the protocol for the service, and a protocol port number for the service.

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

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

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

The **getservbyport** routine overwrites static data returned on subsequent calls. The **getservbyport\_r** routine does not.

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

## Parameters

Item	Description
<i>Port</i>	Specifies the port where a service resides.
<i>Proto</i>	Specifies a protocol for use with the service.
<i>servent</i>	Points to the <b>servent</b> structure.
<i>serv_data</i>	Points to the <b>serv_data</b> structure.

## Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

**Note:** If an error occurs or the end of the file is reached, the **getservbyport\_r** subroutine returns a -1 to indicate error.

## Files

Item	Description
<i>/etc/services</i>	Contains service names.

## getservent Subroutine

### Purpose

Gets services file entry.

### Library

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

### Syntax

```
#include <netdb.h>
```

```
struct servent *getservent ( )
```

### Description

The **getservent** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

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

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

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

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

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

## Return Values

The **getservent** subroutine returns a pointer to a **servent** structure when a successful match occurs. If an error occurs or the end of the file is reached, the **getservent** subroutine returns a null pointer.

## Files

Item	Description
<a href="#"><u>/etc/services</u></a>	Contains service names.

### Related reference

[endprotoent Subroutine](#)

### Related information

[Sockets Overview](#)

[Understanding Network Address Translation](#)

## getservent\_r Subroutine

### Purpose

Gets services file entry.

### Library

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

### Syntax

```
#include <netdb.h>
int getservent_r(servent, serv_data)
struct servent *servent;
struct servent_data *serv_data;
```

### Description

The **getservent\_r** subroutine opens and reads the next line of the **/etc/services** file. An application program can use the **getservent\_r** subroutine to retrieve information about network services and the protocol ports they use.

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

### Parameters

Item	Description
<i>servent</i>	Points to the <b>servent</b> structure.
<i>serv_data</i>	Points to the <b>serv_data</b> structure.

### Return Values

The **getservent\_r** fails when a successful match occurs. The **getservent** subroutine overwrites static data returned on subsequent calls. The **getservent\_r** subroutine does not.

## Files

Item	Description
<code>/etc/services</code>	Contains service names.

## getsockname Subroutine

### Purpose

Gets the socket name.

### Library

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

### Syntax

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

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

### Description

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

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

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

### Parameters

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

### Return Values

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

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

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

- For sockets in the AF\_UNIX domain, if the returned value of the **NameLength** parameter is greater than 255, the corresponding value of the **sun\_len** field in the overloaded sockaddr structure is assigned an address of 0xFF because of the bit size limitations of the **sun\_len** field.

## Error Codes

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

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

### Related reference

[getpeername Subroutine](#)

[socket Subroutine](#)

[socks5tcp\\_bind Subroutine](#)

[socks5tcp\\_connect Subroutine](#)

### Related information

[Checking for Pending Connections Example Program](#)

[Sockets Overview](#)

## getsockopt Subroutine

### Purpose

Gets options on sockets.

### Library

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

### Syntax

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

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

### Description

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

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

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

# Parameters

Item	Description
<i>Socket</i>	Specifies the unique socket name.
<i>Level</i>	Specifies the protocol level where the option resides. Options can be retrieved at the following levels: <b>Socket level</b> Specifies the <i>Level</i> parameter as the <b>SOL_SOCKET</b> option. <b>Other levels</b> Supplies the appropriate protocol number for the protocol controlling the option. For example, to indicate that an option will be interpreted by the TCP protocol, set the <i>Level</i> parameter to the protocol number of TCP, as defined in the <b>netinet/in.h</b> file.
<i>OptionName</i>	Specifies a single option. The <i>OptionName</i> parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The <b>sys/socket.h</b> file contains definitions for socket level options. The <b>netinet/tcp.h</b> file contains definitions for TCP protocol level options. Socket-level options can be enabled or disabled; they operate in a toggle fashion. The <b>sys/atmsock.h</b> file contains definitions for ATM protocol level options. The following list defines socket protocol level options found in the <b>sys/socket.h</b> file: <b>SO_DEBUG</b> Specifies the recording of debugging information. This option enables or disables debugging in the underlying protocol modules. <b>SO_BROADCAST</b> Specifies whether transmission of broadcast messages is supported. The option enables or disables broadcast support. <b>SO_CKSUMREV</b> Enables performance enhancements in the protocol layers. If the protocol supports this option, enabling causes the protocol to defer checksum verification until the user's data is moved into the user's buffer (on <b>recv</b> , <b>recvfrom</b> , <b>read</b> , or <b>recvmsg</b> thread). This can cause applications to be awakened when no data is available, in the case of a checksum error. In this case, EAGAIN is returned. Applications that set this option must handle the EAGAIN error code returned from a receive call. <b>SO_REUSEADDR</b> Specifies that the rules used in validating addresses supplied by a <b>bind</b> subroutine should allow reuse of a local port. A particular IP address can only be bound once to the same port. This option enables or disables reuse of local ports. <b>SO_REUSEADDR</b> allows an application to explicitly deny subsequent <b>bind</b> subroutine to the port/address of the socket with <b>SO_REUSEADDR</b> set. This allows an application to block other applications from binding with the <b>bind</b> subroutine. <b>SO_REUSEPORT</b> Specifies that the rules used in validating addresses supplied by a <b>bind</b> subroutine should allow reuse of a local port/address combination. Each binding of the port/address combination must specify the <b>SO_REUSEPORT</b> socket option. This option enables or disables the reuse of local port/address combinations. <b>SO_KEEPAIVE</b> Monitors the activity of a connection by enabling or disabling the periodic transmission of ACK messages on a connected socket. The idle interval time can be designated using the TCP/IP <b>no</b> command. Broken connections are discussed in " <a href="#">Understanding Socket Types and Protocols</a> " in <i>Communications Programming Concepts</i> . <b>SO_DONTROUTE</b> Indicates outgoing messages should bypass the standard routing facilities. Does not apply routing on outgoing messages. Directs messages to the appropriate network interface according to the network portion of the destination address. This option enables or disables routing of outgoing messages. <b>SO_LINGER</b> Lingers on a <b>close</b> subroutine if data is present. This option controls the action taken when an unsent messages queue exists for a socket, and a process performs a <b>close</b> subroutine on the socket. If the <b>SO_LINGER</b> option is set, the system blocks the process during the <b>close</b> subroutine until it can transmit the data or until the time expires. If the <b>SO_LINGER</b> option is not specified, and a <b>close</b> subroutine is issued, the system handles the call in a way that allows the process to continue as quickly as possible. The <b>sys/socket.h</b> file defines the <b>linger</b> structure that contains the <b>L_linger</b> member for specifying linger time interval. If linger time is set to anything but 0, the system tries to send any messages queued on the socket. The maximum value that the <b>L_linger</b> member can be set to is 65535. If the application has requested SPEC1170 compliant behavior by exporting the XPG_SUS_ENV environment variable, the linger time is <i>n</i> seconds; otherwise, the linger time is <i>n</i> /100 seconds (ticks), where <i>n</i> is the value of the <b>L_linger</b> member. <b>SO_OOBINLINE</b> Leaves received out-of-band data (data marked urgent) in line. This option enables or disables the receipt of out-of-band data. <b>SO_SNDBUF</b> Retrieves buffer size information. <b>SO_RCVBUF</b> Retrieves buffer size information. <b>SO_SNDLOWAT</b> Retrieves send buffer low-water mark information. <b>SO_RCVLOWAT</b> Retrieves receive buffer low-water mark information.
<i>OptionName (contd)</i>	

Item	Description
<i>OptionName</i> (contd)	<p><b>SO_SNDTIMEO</b> Retrieves time-out information. This option is settable, but currently not used.</p> <p><b>SO_RCVTIMEO</b> Retrieves time-out information. This option is settable, but currently not used.</p> <p><b>SO_PEERID</b> Retrieves the credential information of the process associated with a peer UNIX domain socket. This information includes the process ID, effective user ID, and effective group ID. The <code>peercred_struct</code> structure must be used in order to get the credential information. This structure is defined in the <code>sys/socket.h</code> file.</p> <p><b>SO_ERROR</b> Retrieves information about error status and clears.</p> <p><b>SO_TYPE</b> Sets the retrieval of a socket type.</p> <p>The following list defines TCP protocol level options found in the <code>netinet/tcp.h</code> file:</p> <p><b>TCP_CWND_IF</b> Increases the factor of the TCP congestion window (cwnd) during the congestion avoidance. The value must be in the range 0 - 100 (0 is disable). The <code>tcp_cwnd_modified</code> network tunable option must be enabled.</p> <p><b>TCP_CWND_DF</b> Decrease the factor of the TCP cwnd during the congestion avoidance. The value must be in the range 0 - 100 (0 is disable). The <code>tcp_cwnd_modified</code> network tunable option must be enabled.</p> <p><b>TCP_NOTENTER_SSTART</b> Avoids reentering the slow start after the retransmit timeout, which might reset the cwnd to the initial window size, instead of the size of the current slow-start threshold (<code>ss_threshold</code>) value or half of the maximum cwnd (<code>max_cwnd/2</code>). The values are 1 for enable and 0 for disable. The <code>tcp_cwnd_modified</code> network tunable option must be enabled.</p> <p><b>TCP_NOREDUCE_CWND_IN_FRXMT</b> Not decrease the cwnd size when in the fast retransmit phrase. The values are 1 for enable and 0 for disable. The <code>tcp_cwnd_modified</code> network tunable option must be enabled.</p> <p><b>TCP_NOREDUCE_CWND_EXIT_FRXMT</b> Not decrease the cwnd size when exits the fast retransmit phrase. The values are 1 for enable and 0 for disable. The <code>tcp_cwnd_modified</code> network tunable option must be enabled.</p> <p><b>TCP_RFC1323</b> Indicates whether RFC 1323 is enabled or disabled on the specified socket. A non-zero <i>OptionValue</i> returned by the <code>getsockopt</code> subroutine indicates the RFC is enabled.</p> <p><b>TCP_NODELAY</b> Specifies whether TCP should follow the Nagle algorithm for deciding when to send data. By default TCP will follow the Nagle algorithm. To disable this behavior, applications can enable <b>TCP_NODELAY</b> to force TCP to always send data immediately. A non-zero <i>OptionValue</i> returned by the <code>getsockopt</code> subroutine indicates <b>TCP_NODELAY</b> is enabled. For example, <b>TCP_NODELAY</b> should be used when there is an application using TCP for a request/response.</p>
<i>OptionName</i> (contd)	<p><b>TCP_NODELAYACK</b> Specifies if TCP needs to send immediate acknowledgement packets to the sender. If this option is not set, TCP delays sending the acknowledgement packets by up to 200 ms. This allows the acknowledgements to be sent along with the data on a response and minimizes system overhead. Setting this TCP option might cause a slight increase in system overhead, but can result in higher performance for network transfers if the sender is waiting on the receiver's acknowledgements.</p> <p>The following list defines ATM protocol level options found in the <code>sys/atmsock.h</code> file:</p> <p><b>SO_ATM_PARM</b> Retrieves all ATM parameters. This socket option can be used instead of using individual sockets options described below. It uses the <code>connect_ie</code> structure defined in <code>sys/call_ie.h</code> file.</p> <p><b>SO_ATM_AAL_PARM</b> Retrieves ATM AAL (Adaptation Layer) parameters. It uses the <code>aal_parm</code> structure defined in <code>sys/call_ie.h</code> file.</p> <p><b>SO_ATM_TRAFFIC_DES</b> Retrieves ATM Traffic Descriptor values. It uses the <code>traffic_desc</code> structure defined in <code>sys/call_ie.h</code> file.</p> <p><b>SO_ATM_BEARER</b> Retrieves ATM Bearer capability information. It uses the <code>bearer</code> structure defined in <code>sys/call_ie.h</code> file.</p> <p><b>SO_ATM_BHLI</b> Retrieves ATM Broadband High Layer Information. It uses the <code>bhli</code> structure defined in <code>sys/call_ie.h</code> file.</p> <p><b>SO_ATM_BLLI</b> Retrieves ATM Broadband Low Layer Information. It uses the <code>blli</code> structure defined in <code>sys/call_ie.h</code> file.</p> <p><b>SO_ATM_QoS</b> Retrieves ATM Quality Of Service values. It uses the <code>qos_parm</code> structure defined in <code>sys/call_ie.h</code> file.</p> <p><b>SO_ATM_TRANSIT_SEL</b> Retrieves ATM Transit Selector Carrier. It uses the <code>transit_sel</code> structure defined in <code>sys/call_ie.h</code> file.</p> <p><b>SO_ATM_MAX_PEND</b> Retrieves the number of outstanding transmit buffers that are permitted before an error indication is returned to applications as a result of a transmit operation. This option is only valid for non best effort types of virtual circuits.</p> <p><b>SO_ATM_CAUSE</b> Retrieves cause for the connection failure. It uses the <code>cause_t</code> structure defined in the <code>sys/call_ie.h</code> file.</p>

Item	Description
<i>OptionValue</i>	Specifies a pointer to the address of a buffer. The <i>OptionValue</i> parameter takes an integer parameter. The <i>OptionValue</i> parameter should be set to a nonzero value to enable a Boolean option or to a value of 0 to disable the option. The following options enable and disable in the same manner: <ul style="list-style-type: none"> <li>• <b>SO_DEBUG</b></li> <li>• <b>SO_REUSEADDR</b></li> <li>• <b>SO_KEEPALIVE</b></li> <li>• <b>SO_DONTROUTE</b></li> <li>• <b>SO_BROADCAST</b></li> <li>• <b>SO_OOBINLINE</b></li> <li>• <b>TCP_RFC1323</b></li> </ul>
<i>OptionLength</i>	Specifies the length of the <i>OptionValue</i> parameter. The <i>OptionLength</i> parameter initially contains the size of the buffer pointed to by the <i>OptionValue</i> parameter. On return, the <i>OptionLength</i> parameter is modified to indicate the actual size of the value returned. If no option value is supplied or returned, the <i>OptionValue</i> parameter can be 0.

Options at other protocol levels vary in format and name.

Item	Description
<b>IP_DONTFRAG</b>	Get current <b>IP_DONTFRAG</b> option value.
<b>IP_FINDPMTU</b>	Get current PMTU value.
<b>IP_PMTUAGE</b>	Get current PMTU time out value.

Item	Description
<b>IP_DONTGRAG</b>	Not supported.
<b>IP_FINDPMTU</b>	Get current PMTU value.
<b>IP_PMTUAGE</b>	Not supported.

Item	Description	Value
<b>IPV6_V6ONLY</b>	Determines whether the socket is restricted to IPv6 communications only.	<b>Option Type:</b> int (Boolean interpretation)
	Allows the user to determine the outgoing hop limit value for unicast IPv6 packets.	<b>Option Type:</b> int
	Allows the user to determine the outgoing hop limit value for multicast IPv6 packets.	<b>Option Type:</b> int
	Allows the user to determine the interface being used for outgoing multicast packets.	<b>Option Type:</b> unsigned int
	If a multicast datagram is sent to a group that the sending host belongs to, a copy of the datagram is looped back by the IP layer for local delivery (if the option is set to 1). If the option is set to 0, a copy is not looped back.	<b>Option Type:</b> unsigned int
	Determines whether the destination IPv6 address and arriving interface index of incoming IPv6 packets are being received as ancillary data on UDP and raw sockets.	<b>Option Type:</b> int (Boolean interpretation)
	Determines whether the hop limit of incoming IPv6 packets is being received as ancillary data on UDP and raw sockets.	<b>Option Type:</b> int (Boolean interpretation)
	Determines whether the traffic class of incoming IPv6 packets is being received as ancillary data on UDP and raw sockets.	<b>Option Type:</b> int (Boolean interpretation)
	Determines whether the routing header of incoming IPv6 packets is being received as ancillary data on UDP and raw sockets.	<b>Option Type:</b> int (Boolean interpretation)
	Determine whether the hop-by-hop options header of incoming IPv6 packets is being received as ancillary data on UDP and raw sockets.	<b>Option Type:</b> int (Boolean interpretation)
	Determines whether the destination options header of incoming IPv6 packets is being received as ancillary data on UDP and raw sockets.	<b>Option Type:</b> int (Boolean interpretation)

Item	Description	Value
	Determines the source IPv6 address and outgoing interface index for all IPv6 packets being sent on this socket.	<b>Option Type:</b> <b>struct in6_pktinfo</b> defined in the <b>netinet/in.h</b> file.
	Determines the next hop being used for outgoing IPv6 datagrams on this socket.	<b>Option Type:</b> <b>struct sockaddr_in6</b> defined in the <b>netinet/in.h</b> file.
	Determines the traffic class for outgoing IPv6 datagrams on this socket.	<b>Option Type:</b> <b>int</b>
	Determines the routing header to be used for outgoing IPv6 datagrams on this socket.	<b>Option Type:</b> <b>struct ip6_rthdr</b> defined in the <b>netinet/ip6.h</b> file.
	Determines the hop-by-hop options header to be used for outgoing IPv6 datagrams on this socket.	<b>Option Type:</b> <b>struct ip6_rthdr</b> defined in the <b>netinet/ip6.h</b> file.
	Determines the destination options header to be used for outgoing IPv6 datagrams on this socket. This header will follow a routing header (if present) and will also be used when there is no routing header specified.	<b>Option Type:</b> <b>struct ip6_dest</b> defined in the <b>netinet/ip6.h</b> file.
	Determines the destination options header to be used for outgoing IPv6 datagrams on this socket. This header will precede a routing header (if present). If no routing header is specified, this option will be silently ignored.	<b>Option Type:</b> <b>struct ip6_dest</b> defined in the <b>netinet/ip6.h</b> file.
	Determines how IPv6 path MTU discovery is being controlled for this socket.	<b>Option Type:</b> <b>int</b>
	Determines whether fragmentation of outgoing IPv6 packets has been disabled on this socket.	<b>Option Type:</b> <b>int</b> (Boolean interpretation)
	Determines whether IPV6_PATHMTU messages are being received as ancillary data on this socket.	<b>Option Type:</b> <b>int</b> (Boolean interpretation)
	Gets the address selection preferences for a socket.	<b>Option Type:</b> <b>int</b> (Boolean interpretation)
	Determines the current Path MTU for a connected socket.	<b>Option Type:</b> <b>struct ip6_mtuinfo</b> defined in the <b>netinet/in.h</b> file.

Item	Description	Value
<b>IPPROTO_ICMPV6</b>	Allows the user to filter ICMPV6 messages by the ICMPV6 type field. If no filter was set, the default kernel filter will be returned.	<b>Option Type:</b> The <b>icmp6_filter</b> structure defined in the <b>netinet/icmp6.h</b> file.

## Return Values

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

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

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

Upon successful completion of the **IPPROTO\_IP** option **IP\_PMTUAGE** the returns are:

With AIX Version 6.1:

- Positive non-zero OptionValue.

Upon successful completion of TCP protocol sockets option **IP\_FINDPMTU** the returns are:

With AIX Version 6.1:

- OptionValue 0 if PMTU discovery (tcp\_pmtu\_discover) is not enabled/not available.
- Positive non-zero OptionValue if PMTU is available.

## Error Codes

Item	Description
<b>EBADF</b>	The <i>Socket</i> parameter is not valid.
<b>EFAULT</b>	The address pointed to by the <i>OptionValue</i> parameter is not in a valid (writable) part of the process space, or the <i>OptionLength</i> parameter is not in a valid part of the process address space.
<b>EINVAL</b>	The <i>Level</i> , <i>OptionName</i> , or <i>OptionLength</i> is invalid.
<b>ENOBUF</b>	Insufficient resources are available in the system to complete the call.
<b>ENOTSOCK</b>	The <i>Socket</i> parameter refers to a file, not a socket.
<b>ENOPROTOPT</b>	The option is unknown.
<b>EOPNOTSUPP</b>	The option is not supported by the socket family or socket type.
<b>EPERM</b>	The user application does not have the permission to get or to set this socket option. Check the network tunable option.

## Examples

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

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

### Related reference

[bind Subroutine](#)

[shutdown Subroutine](#)

### Related information

[no subroutine](#)

[Sockets Overview](#)

## h

AIX runtime services beginning with the letter *h*.

## htonl Subroutine

### Purpose

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

### Library

ISODE Library (**libisode.a**)

### Syntax

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

```
uint32_t htonl ( HostLong)
uint32_t HostLong;
```

## Description

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

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

The **htonl** subroutine is defined in the **net/nh.h** file as a null macro if the host byte order is the same as the network byte order.

The **htonl** subroutine is declared in the **net/nh.h** file as a function if the host byte order is not the same as the network byte order.

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

## Parameters

Item	Description
<i>HostLong</i>	Specifies a 32-bit integer in host byte order.

## Return Values

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

## Related information

[Sockets Overview](#)

## htonll Subroutine

### Purpose

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

### Library

ISODE Library (**libisode.a**)

### Syntax

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

```
uint64_t htonll ( HostLong)
uint64_t HostLong;
```

## Description

The **htonll** subroutine converts an unsigned long (64-bit) integer from host byte order to Internet network byte order.

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

The **htonll** subroutine is defined in the **net/nh.h** file as a null macro if the host byte order is the same as the network byte order.

The **htonll** subroutine is declared in the **net/nh.h** file as a function if the host byte order is not the same as the network byte order.

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

## Parameters

Item	Description
<i>HostLong</i>	Specifies a 64-bit integer in host byte order.

## Return Values

The **htonll** subroutine returns a 64-bit integer in Internet network byte order (most significant byte first).

### Related information

[Sockets Overview](#)

## htons Subroutine

### Purpose

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

### Library

ISODE Library (**libisode.a**)

### Syntax

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

```
uint16_t htons ( HostShort)
```

```
uint16_t HostShort;
```

### Description

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

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

The **htons** subroutine is defined in the **net/nh.h** file as a null macro if the host byte order is the same as the network byte order.

The **htons** subroutine is declared in the **net/nh.h** file as a function if the host byte order is not the same as the network byte order.

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

## Parameters

Item	Description
<i>HostShort</i>	Specifies a 16-bit integer in host byte order that is a host address or port.

## Return Values

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

## Related information

[Sockets Overview](#)

## i

AIX runtime services beginning with the letter *i*.

## if\_freenameindex Subroutine

### Purpose

Frees the dynamic memory that was allocated by the [“if\\_nameindex Subroutine”](#) on page 161.

### Library

Library (**libc.a**)

### Syntax

```
#include <net/if.h>
```

```
void if_freenameindex (struct if_nameindex *ptr);
```

### Description

The *ptr* parameter is a pointer returned by the **if\_nameindex** subroutine. After the **if\_freenameindex** subroutine has been called, the application must not use the array of which *ptr* is the address.

## Parameters

Item	Description
<i>ptr</i>	Pointer returned by the <b>if_nameindex</b> subroutine

## Related information

[Subroutines Overview](#)

## if\_indextoname Subroutine

### Purpose

Maps an interface index into its corresponding name.

## Library

Standard C Library <libc.a>

## Syntax

```
#include <net/if.h>
char *if_indextoname(unsigned int ifindex, char *ifname);
```

## Description

When the **if\_indextoname** subroutine is called, the *ifname* parameter points to a buffer of at least IF\_NAMESIZE bytes. The **if\_indextoname** subroutine places the name of the interface in this buffer with the *ifindex* index.

**Note:** IF\_NAMESIZE is also defined in <net/if.h> and its value includes a terminating null byte at the end of the interface name.

If *ifindex* is an interface index, the **if\_indextoname** Subroutine returns the *ifname* value, which points to a buffer containing the interface name. Otherwise, it returns a NULL pointer and sets the **errno** global value to indicate the error.

If there is no interface corresponding to the specified index, the **errno** global value is set to **ENXIO**. If a system error occurs (such as insufficient memory), the **errno** global value is set to the proper value (such as, **ENOMEM**).

## Parameters

Item	Description
<i>ifindex</i>	Possible interface index
<i>ifname</i>	Possible name of an interface

## Error Codes

Item	Description
<b>ENXIO</b>	There is no interface corresponding to the specified index
<b>ENOMEM</b>	Insufficient memory

## Related information

[Subroutines Overview](#)

## if\_nameindex Subroutine

### Purpose

Retrieves index and name information for all interfaces.

## Library

The Standard C Library (<libc.a>)

## Syntax

```
#include <net/if.h>
struct if_nameindex *if_nameindex(void)
```

```
struct if_nameindex {
    unsigned int if_index; /* 1, 2, ... */
    char *if_name; /* null terminated name: "le0", ... */
};
```

## Description

The **if\_nameindex** subroutine returns an array of **if\_nameindex** structures (one per interface).

The memory used for this array of structures is obtained dynamically. The interface names pointed to by the *if\_name* members are obtained dynamically as well. This memory is freed by the **if\_freenameindex** subroutine.

The function returns a NULL pointer upon error, and sets the **errno** global value to the appropriate value. If successful, the function returns an array of structures. The end of an array of structures is indicated by a structure with an *if\_index* value of 0 and an *if\_name* value of NULL.

## Related information

[Subroutines Overview](#)

## if\_nametoindex Subroutine

### Purpose

Maps an interface name to its corresponding index.

### Library

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

### Syntax

```
#include <net/if.h>
unsigned int if_nametoindex(const char *ifname);
```

## Description

If the *ifname* parameter is the name of an interface, the **if\_nametoindex** subroutine returns the interface index corresponding to the *ifname* name. If the *ifname* parameter is not the name of an interface, the **if\_nametoindex** subroutine returns a 0 and the **errno** global variable is set to the appropriate value.

## Parameters

Item	Description
<i>ifname</i>	Possible name of an interface.

## Related information

[Subroutines Overview](#)

## inet\_ntop6\_zone Subroutine

### Purpose

Converts a binary IPv6 address with the possible zone ID into a text string that is suitable for presentation.

## Syntax

```
const char
inet_ntop6_zone (const void src, char dst, size_t size)
```

## Description

The **inet\_ntop6\_zone** subroutine is preferred over the **inet\_ntop** subroutine because it can infer the zone ID (defined in Section 11 of RFC 4007) that might be present in the **sin6\_scope\_id** field of the **sockaddr\_in6** structure.

Functionally, this subroutine uses the **inet\_ntop** subroutine to generate the textual representation of the address. It appends the **%zoneid** suffix to the string if the **sin6\_scope\_id** field is non-zero.

## Parameters

Item	Description
<i>src</i>	Specifies the <b>sockaddr_in6</b> structure that contains the address in the <b>sin6_addr</b> field and the zone ID in the <b>sin6_scope_id</b> field.
<i>dst</i>	Specifies a buffer where the textual representation of the address is stored, and if non-zero, the zone ID is stored.
<i>size</i>	Specifies the size (in bytes) of the buffer pointed to by the <b>dst</b> parameter.

## Return Values

If successful, a pointer to the buffer containing the converted address is returned. If unsuccessful, NULL is returned. Upon failure, the **errno** global variable is set to ENOSPC if the **size** parameter indicates that the destination buffer is small.

### Related information

[inet\\_ntop Subroutines](#)

## inet\_pton6\_zone Subroutine

### Purpose

Converts an IPv6 address in its standard text form which might include a zone ID suffix, into its numeric binary form.

## Syntax

```
int
inet_pton6_zone (const char *src, void *dst)
```

## Description

The **inet\_pton6\_zone** subroutine is preferred over the **inet\_pton** subroutine because it can infer the zone ID suffix (defined in Section 11 of RFC 4007) that might be present in the textual representation of an IPv6 address.

Functionally, this subroutine removes the zone ID, if present, and stores it in the **sin6\_scope\_id** field of the **sockaddr\_in6** structure pointed to by the **dst** parameter. It uses the **inet\_pton** subroutine to convert the removed address, and stores it in the **sin6\_addr** field.

## Parameters

Item	Description
<i>src</i>	The string that contains the textual representation of the address.
<i>dst</i>	A pointer to the <b>sockaddr_in6</b> structure where the numeric representation is stored. The zone ID, if present, is stored in the <b>sin6_scope_id</b> field, and the address is stored in the <b>sin6_addr</b> field.

## Return Values

If successful, one is returned. If the input is not a valid IPv6 address, zero is returned.

### Related information

[inet\\_pton Subroutine](#)

## inet6\_is\_srcaddr Subroutine

### Purpose

Verifies that a given local address meets address selection preferences.

### Library

Library (**libc.a**)

### Syntax

```
# include <netinet/in.h>
int inet6_is_srcaddr(struct sockaddr_in6 *srcaddr, uint32_t flags);
```

### Description

`inet6_is_src_addr` verifies that a local address corresponds to the set of address selection preference flags specified in `flags`.

The values of address selection preference flags are:

- `IPV6_PREFER_SRC_HOME`: prefer addresses reachable from a Home source address
- `IPV6_PREFER_SRC_COA`: prefer addresses reachable from a Care-of source address
- `IPV6_PREFER_SRC_TMP`: prefer addresses reachable from a temporary address
- `IPV6_PREFER_SRC_PUBLIC`: the prefer addresses reachable from a public source address
- `IPV6_PREFER_SRC_CGA`: the prefer addresses reachable from a Cryptographically Generated Address (CGA) source address
- `IPV6_PREFER_SRC_NONCGA`: the prefer addresses reachable from a non-CGA source address.

For example:

- To check if `srcaddr` is a Care-of address, `flags` must be set to `IPV6_PREFER_SRC_COA`.
- To check if `srcaddr` is a CGA and a public address, `flags` must be set to `IPV6_PREFER_SRC_CGA | IPV6_PREFER_SRC_PUBLIC`.

## Parameters

Item	Description
<b>srcaddr</b>	Points to a <code>sockaddr_in6</code> structure containing the source address to check

Item	Description
flags	Specifies the address selection preferences.

## Return Values

- The subroutine returns 1 when the given address corresponds to a local address and satisfies the address selection preferences.
- The subroutine returns -1 if the given address is not a local address or if flags does not specify one of the valid address selection flag value
- The subroutine returns 0 if the given address is a local address but does not satisfies the address selection preferences

## inet6\_opt\_append Subroutine

### Purpose

Returns the updated total length of the extension header.

### Syntax

```
int inet6_opt_append(void *extbuf, socklen_t extlen, int offset,
                    uint8_t type, socklen_t len, uint_t align,
                    void **databufp);
```

### Description

The **inet6\_opt\_append** subroutine returns the updated total length of the extension header, taking into account adding an option with length *len* and alignment *align*. If *extbuf* is not NULL, then, in addition to returning the length, the subroutine inserts any needed pad option, initializes the option (setting the type and length fields), and returns a pointer to the location for the option content in *databufp*. After **inet6\_opt\_append()** has been called, the application can use the *databuf* directly, or use **inet6\_opt\_set\_val()** to specify the content of the option.

### Parameters

Item	Description
<i>extbuf</i>	If NULL, <b>inet6_opt_append</b> will return only the updated length. If <i>extbuf</i> is not NULL, in addition to returning the length, the function inserts any needed pad option, initializes the option (setting the <b>type</b> and <b>length</b> fields) and returns a pointer to the location for the option content in <i>databufp</i> .
<i>extlen</i>	Size of the buffer pointed to by <i>extbuf</i> .
<i>offset</i>	The length returned by <b>inet6_opt_init()</b> or a previous <b>inet6_opt_append()</b> .
<i>type</i>	8-bit option type. Must have a value from 2 to 255, inclusive. (0 and 1 are reserved for the <i>Pad1</i> and <i>PadN</i> options, respectively.)
<i>len</i>	Length of the option data (excluding the option type and option length fields). Must be a value between 0 and 255, inclusive, and is the length of the option data that follows.

Item	Description
<i>align</i>	Alignment of the option data. Must be a value of 1, 2, 4, or 8. The <i>align</i> value can not exceed the value of <i>len</i> .
<i>databufp</i>	Specifies the content of the option.

## Return Values

Item	Description
-1	Option content does not fit in the extension header buffer.
integer value	Updated total length of the extension header.

## inet6\_opt\_find Subroutine

### Purpose

Looks for a specified option in the extension header.

### Syntax

```
int inet6_opt_find(void *extbuf, socklen_t extlen, int offset,
                 uint8_t *typep, socklen_t *lenp,
                 void **databufp);
```

### Description

The **inet6\_opt\_find** subroutine is similar to the **inet6\_opt\_next()** function, except this subroutine lets the caller specify the option type to be searched for, instead of always returning the next option in the extension header.

### Parameters

Item	Description
<i>extbuf</i>	Specifies the extension header.
<i>extlen</i>	Size of the buffer pointed to by <i>extbuf</i> .
<i>offset</i>	Specifies the position where scanning of the extension buffer can continue. Should either be 0 (for the first option) or the length returned by a previous call to <b>inet6_opt_next()</b> or <b>inet6_opt_find()</b> .
<i>typep</i>	Stores the option type.
<i>lenp</i>	Stores the length of the option data (excluding the option type and option length fields).
<i>databufp</i>	Points to the data field of the option.

### Return Values

The **inet6\_opt\_find** subroutine returns the updated "previous" total length computed by advancing past the option that was returned and past any options that did not match the type. This returned "previous"

length can then be passed to subsequent calls to **inet6\_opt\_find()** for finding the next occurrence of the same option type.

Item	Description
-1	The option cannot be located, there are no more options, or the option extension header is malformed.

## inet6\_opt\_finish Subroutine

### Purpose

Returns the final length of an extension header.

### Syntax

```
int inet6_opt_finish(void *extbuf, socklen_t extlen, int offset);
```

### Description

The **inet6\_opt\_finish** subroutine returns the final length of an extension header, taking into account the final padding of the extension header to make it a multiple of 8 bytes.

### Parameters

Item	Description
<i>extbuf</i>	If NULL, <b>inet6_opt_finish</b> will only return the final length. If <i>extbuf</i> is not NULL, in addition to returning the length, the function initializes the option by inserting a <i>Pad1</i> or <i>PadN</i> option of the proper length.
<i>extlen</i>	Size of the buffer pointed to by <i>extbuf</i> .
<i>offset</i>	The length returned by <b>inet6_opt_init()</b> or a previous <b>inet6_opt_append()</b> .

### Return Values

Item	Description
-1	The necessary pad does not fit in the extension header buffer.
integer value	Final length of the extension header.

## inet6\_opt\_get\_val Subroutine

### Purpose

Extracts data items of various sizes in the data portion of the option.

### Syntax

```
int inet6_opt_get_val(void *databuf, int offset, void *val,  
                    socklen_t vallen);
```

## Description

The `inet6_opt_get_val` subroutine extracts data items of various sizes in the data portion of the option. It is expected that each field is aligned on its natural boundaries, but the subroutine will not rely on the alignment.

## Parameters

Item	Description
<i>databuf</i>	Pointer to the data content returned by <code>inet6_opt_next()</code> or <code>inet6_opt_find()</code> .
<i>offset</i>	Specifies where in the data portion of the option the value should be extracted. The first byte after the option type and length is accessed by specifying an offset of 0.
<i>val</i>	Pointer to the destination for the extracted data.
<i>vallen</i>	Specifies the size of the data content to be extracted.

## Return Values

The `inet6_opt_get_val` subroutine returns the offset for the next field (that is, *offset* + *vallen*), which can be used when extracting option content with multiple fields.

## inet6\_opt\_init Subroutine

### Purpose

Returns the number of bytes needed for an empty extension header.

### Syntax

```
int inet6_opt_init(void *extbuf, socklen_t extlen);
```

## Description

The `inet6_opt_init` subroutine returns the number of bytes needed for the empty extension header (that is, a header without any options).

## Parameters

Item	Description
<i>extbuf</i>	Specifies NULL for an empty header. If <i>extbuf</i> is not NULL, it initializes the extension header to have the correct length field.
<i>extlen</i>	Specifies the size of the extension header. The value of <i>extlen</i> must be a positive value that is a multiple of 8.

## Return Values

Item	Description
-1	The value of <i>extlen</i> is not a positive (non-zero) multiple of 8.
integer value	Number of bytes needed for an empty extension header.

## inet6\_opt\_next Subroutine

Item	Description
-1	There are no more options or the option extension header is malformed.

### Purpose

Parses received option extension headers returning the next option.

### Syntax

```
int inet6_opt_next(void *extbuf, socklen_t extlen, int offset,
                  uint8_t *typep, socklen_t *lenp,
                  void **databufp);
```

### Description

The **inet6\_opt\_next** subroutine parses received option extension headers, returning the next option. The next option is returned by updating the *typep*, *lenp*, and *databufp* parameters.

### Parameters

Item	Description
<i>extbuf</i>	Specifies the extension header.
<i>extlen</i>	Size of the buffer pointed to by <i>extbuf</i> .
<i>offset</i>	Specifies the position where scanning of the extension buffer can continue. Should either be 0 (for the first option) or the length returned by a previous call to <b>inet6_opt_next()</b> or <b>inet6_opt_find()</b> .
<i>typep</i>	Stores the option type.
<i>lenp</i>	Stores the length of the option data (excluding the option type and option length fields).
<i>databufp</i>	Points to the data field of the option.

### Return Values

The **inet6\_opt\_next** subroutine returns the updated "previous" length computed by advancing past the option that was returned. This returned "previous" length can then be passed to subsequent calls to **inet6\_opt\_next()**. This function does not return any *PAD1* or *PADN* options.

## inet6\_opt\_set\_val Subroutine

### Purpose

Inserts data items into the data portion of an option.

### Syntax

```
int inet6_opt_set_val(void *databuf, int offset, void *val,
                     socklen_t vallen);
```

## Description

The **inet6\_opt\_set\_val** subroutine inserts data items of various sizes into the data portion of the option. The caller must ensure that each field is aligned on its natural boundaries. However, even when the alignment requirement is not satisfied, **inet6\_opt\_set\_val** will just copy the data as required.

## Parameters

Item	Description
<i>databuf</i>	Pointer to the data area returned by <b>inet6_opt_append()</b> .
<i>offset</i>	Specifies where in the data portion of the option the value should be inserted; the first byte after the option type and length is accessed by specifying an offset of 0.
<i>val</i>	Pointer to the data content to be inserted.
<i>vallen</i>	Specifies the size of the data content to be inserted.

## Return Values

The function returns the offset for the next field (that is, *offset* + *vallen*), which can be used when composing option content with multiple fields.

## inet6\_rth\_add Subroutine

### Purpose

Adds an IPv6 address to the end of the Routing header being constructed.

### Syntax

```
int inet6_rth_add(void *bp, const struct in6_addr *addr);
```

## Description

The **inet6\_rth\_add** subroutine adds the IPv6 address pointed to by *addr* to the end of the Routing header being constructed.

## Parameters

Item	Description
<i>bp</i>	Points to the buffer of the Routing header.
<i>addr</i>	Specifies which IPv6 address is to be added.

## Return Values

Item	Description
0	Success. The <i>segleft</i> member of the Routing Header is updated to account for the new address in the Routing header.
-1	The new address could not be added.

## inet6\_rth\_getaddr Subroutine

## Purpose

Returns a pointer to a specific IPv6 address in a Routing header.

## Syntax

```
struct in6_addr *inet6_rth_getaddr(const void *bp, int index);
```

## Description

The **inet6\_rth\_getaddr** subroutine returns a pointer to the IPv6 address specified by *index* in the Routing header described by *bp*. An application should first call **inet6\_rth\_segments()** to obtain the number of segments in the Routing header.

## Parameters

Item	Description
<i>bp</i>	Points to the Routing header.
<i>index</i>	Specifies the index of the IPv6 address that must be returned. The value of <i>index</i> must be between 0 and one less than the value returned by <b>inet6_rth_segments()</b> .

## Return Values

Item	Description
NULL	The <b>inet6_rth_getaddr</b> subroutine failed.
Valid pointer	Pointer to the address indexed by <i>index</i> .

## inet6\_rth\_init Subroutine

### Purpose

Initializes a buffer to contain a Routing header.

### Syntax

```
void *inet6_rth_init(void *bp, socklen_t bp_len, int type,  
int segments);
```

### Description

The **inet6\_rth\_init** subroutine initializes the buffer pointed to by *bp* to contain a Routing header of the specified *type* and sets **ip6r\_len** based on the *segments* parameter. *bp\_len* is only used to verify that the buffer is large enough. The **ip6r\_segleft** field is set to 0; **inet6\_rth\_add()** increments it.

When the application uses ancillary data, the application must initialize any **cmsghdr** fields. The caller must allocate the buffer, and the size of the buffer can be determined by calling **inet6\_rth\_space()**.

### Parameters

Item	Description
<i>bp</i>	Points to the buffer to be initialized.
<i>bp_len</i>	Size of the buffer pointed to by <i>bp</i> .

Item	Description
<i>type</i>	Specifies the type of Routing header to be held.
<i>segments</i>	Specifies the number of addresses within the Routing header.

## Return Values

Upon success, the return value is the pointer to the buffer (*bp*), and this is then used as the first argument to the `inet6_rth_add()` function.

Item	Description
NULL	The buffer could not be initialized.

## inet6\_rth\_reverse Subroutine

### Purpose

Writes a new Routing header that sends datagrams along the reverse route of a Routing header extension header.

### Syntax

```
int inet6_rth_reverse(const void *in, void *out);
```

### Description

The `inet6_rth_reverse` subroutine takes a Routing header extension header (pointed to by the first argument) and writes a new Routing header that sends datagrams along the reverse of that route. The function reverses the order of the addresses and sets the *segleft* member in the new Routing header to the number of segments. Both arguments are allowed to point to the same buffer (that is, the reversal can occur in place).

### Parameters

Item	Description
<i>in</i>	Points to the original Routing header extension header.
<i>out</i>	Points to the new Routing header route that reverses the route of <i>in</i> .

### Return Values

Item	Description
0	The reverse Routing header was successfully created.
-1	The reverse Routing header could not be created.

## inet6\_rth\_segments Subroutine

### Purpose

Returns the number of segments (addresses) contained in a Routing header.

## Syntax

```
int inet6_rth_segments(const void *bp);
```

## Description

The **inet6\_rth\_segments** subroutine returns the number of segments (addresses) contained in the Routing header described by *bp*.

## Parameters

Item	Description
<i>bp</i>	Points to the Routing header.

## Return Values

Item	Description
0 (or greater)	The number of addresses in the Routing header was returned.
-1	The number of addresses of the Routing header could not be returned.

## inet6\_rth\_space Subroutine

### Purpose

Returns the required number of bytes to hold a Routing header.

### Syntax

```
socklen_t inet6_rth_space(int type, int segments);
```

## Description

The **inet6\_rth\_space** subroutine returns the number of bytes required to hold a Routing header of the specified *type* containing the specified number of *segments* (addresses). For an IPv6 Type 0 Routing header, the number of segments must be between 0 and 127, inclusive. For an IPv6 Type 2 Routing Header, the number of segments must be 1. The return value is simply the space for the Routing header. When the application uses ancillary data, the application must pass the returned length to **CMSG\_SPACE()** in order to determine how much memory is needed for the ancillary data object (including the **cmsghdr** structure).

**Note:** Although **inet6\_rth\_space** returns the size of the ancillary data, it does not allocate the space required for the ancillary data. This allows an application to allocate a larger buffer, so that other ancillary data objects can be added, because all the ancillary data objects must be specified to **sendmsg()** as a single **msg\_control** buffer.

## Parameters

Item	Description
<i>type</i>	Specifies the type of Routing header to be held.
<i>segments</i>	Specifies the number of addresses within the Routing header.

## Return Values

Item	Description
0	Either the type of the Routing header is not supported by this implementation or the number of segments is invalid for this type of Routing header.
<b>length</b>	Determines how much memory is needed for the ancillary data object.

## inet\_addr Subroutine

### Purpose

Converts Internet addresses to Internet numbers.

### Library

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

### Syntax

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

```
in_addr_t inet_addr ( CharString )
register const char *CharString;
```

### Description

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

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

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

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

Sample return values for each subroutine are as follows:

Application String	inet_addr Returns	inet_network Returns
0x1234567890abcdef 0x1234567890abcdef. 256.257.258.259	0x090abcdef 0xFFFFFFFF (= -1) 0xFFFFFFFF (= -1)	0x000000ef 0x0000ef00 0x00010203

The ASCII string for the **inet\_addr** subroutine must conform to the following format:

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

**Note:**

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

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

The following describes the restrictions on the field values for the **inet\_addr** subroutine:

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

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

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

**Note:**

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

## Parameters

<b>Item</b>	<b>Description</b>
<i>CharString</i>	Represents a string of characters in the Internet address form.

## Return Values

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

The **inet\_addr** subroutine returns an error value of -1 for invalid strings.

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

## Files

Item	Description
<a href="#"><u>/etc/hosts</u></a>	Contains host names.
<a href="#"><u>/etc/networks</u></a>	Contains network names.

### Related reference

[gethostbyname Subroutine](#)

### Related information

[Sockets Overview](#)

[Understanding Network Address Translation](#)

## inet\_lnaof Subroutine

### Purpose

Returns the host ID of an Internet address.

### Library

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

### Syntax

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

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

### Description

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

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

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

### Parameters

Item	Description
<i>InternetAddr</i>	Specifies the Internet address to separate.

## Return Values

The return values of the **inet\_lnaof** subroutine depend on the class of Internet address the application provides:

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

## Files

Item	Description
<a href="#"><u>/etc/hosts</u></a>	Contains host names.

## Related information

[Sockets Overview](#)

[Understanding Network Address Translation](#)

## inet\_makeaddr Subroutine

### Purpose

Returns a structure containing an Internet Protocol address based on a network ID and host ID provided by the application.

### Library

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

### Syntax

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

```
struct in_addr inet_makeaddr ( Net, LocalNetAddr )
int Net, LocalNetAddr;
```

### Description

The **inet\_makeaddr** subroutine forms an Internet Protocol (IP) address from the network ID and Host ID provided by the application (as integer types). If the application provides a Class A network ID, the **inet\_makeaddr** subroutine forms the IP address using the net ID in the highest-order byte and the logical product of the host ID and 0x00FFFFFF in the 3 lowest-order bytes. If the application provides a Class B network ID, the **inet\_makeaddr** subroutine forms the IP address using the net ID in the two highest-order bytes and the logical product of the host ID and 0x0000FFFF in the lowest two ordered bytes. If the application does not provide either a Class A or Class B network ID, the **inet\_makeaddr** subroutine forms the IP address using the network ID in the 3 highest-order bytes and the logical product of the host ID and 0x0000FFFF in the lowest-ordered byte.

The **inet\_makeaddr** subroutine ensures that the IP address format conforms to network order, with the first byte representing the high-order byte. The **inet\_makeaddr** subroutine stores the IP address in the structure as an unsigned long value.

The application must verify that the network ID and host ID for the IP address conform to class A, B, or C. The **inet\_makeaddr** subroutine processes any nonconforming number as a Class C address.

The **inet\_makeaddr** subroutine expects the **in\_addr** structure to contain only the IP address field. If the application defines the **in\_addr** structure otherwise, then the value returned in **in\_addr** by the **inet\_makeaddr** subroutine is undefined.

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

## Parameters

Item	Description
<i>Net</i>	Contains an Internet network number.
<i>LocalNetAddr</i>	Contains a local network address.

## Return Values

Upon successful completion, the **inet\_makeaddr** subroutine returns a structure containing an IP address. If the **inet\_makeaddr** subroutine is unsuccessful, the subroutine returns a -1.

## Files

Item	Description
<u><a href="#">/etc/hosts</a></u>	Contains host names.

## Related information

[Sockets Overview](#)

[Understanding Network Address Translation](#)

## inet\_net\_ntop Subroutine

### Purpose

Converts between binary and text address formats.

### Library

Library (**libc.a**)

### Syntax

```
char *inet_net_ntop (af, src, bits, dst, size)
int af;
const void *src;
int bits;
char *dst;
size_t size;
```

### Description

This function converts a network address and the number of bits in the network part of the address into the CIDR format ascii text (for example, 9.3.149.0/24). The *af* parameter specifies the family of the address. The *src* parameter points to a buffer holding an IPv4 address if the *af* parameter is AF\_INET. The *bits* parameter is the size (in bits) of the buffer pointed to by the *src* parameter. The *dst* parameter points

to a buffer where the function stores the resulting text string. The *size* parameter is the size (in bytes) of the buffer pointed to by the *dst* parameter.

## Parameters

Item	Description
<i>af</i>	Specifies the family of the address.
<i>src</i>	Points to a buffer holding and IPv4 address if the <i>af</i> parameter is AF_INET.
<i>bits</i>	Specifies the size of the buffer pointed to by the <i>src</i> parameter.
<i>dst</i>	Points to a buffer where the resulting text string is stored.
<i>size</i>	Specifies the size of the buffer pointed to by the <i>dst</i> parameter.

## Return Values

If successful, a pointer to a buffer containing the text string is returned. If unsuccessful, NULL is returned. Upon failure, **errno** is set to EAFNOSUPPORT if the *af* parameter is invalid or ENOSPC if the size of the result buffer is inadequate.

### Related information

[Subroutines Overview](#)

## inet\_net\_pton Subroutine

### Purpose

Converts between text and binary address formats.

### Library

Library (**libc.a**)

### Syntax

```
int inet_net_pton (af, src, dst, size)
int af;
const char *src;
void *dst;
size_t size;
```

### Description

This function converts a network address in ascii into the binary network address. The ascii representation can be CIDR-based (for example, 9.3.149.0/24) or class-based (for example, 9.3.149.0). The *af* parameter specifies the family of the address. The *src* parameter points to the string being passed in. The *dst* parameter points to a buffer where the function will store the resulting numeric address. The *size* parameter is the size (in bytes) of the buffer pointed to by the *dst* parameter.

## Parameters

Item	Description
<i>af</i>	Specifies the family of the address.
<i>src</i>	Points to the string being passed in.
<i>dst</i>	Points to a buffer where the resulting numeric address is stored.

Item	Description
<i>size</i>	Specifies the size (in bytes) of the buffer pointed to by the <i>dst</i> parameter.

## Return Values

If successful, the number of bits, either inputted classfully or specified with */CIDR*, is returned. If unsuccessful, a -1 (negative one) is returned (check *errno*). *ENOENT* means it was not a valid network specification.

## Related information

[Subroutines Overview](#)

## inet\_netof Subroutine

### Purpose

Returns the network id of the given Internet address.

### Library

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

### Syntax

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

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

### Description

The **inet\_netof** subroutine returns the network number from the specified Internet address number typed as unsigned long value. The **inet\_netof** subroutine masks off the network number and the host number from the Internet address based on the Internet address class.

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

**Note:** The application assumes responsibility for verifying that the network number and the host number for the Internet address conforms to a class A or B or C Internet address. The **inet\_netof** subroutine processes any other number as a class C address.

### Parameters

Item	Description
<u><i>InternetAddr</i></u>	Specifies the Internet address to separate.

### Return Values

Upon successful completion, the **inet\_netof** subroutine returns a network number from the specified long value representing the Internet address. If the application gives a class A Internet address, the **inet\_lnoaf** subroutine returns the logical product of the Internet address and 0xFF000000. If the application gives a class B Internet address, the **inet\_lnoaf** subroutine returns the logical product of the Internet address

and 0xFFFF0000 . If the application does not give a class A or B Internet address, the **inet\_lnoaf** subroutine returns the logical product of the Internet address and 0FFFFFF00 .

## Files

Item	Description
<a href="#"><u>/etc/hosts</u></a>	Contains host names.
<a href="#"><u>/etc/networks</u></a>	Contains network names.

## Related information

[Sockets Overview](#)

[Understanding Network Address Translation](#)

## inet\_network Subroutine

### Purpose

Converts an ASCII string containing an Internet network addressee in . (dot) notation to an Internet address number.

### Library

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

### Syntax

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

```
in_addr_t inet_network ( CharString)
register const char *CharString;
```

### Description

The **inet\_network** subroutine converts an ASCII string containing a valid Internet address using . (dot) notation (such as, 120 . 121 . 122 . 123) to an Internet address number formatted as an unsigned integer value. The **inet\_network** subroutine returns an error value if the application does not provide an ASCII string containing a valid Internet address using . notation.

The input ASCII string must represent a valid Internet address number, as described in "[TCP/IP addressing](#)" in *Networks and communication management*. The input string must be terminated with a null terminator (0x00) or a space (0x30). The **inet\_network** subroutine ignores characters that follow the terminating character.

The input string can express an Internet address number in decimal, hexadecimal, or octal format. In hexadecimal format, the string must begin with 0x. The string must begin with 0 to indicate octal format. In decimal format, the string requires no prefix.

Each octet of the input string must be delimited from another by a period. The application can omit values between delimiters. The **inet\_network** subroutine interprets missing values as 0.

The following examples show valid strings and their output values in both decimal and hexadecimal notation:

Examples of valid strings		
Input String	Output Value (in decimal)	Output Value (in hex)
...1	1	0x00000001
.1..	65536	0x00010000
1	1	0x1
0xFFFFFFFF	255	0x000000FF
1.	256	0x100
1.2.3.4	66048	0x010200
0x01.0x2.03.004	16909060	0x01020304
1.2. 3.4	16777218	0x01000002
9999.1.1.1	251724033	0x0F010101

The following examples show invalid input strings and the reasons they are not valid:

Examples of invalid strings	
Input String	Reason
1.2.3.4.5	Excessive fields.
1.2.3.4.	Excessive delimiters (and therefore fields).
1,2	Bad delimiter.
1p	String not terminated by null terminator nor space.
{empty string}	No field or delimiter present.

Typically, the value of each octet of an Internet address cannot exceed 246. The **inet\_network** subroutine can accept larger values, but it uses only the eight least significant bits for each field value. For example, if an application passes 0x1234567890.0xabcdef, the **inet\_network** subroutine returns 37103 (0x000090EF).

The application must verify that the network ID and host ID for the Internet address conform to class A, class B, or class C. The **inet\_makeaddr** subroutine processes any nonconforming number as a class C address.

The **inet\_network** subroutine does not check the pointer to the ASCII input string. The application must verify the validity of the address of the string.

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

## Parameters

Item	Description
<i>CharString</i>	Represents a string of characters in the Internet address form.

## Return Values

For valid input strings, the **inet\_network** subroutine returns an unsigned integer value that comprises the bit patterns of the input fields concatenated together. The **inet\_network** subroutine places the first pattern in the leftmost (most significant) position and appends subsequent patterns if they exist.

For invalid input strings, the **inet\_network** subroutine returns a value of -1.

## Files

Item	Description
<a href="#">/etc/hosts</a>	Contains host names.
<a href="#">/etc/networks</a>	Contains network names.

## Related information

[Sockets Overview](#)

[Understanding Network Address Translation](#)

## inet\_ntoa Subroutine

### Purpose

Converts an Internet address into an ASCII string.

### Library

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

### Syntax

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

```
char *inet_ntoa ( InternetAddr )
struct in_addr InternetAddr;
```

### Description

The **inet\_ntoa** subroutine takes an Internet address and returns an ASCII string representing the Internet address in dot notation. All Internet addresses are returned in network order, with the first byte being the high-order byte.

Use C language integers when specifying each part of a dot notation.

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

### Parameters

Item	Description
<i>InternetAddr</i>	Contains the Internet address to be converted to ASCII.

### Return Values

Upon successful completion, the **inet\_ntoa** subroutine returns an Internet address.

If the **inet\_ntoa** subroutine is unsuccessful, the subroutine returns a -1.

## Files

Item	Description
<a href="#">/etc/hosts</a>	Contains host names.
<a href="#">/etc/networks</a>	Contains network names.

## Related information

[Sockets Overview](#)

[Understanding Network Address Translation](#)

## inet\_ntop Subroutine

### Purpose

This function is deprecated for AF\_INET6 in favor of the [inet\\_ntop6\\_zone Subroutine](#).

### Library

Library (**libc.a**)

### Syntax

```
const char *inet_ntop (af, src, dst, size)
int af;
const void *src;
char *dst;
size_t size;
```

### Description

This function converts from an address in binary format (as specified by the *src* parameter) to standard text format, and places the result in the *dst* parameter (if *size*, which specifies the space available in the *dst* parameter, is sufficient). The *af* parameter specifies the family of the address. This can be AF\_INET or AF\_INET6.

The *src* parameter points to a buffer holding an IPv4 address if the *af* parameter is AF\_INET, or an IPv6 address if the *af* parameter is AF\_INET6. The *dst* parameter points to a buffer where the function will store the resulting text string. The *size* parameter specifies the size of this buffer (in bytes). The application must specify a non-NULL *dst* parameter. For IPv6 addresses, the buffer must be at least INET6\_ADDRSTRLEN bytes. For IPv4 addresses, the buffer must be at least INET\_ADDRSTRLEN bytes.

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 the `<netinet/in.h>` library:

```
#define INET_ADDRSTRLEN 16
#define INET6_ADDRSTRLEN 46
```

### Parameters

Item	Description
<i>af</i>	Specifies the family of the address. This can be AF_INET or AF_INET6.
<i>src</i>	Points to a buffer holding an IPv4 address if the <i>af</i> parameter is set to AF_INET, or an IPv6 address if the <i>af</i> parameter is set to AF_INET6.
<i>dst</i>	Points to a buffer where the resulting text string is stored.
<i>size</i>	Specifies the size (in bytes) of the buffer pointed to by the <i>dst</i> parameter.

## Return Values

If successful, a pointer to the buffer containing the converted address is returned. If unsuccessful, NULL is returned. Upon failure, the **errno** global variable is set to EAFNOSUPPORT if the specified address family (*af*) is unsupported, or to ENOSPC if the *size* parameter indicates the destination buffer is too small.

## Related information

[Subroutines Overview](#)

## inet\_pton Subroutine

### Purpose

This function is deprecated for AF\_INET6 in favor of the [inet\\_pton6\\_zone Subroutine](#).

### Library

Library (**libc.a**)

### Syntax

```
int inet_pton (af, src, dst)
int af;
const char *src;
void *dst;
```

### Description

This function converts an address in its standard text format into its numeric binary form. The *af* parameter specifies the family of the address.

**Note:** Only the AF\_INET and AF\_INET6 address families are supported.

### Parameters

Item	Description
<i>af</i>	Specifies the family of the address. This can be AF_INET or AF_INET6.
<i>src</i>	Points to the string being passed in.
<i>dst</i>	Points to a buffer where the function stores the numeric address. The address is returned in network byte order.

### Return Values

If successful, one is returned. If unsuccessful, zero is returned if the input is not a valid IPv4 dotted-decimal string or a valid IPv6 address string; or a negative one with the **errno** global variable set to EAFNOSUPPORT if the *af* parameter is unknown. The calling application must ensure that the buffer referred to by the *dst* parameter is large enough to hold the numeric address (4 bytes for AF\_INET or 16 bytes for AF\_INET6).

If the *af* parameter is AF\_INET, the function accepts a string in the standard IPv4 dotted-decimal form.

```
ddd.ddd.ddd.ddd
```

Where *ddd* is a one to three digit decimal number between 0 and 255.

**Note:** Many implementations of the existing **inet\_addr** and **inet\_aton** functions accept nonstandard input such as octal numbers, hexadecimal numbers, and fewer than four numbers. **inet\_pton** does not accept these formats.

If the *af* parameter is AF\_INET6, then the function accepts a string in one of the standard IPv6 text forms defined in the addressing architecture specification.

### Related information

[Subroutines Overview](#)

## inetgr, getnetgrent, setnetgrent, or endnetgrent Subroutine

### Purpose

Handles the group network entries.

### Library

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

### Syntax

```
#include <netdb.h>
```

```
inetgr (NetGroup, Machine, User, Domain)  
char * NetGroup, * Machine, * User, * Domain;
```

```
getnetgrent (MachinePointer, UserPointer, DomainPointer)  
char ** MachinePointer, ** UserPointer, ** DomainPointer;
```

```
void setnetgrent (NetGroup)  
char *NetGroup
```

```
void endnetgrent ()
```

### Description

The **inetgr** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **inetgr** subroutine returns *1* or *0*, depending on if **netgroup** contains the *machine, user, domain* triple as a member. Any of these three strings; *machine, user, or domain*, can be NULL, in which case it signifies a wild card.

The **getnetgrent** subroutine returns the next member of a network group. After the call, *machinepointer* will contain a pointer to a string containing the name of the machine part of the network group member, and similarly for *userpointer* and *domainpointer*. If any of *machinepointer, userpointer, or domainpointer* is returned as a NULL pointer, it signifies a wild card. The **getnetgrent** subroutine uses malloc to allocate space for the name. This space is released when the **endnetgrent** subroutine is called. **getnetgrent** returns *1* if it succeeded in obtaining another member of the network group or *0* when it has reached the end of the group.

The **setnetgrent** subroutine establishes the network group from which the **getnetgrent** subroutine will obtain members, and also restarts calls to the **getnetgrent** subroutine from the beginning of the list. If the previous **setnetgrent()** call was to a different network group, an **endnetgrent()** call is implied. **endnetgrent()** frees the space allocated during the **getnetgrent()** calls.

### Parameters

Item	Description
<i>Domain</i>	Specifies the domain.
<i>DomainPointer</i>	Points to the string containing <i>Domain</i> part of the network group.

Item	Description
<i>Machine</i>	Specifies the machine.
<i>MachinePointer</i>	Points to the string containing <i>Machine</i> part of the network group.
<i>NetGroup</i>	Points to a network group.
<i>User</i>	Specifies a user.
<i>UserPointer</i>	Points to the string containing <i>User</i> part of the network group.

## Return Values

### Item Description

- 1** Indicates that the subroutine was successful in obtaining a member.
- 0** Indicates that the subroutine was not successful in obtaining a member.

## Files

Item	Description
<i>/etc/netgroup</i>	Contains network groups recognized by the system.
<i>/usr/include/netdb.h</i>	Contains the network database structures.

## Related information

[Sockets Overview](#)

## ioctl Socket Control Operations

### Purpose

Performs network-related control operations.

### Syntax

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

int ioctl (fd, cmd, .../* arg */)
int fd;
int cmd;
int ... /* arg */
```

### Description

The socket `ioctl` commands does various network-related control. The *fd* argument is a socket descriptor. For non-socket descriptors, the functions that are performed by this call are unspecified.

The *cmd* argument and an optional third argument (with varying type) are passed to and interpreted by the socket `ioctl` function to perform an appropriate control operation that is specified by the user.

The socket `ioctl` control operations can be in the following control operations categories:

- [Socket](#)
- [Routing table](#)
- [ARP table](#)
- [Global network parameters](#)

- [Interface](#)

## Parameters

Item	Description
<i>fd</i>	Open file descriptor that refers to a socket created by using <code>socket</code> or <code>accept</code> calls.
<i>cmd</i>	Selects the control function to be performed.
<i>.../* arg */</i>	Represents information that is required for the requested function. The type of <i>arg</i> depends on the particular control request, but it is either an integer or a pointer to a socket-specific data structure.

## Socket Control Operations

The following `ioctl` commands operate on sockets:

ioctl command	Description
SIOCATMARK	<p>Determines whether the read pointer is pointing to the logical mark in the DataStream. The logical mark indicates the point at which the out-of-band data is sent.</p> <pre>ioctl(fd, SIOCATMARK, &amp;atmark); int atmark;</pre> <p>If <i>atmark</i> is set to 1 on return, the read pointer points to the mark and the next read returns data after the mark. If <i>atmark</i> is set to 0 on return (assuming out-of-band data is present on the DataStream), the next read returns data that is sent before the out-of-band mark.</p> <p><b>Note:</b> The out-of-band data is a logically independent data channel that is delivered to the user independently of normal data; in addition, a signal is also sent because of the immediate attention required. Ctrl-C characters are an example.</p>
SIOCSPGRP SIOCGPGRP	<p>SIOCSPGRP sets the process group information for a socket. SIOCGPGRP gets the process group ID associated with a socket.</p> <pre>ioctl (fd, cmd, (int)&amp;pgrp); int pgrp;</pre> <p><b>cmd</b> Set to SIOCSPGRP or SIOCGPGRP.</p> <p><b>pgrp</b> Specifies the process group ID for the socket.</p>

## Routing Table Control Operations

The following `ioctl` commands operate on the kernel routing table:

<b>ioctl command</b>	<b>Description</b>
SIOCADDRT SIOCDELRT	<p>SIOCADDRT adds a route entry in the routing table. SIOCDELRT deletes a route entry from the routing table.</p> <pre>ioctl(fd, cmd, (caddr_t)&amp;route); struct rtable route;</pre> <p><b>cmd</b> Set to SIOCADDRT or SIOCDELRT. The route entry information is passed in the <code>rtable</code> structure.</p>
SIOUPDRUTE	<p>Updates the routing table by using the information that is passed in the <code>ifreq</code> structure.</p> <pre>ioctl(fd, SIOUPDRUTE, (caddr_t)&amp;ifr); struct ifreq ifr;</pre>

## ARP Table Control Operations

The following ioctl commands operate on the kernel ARP table. The `net/if_arp.h` header file must be included.

<b>ioctl command</b>	<b>Description</b>
SIOCSARP SIOCDEARP SIOCGARP	<p>SIOCSARP adds or modifies an ARP entry in the ARP table. SIOCDEARP deletes an ARP entry from the ARP table. SIOCGARP gets an ARP entry from the ARP table.</p> <pre>ioctl(fd, cmd, (caddr_t)&amp;ar); struct arpreq ar;</pre> <p><b>cmd</b> Set to SIOCSARP, SIOCDEARP, or SIOCGARP. The ARP entry information is passed in the <code>arpreq</code> structure. If <code>ar.if Type = IFT_IB</code> and the command is SIOCDEARP, the InfiniBand (IB) ARP entry is deleted.</p>

## Global Network Parameters Control Operations

The following ioctl commands operate as global network parameters:

ioctl command	Description
SIOCSNETOPT SIOCGNETOPT SIOCNETOPT SIOCGNETOPT1	<p>SIOCSNETOPT sets the value of a network option. SIOCGNETOPT gets the value of a network option. SIOCNETOPT sets the default values of a network option.</p> <pre data-bbox="646 306 1040 359">ioctl(fd, cmd, (caddr_t)&amp;oreq); struct optreq oreq;</pre> <p><b>cmd</b> Set to SIOCSNETOPT, SIOCGNETOPT, or SIOCNETOPT.</p> <p>The network option value is stored in the optreq structure.</p> <p>SIOCGNETOPT1 gets the current value, default value, and the range of a network option.</p> <pre data-bbox="646 600 1154 653">ioctl(fd, SIOCGNETOPT1, (caddr_t)&amp;oreq); struct optreq1 oreq;</pre> <p>The network option information is stored in the optreq1 structure upon return. The optreq and optreq1 structures are defined in net/netopt.h.</p>
SIOCGNMTUS SIOCGETMTUS SIOCADDMTU SIOCDELMTU	<p>SIOCGNMTUS gets the number of MTUs maintained in the list of common MTUs. SIOCADDMTU adds an MTU in the list of common MTUs. SIOCDELMTU deletes an MTU from the list of common MTUs.</p> <pre data-bbox="646 919 1052 972">ioctl(fd, cmd, (caddr_t)&amp;mntus); int mntus;</pre> <p><b>cmd</b> Set to SIOCGNMTUS, SIOCADDMTU, or SIOCDELMTU.</p> <p>SIOCGETMTUS gets the MTUs maintained in the list of common MTUs.</p> <pre data-bbox="646 1129 1117 1182">ioctl(fd, SIOCGETMTUS, (caddr_t)&amp;gm); struct get_mtus gm;</pre> <p>The get_mtus structure is defined in netinet/in.h.</p>

## Interface Control Operations

The following ioctl commands operate on interfaces. The net/if.h header file must be included.

ioctl command	Description
SIOCSIFADDR SIOCIFADDR	<p>SIOCSIFADDR sets an interface address. SIOCIFADDR deletes an interface address. The interface address is specified in the <i>ifr.ifr_addr</i> field. SIOCIFADDR gets an interface address. The address is returned in the <i>ifr.ifr_addr</i> field.</p> <pre data-bbox="659 1608 1325 1661">ioctl(fd, cmd, (caddr_t)&amp;ifr, sizeof(struct ifreq)); struct ifreq ifr;</pre> <p><b>cmd</b> Set to SIOCSIFADDR, or SIOCIFADDR.</p>

ioctl command	Description
SIOCAIFADDR	<p>SIOCAIFADDR adds an interface address. The interface name is specified in the <i>ifr.ifra_name</i> field. The alias IP address is specified in the <i>theifr.ifra_addr</i> field. The alias IP broadcast address might be specified in the <i>ifr.ifra_broadaddr</i> field, and the alias IP network mask might be specified in the <i>ifr.ifra_mask</i>.</p> <pre data-bbox="662 373 1386 422">ioctl(fd, cmd, (caddr_t)&amp;ifr, sizeof(struct ifaliasreq)); struct ifaliasreq ifr;</pre> <p><b>cmd</b> Set to SIOCAIFADDR</p>
SIOCGIFADDRS	<p>Gets the list of addresses that are associated with an interface.</p> <pre data-bbox="662 596 1219 644">ioctl (fd, SIOCGIFADDRS, (caddr_t)ifaddrsp); struct ifreqaddrs *ifaddrsp;</pre> <p>The interface name is passed in the <i>ifaddrsp-&gt;ifr_name</i> field. The addresses that are associated with the interface are stored in <i>ifaddrsp-&gt;ifrasu</i> array on return.</p> <p><b>Note:</b> The <i>ifreqaddrs</i> structure contains space for storing only one <i>sockaddr_in/sockaddr_in6</i> structure (array of one <i>sockaddr_in/sockaddr_in6</i> element). To get <i>n</i> addresses associated with an interface, the caller of the <i>ioctl</i> command must allocate space for <math>\{sizeof(struct ifreqaddrs) + (n * sizeof(struct sockaddr_in))\}</math> bytes.</p>
SIOCSIFDSTADDR SIOCGIFDSTADDR	<p>SIOCSIFDSTADDR sets the point-to-point address for an interface that is specified in the <i>ifr.ifr_dstaddr</i> field. SIOCGIFDSTADDR gets the point-to-point address that is associated with an interface. The address is stored in the <i>ifr.ifr_dstaddr</i> field on return.</p> <pre data-bbox="662 1152 1321 1201">ioctl(fd, cmd, (caddr_t)&amp;ifr, sizeof(struct ifreq)); struct ifreq ifr;</pre> <p><b>cmd</b> Set to SIOCSIFDSTADDR or SIOCGIFDSTADDR.</p>
SIOCSIFNETMASK SIOCGIFNETMASK	<p>SIOCSIFNETMASK sets the interface netmask that is specified in the <i>ifr.ifr_addr</i> field. SIOCGIFNETMASK gets the interface netmask.</p> <pre data-bbox="662 1409 1321 1457">ioctl(fd, cmd, (caddr_t)&amp;ifr, sizeof(struct ifreq)); struct ifreq ifr;</pre> <p><b>cmd</b> Set to SIOCSIFNETMASK or SIOCGIFNETMASK.</p>
SIOCSIFBRDADDR SIOCGIFBRDADDR	<p>SIOCSIFBRDADDR sets the interface broadcast address that is specified in the <i>ifr.ifr_broadaddr</i> field. SIOCGIFBRDADDR gets the interface broadcast address. The broadcast address is placed in the <i>ifr.ifr_broadaddr</i> field.</p> <pre data-bbox="662 1724 1321 1772">ioctl(fd, cmd, (caddr_t)&amp;ifr, sizeof(struct ifreq)); struct ifreq ifr;</pre> <p><b>cmd</b> Set to SIOCSIFBRDADDR or SIOCGIFBRDADDR.</p>

ioctl command	Description
SIOCGSIZIFCONF	<p>Gets the size of memory that is required to get configuration information for all interfaces returned by SIOCGIFCONF.</p> <pre data-bbox="662 275 1133 327">ioctl(fd, cmd, (caddr_t)&amp;ifconfsize); int ifconfsize;</pre>
SIOCGIFCONF	<p>Returns configuration information for all the interfaces that are configured on the system.</p> <pre data-bbox="662 453 1146 506">ioctl(fd, SIOCGIFCONF, (caddr_t)&amp;ifc); struct ifconf ifc;</pre> <p>The configuration information is returned in a list of <code>ifreq</code> structures pointed to by the <code>ifc.ifc_req</code> field, with one <code>ifreq</code> structure per interface.</p> <p><b>Note:</b> The caller of the <code>ioctl</code> command must allocate sufficient space to store the configuration information, returned as a list of <code>ifreq</code> structures for all of the interfaces that are configured on the system. For example, if <math>n</math> interfaces are configured on the system, <code>ifc.ifc_req</code> must point to <math>\{n * \text{sizeof}(\text{struct ifreq})\}</math> bytes of space allocated.</p> <p><b>Note:</b> Alternatively, the SIOCGSIZIFCONF <code>ioctl</code> command can be used for this purpose.</p>
SIOCSIFFLAGS SIOCGIFFLAGS	<p>SIOCSIFFLAGS sets the interface flags. SIOCGIFFLAGS gets the interface flags.</p> <pre data-bbox="662 995 1044 1047">ioctl(fd, cmd, (caddr_t)&amp;ifr); struct ifreq ifr;</pre> <p>Refer to <code>/usr/include/net/if.h</code> for the interface flags, denoted by <code>IFF_XXX</code>.</p> <p><b>Note:</b> The <code>IFF_BROADCAST</code>, <code>IFF_POINTTOPPOINT</code>, <code>IFF_SIMPLEX</code>, <code>IFF_RUNNING</code>, <code>IFF_OACTIVE</code>, and <code>IFF_MULTICAST</code> flags cannot be changed by using <code>ioctl</code>.</p>
SIOCSIFMETRIC SIOCGIFMETRIC	<p>SIOCSIFMETRIC sets the interface metric that is specified in the <code>ifr.ifr_metric</code> field. SIOCGIFMETRIC gets the interface metric. The interface metric is placed in the <code>ifr.ifr_metric</code> field on return.</p> <pre data-bbox="662 1394 1044 1446">ioctl(fd, cmd, (caddr_t)&amp;ifr); struct ifreq ifr;</pre> <p><b>cmd</b> Set to SIOCSIFMETRIC or SIOCGIFMETRIC.</p>
SIOCSIFSUBCHAN SIOCGIFSUBCHAN	<p>SIOCSIFSUBCHAN sets the subchannel address that is specified in the <code>ifr.ifr_flags</code> field. SIOCGIFSUBCHAN gets the subchannel address in the <code>ifr.ifr_flags</code> field.</p> <pre data-bbox="662 1682 1183 1734">ioctl(fd, SIOCSIFSUBCHAN, (caddr_t)&amp;ifr); struct ifreq ifr;</pre>

ioctl command	Description
SIOCSIFOPTIONS SIOCGETIFOPTIONS	<p>SIOCSIFOPTIONS sets the interface options. SIOCGETIFOPTIONS gets the interface options.</p> <pre data-bbox="662 275 1182 327">ioctl(fd, SIOCSIFOPTIONS, (caddr_t)&amp;ifr); struct ifreq ifr;</pre> <p>The interface options are stored in the <i>ifr_flags</i> field of the <i>ifreq</i> structure. Refer to <code>/usr/include/net/if.h</code> file for the list of interface options that are denoted by <code>IFO_XXX</code>.</p>

ioctl command	Description
SIOCADDMULTI SIOCDELMULTI	<p>SIOCADDMULTI adds an address to the list of multicast addresses for an interface. SIOCDELMULTI deletes a multicast address from the list of multicast addresses for an interface.</p> <pre data-bbox="662 663 1040 716">ioctl(fd, cmd, (caddr_t)&amp;ifr); struct ifreq ifr;</pre> <p><b>cmd</b> Set to SIOCADDMULTI or SIOCDELMULTI.</p> <p>The multicast address information is specified in the <i>ifr_addr</i> structure.</p>
SIOCGETVIFCNT	<p>Gets the packet count information for a virtual interface. The information is specified in the <i>sioc_vif_req</i> structure.</p> <pre data-bbox="662 989 1208 1041">ioctl(fd, SIOCGETVIFCNT, (caddr_t)&amp;v_req); struct sioc_vif_req v_req;</pre>
SIOCGETSGCNT	<p>Gets the packet count information for the source group specified. The information is stored in the <i>sioc_sg_req</i> structure on return.</p> <pre data-bbox="662 1167 1182 1220">ioctl(fd, SIOCGETSGCNT, (caddr_t)&amp;v_req); struct sioc_sg_req v_req;</pre>
SIOCSIFMTU SIOCGETIFMTU	<p>SIOCSIFMTU sets the interface maximum transmission unit (MTU). SIOCGETIFMTU gets the interface MTU.</p> <pre data-bbox="662 1346 1040 1398">ioctl(fd, cmd, (caddr_t)&amp;ifr); struct ifreq ifr;</pre> <p>The MTU value is stored in <i>ifr_ifr_mtu</i> field.</p> <p><b>Note:</b> The range of valid values for MTU varies for an interface and is dependent on the interface type.</p>
SIOCIFATTACH SIOCIFDETACH	<p>SIOCIFATTACH attaches an interface. This initializes and adds an interface in the network interface list. SIOCIFDETACH detaches an interface broadcast address. This removes the interface from the network interface list. The interface name is specified in the <i>ifr_ifr_name</i> field.</p> <pre data-bbox="662 1745 1040 1797">ioctl(fd, cmd, (caddr_t)&amp;ifr); struct ifreq ifr;</pre>

ioctl command	Description
SIOCSIFGIDLIST SIOCGIFGIDLIST	<p>SIOCSIFGIDLIST adds or deletes the list of group IDs specified in the <i>ifrg.ifrg_gidlist</i> field to the <i>gidlist</i> interface. The interface name is specified in the <i>ifrg.ifrg_name</i> field. An operation code, ADD_GRP/DEL_GRP, specified in the <i>ifrg.ifrg_gidlist</i> field indicates whether the specified list of group IDs must be added to or deleted from the <i>gidlist</i> interface. SIOCGIFGIDLIST gets the list of group IDs associated with an interface. The group IDs are placed in the <i>ifrg.ifrg_gidlist</i> field on return.</p> <pre data-bbox="662 468 1057 516">ioctl(fd, cmd, (caddr_t)&amp;ifrg); struct ifgidreq ifrg;</pre>
SIOCIF_ATM_UBR SIOCIF_ATM_SNMPARP SIOCIF_ATM_DUMPARP SIOCIF_ATM_IDLE SIOCIF_ATM_SVC SIOCIF_ATM_DARP SIOCIF_ATM_GARP SIOCIF_ATM_SARP	<p>SIOCIF_ATM_UBR sets the UBR rate for an ATM interface. SIOCIF_ATM_SNMPARP gets the SNMP ATM ARP entries. SIOCIF_ATM_DUMPARP gets the specified number of ATM ARP entries. SIOCIF_ATM_DARP deletes an ATM ARP entry from the ARP table. SIOCIF_ATM_GARP gets an ATM ARP entry to the ARP table. SIOCIF_ATM_SARP adds an ATM ARP entry. The ARP information is specified in the <i>atm_arpreq</i> structure. SIOCIF_ATM_SVC specifies whether this interface supports Permanent Virtual Circuit (PVC) and Switched Virtual Circuit (SVC) types of virtual connections. It also specifies whether this interface is an ARP client or an ARP server for this Logical IP Subnetwork (LIS) based on the flag that is set in the <i>ifatm_svc_arg</i> structure. SIOCIF_ATM_IDLE specifies the idle time limit on the interface.</p>
SIOCSISNO SIOCGISNO	<p>SIOCSISNO sets interface specific network options for an interface. SIOCGISNO gets interface specific network options that are associated with an interface.</p> <pre data-bbox="662 1108 1040 1157">ioctl(fd, cmd, (caddr_t)&amp;ifr); struct ifreq ifr;</pre> <p><b>cmd</b> Set to SIOCSISNO or SIOCGISNO.</p> <p>The interface-specific network options are stored in <i>ifr.ifr_isno</i> structure. Refer to <code>/usr/include/net/if.h</code> file for the list of interface-specific network options that are denoted by ISNO_XXX.</p>
SIOCGIFBAUDRATE	<p>Gets the value of the interface baud rate in the <i>ifr_baudrate</i> field.</p> <pre data-bbox="662 1434 1195 1482">ioctl(fd, SIOCGIFBAUDRATE, (caddr_t)&amp;ifr); struct ifreq ifr;</pre> <p>The baud rate is stored in the <i>ifr.ifr_baudrate</i> field.</p>

ioctl command	Description
SIOCADDIFVIPA SIOCDELIFVIPA SIOCLISTIFVIPA	<p>SIOCADDIFVIPA associates the specified list of interfaces pointed to by <i>ifrv.ifrv_ifname</i> with the virtual interface specified by <i>ifrv.ifrv_name</i>. This operation causes the source address for all outgoing packets on these interfaces to be set to the virtual interface address. SIOCDELIFVIPA removes the list of specified interfaces that are pointed to by <i>ifrv.ifrv_ifname</i> and associated with the virtual interface specified by <i>ifrv.ifrv_name</i>, by using SIOCADDIFVIPA. SIOCLISTIFVIPA lists all the interfaces that are associated with the virtual interface specified by <i>ifrv.ifrv_name</i>.</p> <pre data-bbox="662 499 1182 552">           ioctl(fd, SIOCADDIFVIPA, (caddr_t)&amp;ifrv);           struct ifvireq ifrv;         </pre> <p>The virtual interface information is stored in the <i>ifvireq</i> structure.</p> <p><b>Note:</b> These flags operate on a virtual interface only.</p>
SIOCSIFADDR6	Set or Add an IPv6 address. <pre data-bbox="662 741 1157 793">           ioctl(fd, SIOCSIFADDR6, (caddr_t)&amp;ifr);           struct in6_ifreq ifr;         </pre>
SIOCGIFADDR6	Gets an IPv6 address. <pre data-bbox="662 888 1157 940">           ioctl(fd, SIOCGIFADDR6, (caddr_t)&amp;ifr);           struct in6_ifreq ifr;         </pre>
SIOCSIFDSTADDR6	Set the destination (point-to-point) address for a IPv6 address. <pre data-bbox="662 1035 1198 1087">           ioctl(fd, SIOCSIFDSTADDR6, (caddr_t)&amp;ifr);           struct in6_ifreq ifr;         </pre>
SIOCGIFDSTADDR6	Get the destination (point-to-point) address for a IPv6 address. <pre data-bbox="662 1178 1198 1230">           ioctl(fd, SIOCGIFDSTADDR6, (caddr_t)&amp;ifr);           struct in6_ifreq ifr;         </pre>
SIOCSIFNETMASK6	Set the netmask for an IPv6 address. <pre data-bbox="662 1318 1198 1371">           ioctl(fd, SIOCSIFNETMASK6, (caddr_t)&amp;ifr);           struct in6_ifreq ifr;         </pre>
SIOCGIFNETMASK6	Get the netmask for an IPv6 address. <pre data-bbox="662 1465 1198 1518">           ioctl(fd, SIOCGIFNETMASK6, (caddr_t)&amp;ifr);           struct in6_ifreq ifr;         </pre>
SIOCDIFADDR6	Delete an IPv6 address. <pre data-bbox="662 1608 1157 1661">           ioctl(fd, SIOCDIFADDR6, (caddr_t)&amp;ifr);           struct in6_ifreq ifr;         </pre>
SIOCFIFADDR6	Put an IPv6 address at the beginning of the address list. <pre data-bbox="662 1755 1157 1808">           ioctl(fd, SIOCFIFADDR6, (caddr_t)&amp;ifr);           struct in6_ifreq ifr;         </pre>

<b>ioctl command</b>	<b>Description</b>
SIOCAIFADDR6	Add or change an IPv6 alias address.  <pre>ioctl(fd, SIOCAIFADDR6, (caddr_t)&amp;ifra); struct in6_aliasreq ifra;</pre>
SIOCADDANY6	Add an IPv6 anycast address.  <pre>ioctl(fd, SIOCADDANY6, (caddr_t)&amp;ifra); struct in6_ifreq ifr;</pre>
SIOCDELANY6	Delete an IPv6 anycast address.  <pre>ioctl(fd, SIOCDELANY6, (caddr_t)&amp;ifra); struct in6_ifreq ifr;</pre>
SIOCSIFZONE6	Set the IPv6 zone ID of an interface at a particular address scope.  <pre>ioctl(fd, SIOCSIFZONE6, (caddr_t)&amp;ifrz); struct in6_zonereq ifrz;</pre>
SIOCGIFZONE6	Get the IPv6 scope zone IDs of an interface.  <pre>ioctl(fd, SIOCGIFZONE6, (caddr_t)&amp;ifrz); struct in6_zonereq ifrz;</pre>
SIOCSIFADDRORI6	Set the configuration origin for an IPv6 address.  <pre>ioctl(fd, SIOCSIFADDRORI6, (caddr_t)&amp;ifro); struct ifaddrorigin6 ifro;</pre>
SIOCAIFADDR6T	Add or change an IPv6 alias address and type.  <pre>ioctl(fd, SIOCAIFADDR6T, (caddr_t)&amp;ifra); struct in6_aliasreq2 ifra;</pre>
SIOCGIFADDR6T	Get the type of an IPv6 address.  <pre>ioctl(fd, SIOCGIFADDR6T, (caddr_t)&amp;ifra); struct in6_aliasreq2 ifra;</pre>
SIOCSIFADDRSTATE6	Change the state of an IPv6 address.  <pre>ioctl(fd, SIOCSIFADDRSTATE6, (caddr_t)&amp;ifra); struct in6_aliasreq2 ifra;</pre>
SIOCGIFADDRSTATE6	Get the state of an IPv6 address.  <pre>ioctl(fd, SIOCGIFADDRSTATE6, (caddr_t)&amp;ifra); struct in6_aliasreq2 ifra;</pre>
SIOCGSRCFILTER6	Get the IPv6 multicast group source filter for an interface.  <pre>ioctl(fd, SIOCGSRCFILTER6, (caddr_t)&amp;ifrgsf); struct group_source_filter_req ifrgsf;</pre>
SIOCACLADDR6	Add an IPv6 cluster alias address.  <pre>ioctl(fd, SIOCACLADDR6, (caddr_t)&amp;ifra); struct in6_aliasreq ifra;</pre>

<b>ioctl command</b>	<b>Description</b>
SIOCDCCLADDR6	Delete an IPv6 cluster address.  <pre>ioctl(<i>fd</i>, SIOCDCCLADDR6, (<i>caddr_t</i>)&amp;<i>ifr</i>); struct in6_ifreq <i>ifr</i>;</pre>
SIOCSIFADDRFLAG6	Set address source flag for an IPv6 address.  <pre>ioctl(<i>fd</i>, SIOCSIFADDRFLAG6, (<i>caddr_t</i>)&amp;<i>ifra2</i>); struct in6_aliasreq2 <i>ifra2</i>;</pre>
SIOCGIFADDRFLAG6	Get address source flag for an IPv6 address.  <pre>ioctl(<i>fd</i>, SIOCGIFADDRFLAG6, (<i>caddr_t</i>)&amp;<i>ifra2</i>); struct in6_aliasreq2 <i>ifra2</i>;</pre>

## Return Values

Upon successful completion, `ioctl` returns 0. Otherwise, it returns -1 and sets `errno` to indicate the error.

## Error Codes

The `ioctl` commands fail under the following general conditions:

<b>Item</b>	<b>Description</b>
EBADF	The file descriptor <i>fd</i> is not a valid open socket file descriptor.
EINTR	A signal was caught during <code>ioctl</code> operation.
EINVAL	An invalid command or argument was specified.

If the underlying operation specified by the `ioctl` command *cmd* failed, `ioctl` fails with one of the following error codes:

<b>Item</b>	<b>Description</b>
EACCES	Permission that is denied for the specified operation.
EADDRNOTAVAIL	Specified address not available for interface.
EAFNOSUPPORT	Operation that is not supported on sockets.
EBUSY	Resource is busy.
EEXIST	An entry or file exists.
EFAULT	Argument references an inaccessible memory area.
EIO	Input/Output error.
ENETUNREACH	Gateway unreachable.
ENOBUFS	Routing table overflow.
ENOCONNECT	No connection.
ENOMEM	Not enough memory available.
ENOTCONN	The operation is only defined on a connected socket, but the socket was not connected.
ENXIO	Device does not exist.
ESRCH	No such process.

## Related information

[Socket Overview](#)

[ioctl subroutine](#)

## isinet\_addr Subroutine

### Purpose

Determines if the given ASCII string contains an Internet address using dot notation.

### Library

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

### Syntax

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

```
u_long isinet_addr (name)
char * name;
```

### Description

The **isinet\_addr** subroutine determines if the given ASCII string contains an Internet address using dot notation (for example, "120.121.122.123"). The **isaddr\_inet** subroutine considers Internet address strings as a valid string, and considers any other string type as an invalid strings.

The **isinet\_addr** subroutine expects the ASCII string to conform to the following format:

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

#### Value Description

- $A^n$  Indicates  $n$  repetitions of pattern A.
- $A^n-m$  Indicates  $n$  to  $m$  repetitions of pattern A.
- $A^*$  Indicates zero or more repetitions of pattern A, up to environmental limits.

The BNF description explicitly states the space character (' '), if used.

#### Value Description

- $\{text\}$  Indicates  $text$ , not a BNF symbol.

The **isinet\_addr** subroutine allows the application to terminate the string with a null terminator (0x00) or a space (0x30). It ignores characters trailing the space character and considers the string invalid if the application does not terminate the string with a null terminator (0x00) or space (0x30).

The following describes the restrictions on the field values:

Address Format	Field Restrictions (values in decimal base)
a	$a < 4294967296$ .
a.b	$a < 256$ ; $b < 16777216$ .

<b>Address Format</b>	<b>Field Restrictions (values in decimal base)</b>
-----------------------	--

a.b.c	a < 256; b < 256; c < 16777216.
a.b.c.d	a < 256; b < 2 <sup>8</sup> ; c < 256; d < 256.

The **isinet\_addr** subroutine applications can enter field values exceeding the field value restrictions specified previously; **isinet\_addr** accepts the least significant bits up to an integer in length. The **isinet\_addr** subroutine still checks to see if the truncated value exceeds the maximum field value. For example, if an application gives the string 0.0;0;0xFF00000001 then **isinet\_addr** interprets the string as 0.0.0.0x00000001 and considers the string as valid.

**isinet\_addr** applications cannot omit field values between delimiters and considers a string with successive periods as invalid.

Examples of valid strings:

<b>Input String</b>	<b>Comment</b>
1	<b>isinet_addr</b> uses <b>a</b> format.
1.2	<b>isinet_addr</b> uses <b>a.b</b> format.
1.2.3.4	<b>isinet_addr</b> uses <b>a.b.c.d</b> format.
0x01.0X2.03.004	<b>isinet_addr</b> uses <b>a.b.c.d</b> format.
1.2 3.4	<b>isinet_addr</b> uses <b>a.b</b> format; and ignores "3.4".

Examples of invalid strings:

<b>Input String</b>	<b>Reason</b>
...	No explicit field values specified.
1.2.3.4.5	Excessive fields.
1.2.3.4.	Excessive delimiters and fields.
1,2	Bad delimiter.
1p	String not terminated by null terminator nor space.
{empty string}	No field or delimiter present.
9999.1.1.1	Value for field a exceeds limit.

**Note:**

1. The **isinet\_addr** subroutine does not check the pointer to the ASCII string; the user takes responsibility for ensuring validity of the address of the ASCII string.
2. The application assumes responsibility for verifying that the network number and host number for the Internet address conforms to a class A or B or C Internet address; any other string is processed as a class C address.

All applications using **isinet\_addr** must compile with the **\_BSD** macro defined. Also, all socket applications must include the BSD library **libbsd** when applicable.

## Parameters

<b>Item</b>	<b>Description</b>
-------------	--------------------

<i>name</i>	Address of ASCII string buffer.
-------------	---------------------------------

## Return Values

The **isinet\_addr** subroutine returns 1 for valid input strings and 0 for invalid input strings. **isinet\_addr** returns the value as an unsigned long type.

## Files

**#include <ctype.h>**

**#include <sys/types.h>**

## kvalid\_user Subroutine

### Purpose

This routine maps the DCE principal to the local user account and determines if the DCE principal is allowed access to the account.

### Library

Valid User Library (**libvaliduser.a**)

### Syntax

```
int kvalid_user (char *princ_name, char *local_user)
```

### Description

This routine is called when Kerberos 5 authentication is configured to determine if the incoming Kerberos 5 ticket should allow access to the local account.

This routine determines whether the DCE principal, specified by the *princ\_name* parameter, is allowed access to the user's account identified by the *local\_user* parameter. The routine accesses the **\$HOME/.k5login** file for the user's account. It looks for the string pointed to by *princ\_name* in that file.

Access is granted if one of two things is true.

1. The **\$HOME/.k5login** file exists and the *princ\_name* is in it.
2. The **\$HOME/.k5login** file does NOT exist and the DCE principal name is the same as the local user's name.

### Parameters

Item	Description
<i>princ_name</i>	This parameter is a single-string representation of the Kerberos 5 principal. The Kerberos 5 libraries have two services, <code>krb5_unparse_name</code> and <code>krb5_parse_name</code> , which convert a <code>krb5_principal</code> structure to and from a single-string format. This routine expects the <i>princ_name</i> parameter to be a single-string form of the <code>krb5_principal</code> structure.
<i>local_user</i>	This parameter is the character string holding the name of the local account.

### Return Values

If the user is allowed access to the account, the **kvalid\_user** routine returns TRUE.

If the user is NOT allowed access to the account or there was an error, the **kvalid\_user** routine returns FALSE.

## Related information

[Communications and networks](#)

[Authentication and the secure rcmds](#)

# listen Subroutine

## Purpose

Listens for socket connections and limits the backlog of incoming connections.

## Library

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

## Syntax

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

```
int listen ( Socket, Backlog )  
int Socket, Backlog;
```

## Description

The **listen** subroutine performs the following activities:

1. Identifies the socket that receives the connections.
2. Marks the socket as accepting connections.
3. Limits the number of outstanding connection requests in the system queue.

The outstanding connection request queue length limit is specified by the parameter *backlog* per **listen** call. A *no* parameter - *somaxconn* - defines the maximum queue length limit allowed on the system, so the effective queue length limit will be either *backlog* or *somaxconn*, whichever is smaller.

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

## Parameters

Item	Description
<i>Socket</i>	Specifies the unique name for the socket.
<i>Backlog</i>	Specifies the maximum number of outstanding connection requests.

## Return Values

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

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

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

## Error Codes

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

Error	Description
<b>EBADF</b>	The <i>Socket</i> parameter is not valid.
<b>ECONNREFUSED</b>	The host refused service, usually due to a server process missing at the requested name or the request exceeding the backlog amount.
<b>EINVAL</b>	The socket is already connected.
<b>ENOTSOCK</b>	The <i>Socket</i> parameter refers to a file, not a socket.
<b>EOPNOTSUPP</b>	The referenced socket is not a type that supports the <b>listen</b> subroutine.

## Examples

The following program fragment illustrates the use of the **listen** subroutine with 5 as the maximum number of outstanding connections which may be queued awaiting acceptance by the server process.

```
listen(s,5)
```

### Related reference

[accept Subroutine](#)

### Related information

[Accepting Internet Stream Connections Example Program](#)

[Sockets Overview](#)

[Understanding Socket Connections](#)

## n

AIX runtime services beginning with the letter *n*.

## ntohl Subroutine

### Purpose

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

### Library

ISODE Library (**libisode.a**)

### Syntax

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

```
uint32_t ntohl ( NetLong )
uint32_t NetLong;
```

### Description

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

Receiving hosts require addresses and ports in host byte order. Use the **ntohl** subroutine to convert Internet addresses and ports to the host integer representation.

The **ntohl** subroutine is defined in the **net/nh.h** file as a null macro if the host byte order is same as the network byte order.

The **ntohl** subroutine is declared in the **net/nh.h** file as a function if the host byte order is not same as the network byte order.

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

## Parameters

Item	Description
<i>NetLong</i>	Requires a 32-bit integer in network byte order.

## Return Values

The **ntohl** subroutine returns a 32-bit integer in host byte order.

### Related information

[Sockets Overview](#)

## ntohll Subroutine

### Purpose

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

### Library

ISODE Library (**libisode.a**)

### Syntax

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

```
uint64_t ntohll ( NetLong)
uint64_t NetLong;
```

### Description

The **ntohll** subroutine converts an unsigned long (64-bit) integer from Internet network standard byte order to host byte order.

Receiving hosts require addresses and ports in host byte order. Use the **ntohll** subroutine to convert Internet addresses and ports to the host integer representation.

The **ntohll** subroutine is defined in the **net/nh.h** file as a null macro if the host byte order is the same as the network byte order.

The **ntohll** subroutine is declared in the **net/nh.h** file as a function if the host byte order is not the same as the network byte order.

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

## Parameters

Item	Description
<i>NetLong</i>	Requires a 64-bit integer in network byte order.

## Return Values

The **ntohl** subroutine returns a 64-bit integer in host byte order.

## Related information

[Sockets Overview](#)

## ntohs Subroutine

### Purpose

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

### Library

ISODE Library (**libisode.a**)

### Syntax

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

```
uint16_t ntohs ( NetShort )
uint16_t NetShort;
```

### Description

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

Receiving hosts require Internet addresses and ports in host byte order. Use the **ntohs** subroutine to convert Internet addresses and ports to the host integer representation.

The **ntohs** subroutine is defined in the **net/nh.h** file as a null macro if the host byte order is same as the network byte order.

The **ntohs** subroutine is declared in the **net/nh.h** file as a function if the host byte order is not same as the network byte order.

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

## Parameters

Item	Description
<i>NetShort</i>	Requires a 16-bit integer in network standard byte order.

## Return Values

The **ntohs** subroutine returns a 16-bit integer in host byte order.

## Related information

[Sockets Overview](#)

# PostQueuedCompletionStatus Subroutine

## Purpose

Post a completion packet to a specified I/O completion port.

## Syntax

```
#include <iocp.h>
boolean_t PostQueuedCompletionStatus (CompletionPort, TransferCount, CompletionKey, Overlapped,
)
HANDLE CompletionPort;
DWORD TransferCount, CompletionKey;
LPOVERLAPPED Overlapped;
```

## Description

The **PostQueuedCompletionStatus** subroutine attempts to post a completion packet to *CompletionPort* with the values of the completion packet populated by the *TransferCount*, *CompletionKey*, and *Overlapped* parameters.

The **PostQueuedCompletionStatus** subroutine returns a boolean indicating whether or not a completion packet has been posted.

The **PostQueuedCompletionStatus** subroutine is part of the I/O Completion Port (IOCP) kernel extension.

**Note:** This subroutine only works to a socket file descriptor. It does not work with files or other file descriptors.

## Parameters

Item	Description
<i>CompletionPort</i>	Specifies the completion port that this subroutine will attempt to access.
<i>TransferCount</i>	Specifies the number of bytes transferred.
<i>CompletionKey</i>	Specifies the completion key.
<i>Overlapped</i>	Specifies the overlapped structure.

## Return Values

Upon successful completion, the **PostQueuedCompletionStatus** subroutine returns a boolean indicating its success.

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

- Returns a value of 0 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable, see the link in the Related Information section of this document.

## Error Codes

The subroutine is unsuccessful if either of the following errors occur:

Item	Description
<b>EBADF</b>	The <i>CompletionPort</i> parameter was NULL.
<b>EINVAL</b>	The <i>CompletionPort</i> parameter was invalid.

## Examples

The following program fragment illustrates the use of the **PostQueuedCompletionStatus** subroutine to post a completion packet.

```
c = GetQueuedCompletionStatus (34, 128, 25, struct overlapped);
```

## Related information

[Error Notification Object Class](#)

## r

AIX runtime services beginning with the letter *r*.

## rcmd Subroutine

### Purpose

Allows execution of commands on a remote host.

### Library

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

### Syntax

```
int rcmd (Host,
Port, LocalUser, RemoteUser, Command, ErrFileDesc)
char ** Host;
u_short Port;
char * LocalUser;
char * RemoteUser;
char * Command;
int * ErrFileDesc;
```

### Description

The **rcmd** subroutine allows execution of certain commands on a remote host that supports **rshd**, **rlogin**, and **rpc** among others.

Only processes with an effective user ID of root user can use the **rcmd** subroutine. An authentication scheme based on remote port numbers is used to verify permissions. Ports in the range between 0 and 1023 can only be used by a root user. The application must pass in *Port*, which must be in the range 512 to 1023.

The **rcmd** subroutine looks up a host by way of the name server or if the local name server isn't running, in the [/etc/hosts](#) file.

If the connection succeeds, a socket in the Internet domain of type **SOCK\_STREAM** is returned to the calling process and given to the remote command as standard input (**stdin**) and standard output (**stdout**).

Always specify the *Host* parameter. If the local domain and remote domain are the same, specifying the domain parts is optional.

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

## Parameters

Item	Description
<i>Host</i>	Specifies the name of a remote host that is listed in the <b>/etc/hosts</b> file. If the specified name of the host is not found in this file, the <b>rcmd</b> subroutine is unsuccessful.
<i>Port</i>	Specifies the well-known port to use for the connection. The <b>/etc/services</b> file contains the DARPA Internet services, their ports, and socket types.
<i>LocalUser</i> and <i>RemoteUser</i>	Points to user names that are valid at the local and remote host, respectively. Any valid user name can be given.
<i>Command</i>	Specifies the name of the command to be started at the remote host.
<i>ErrFileDesc</i>	Specifies an integer controlling the set up of communication channels. Integer options are as follows: <b>Non-zero</b> Indicates an auxiliary channel to a control process is set up, and the <i>ErrFileDesc</i> parameter points to the file descriptor for the channel. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command. <b>0</b> Indicates the standard error ( <b>stderr</b> ) of the remote command is the same as standard output ( <b>stdout</b> ). No provision is made for sending arbitrary signals to the remote process. However, it is possible to send out-of-band data to the remote command.

## Return Values

Upon successful completion, the **rcmd** subroutine returns a valid socket descriptor.

Upon unsuccessful completion, the **rcmd** subroutine returns a value of -1. The subroutine returns a -1, if the effective user ID of the calling process is not root user or if the subroutine is unsuccessful to resolve the host.

## Files

Item	Description
<b><u>/etc/services</u></b>	Contains the service names, ports, and socket type.
<b><u>/etc/hosts</u></b>	Contains host names and their addresses for hosts in a network.
<b><u>/etc/resolv.conf</u></b>	Contains the name server and domain name.

## Related information

[Sockets Overview](#)

## rcmd\_af Subroutine

### Purpose

Allows execution of commands on a remote host.

### Syntax

```
int rcmd_af(char **ahost, unsigned short rport,  
            const char *locuser, const char *remuser,  
            const char *cmd, int *fd2p, int af)
```

### Description

The **rcmd\_af** subroutine allows execution of certain commands on a remote host that supports **rshd**, **rlogin**, and **rpc** among others. It behaves the same as the existing **rcmd()** function, but instead of creating only an AF\_INET TCP socket, it can also create an AF\_INET6 TCP socket. The existing **rcmd()** function cannot transparently use AF\_INET6 sockets because an application would not be prepared to handle AF\_INET6 addresses returned by subroutines such as **getpeername()** on the file descriptor created by **rcmd()**.

Only processes with an effective user ID of root user can use the **rcmd\_af** subroutine. An authentication scheme based on remote port numbers is used to verify permissions. Ports in the range between 0 and 1023 can only be used by a root user.

The **rcmd\_af** subroutine looks up a host by way of the name server or if the local name server is not running, in the **/etc/hosts** file.

If the connection succeeds, a socket in the Internet domain of type **SOCK\_STREAM** is returned to the calling process and given to the remote command as standard input (**stdin**) and standard output (**stdout**).

Always specify the *ahost* parameter. If the local domain and remote domain are the same, specifying the domain parts is optional.

### Parameters

Item	Description
<i>ahost</i>	Specifies the name of a remote host that is listed in the <b>/etc/hosts</b> file. If the specified name of the host is not found in this file, the <b>rcmd_af</b> subroutine is unsuccessful.
<i>rport</i>	Specifies the well-known port to use for the connection. The <b>/etc/services</b> file contains the DARPA Internet services, their ports, and socket types.
<i>locuser</i>	Points to user names that are valid at the local host. Any valid user name can be given.
<i>remuser</i>	Points to user names that are valid at the remote host. Any valid user name can be given.
<i>cmd</i>	Specifies the name of the command to be started at the remote host.

<b>Item</b>	<b>Description</b>
<i>fd2p</i>	<p>Specifies an integer controlling the set up of communication channels. Integer options are as follows:</p> <p><b>Non-zero</b> Indicates an auxiliary channel to a control process is set up, and the <i>fd2p</i> parameter points to the file descriptor for the channel. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command.</p> <p><b>0</b> Indicates the standard error (<b>stderr</b>) of the remote command is the same as standard output (<b>stdout</b>). No provision is made for sending arbitrary signals to the remote process. However, it is possible to send out-of-band data to the remote command.</p>
<i>af</i>	<p>The family argument is AF_INET, AF_INET6, or AF_UNSPEC. When either AF_INET or AF_INET6 is specified, this function will create a socket of the specified address family. When AF_UNSPEC is specified, it will try all possible address families until a connection can be established, and will return the associated socket of the connection.</p>

## Return Values

Upon successful completion, the **rcmd\_af** subroutine returns a valid socket descriptor. Upon unsuccessful completion, the **rcmd\_af** subroutine returns a value of `-1`. The subroutine returns a `-1` if the effective user ID of the calling process is not the root user or if the subroutine is unsuccessful to resolve the host.

## Files

<b>Item</b>	<b>Description</b>
<u><a href="#">/etc/services</a></u>	Contains the service names, ports, and socket type.
<u><a href="#">/etc/hosts</a></u>	Contains host names and their addresses for hosts in a network.
<u><a href="#">/etc/resolv.conf</a></u>	Contains the name server and domain name.

## rds Subroutine

### Purpose

Reliable Datagram Sockets (RDS) provides reliable, in-order datagram delivery between sockets across various network transport.

### Library

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/bypass.h>
#include <net/rds_rdma.h>
```

## Description

RDS is an implementation of the RDS Application Programming Interface (API). RDS can be transported through InfiniBand and loopback. RDS through TCP is disabled. RDS uses the standard AF\_INET addresses to identify the endpoints.

## Socket Creation

RDS sockets are created as follows:

```
rds_socket = socket(AF_BYPASS, SOCK_SEQPACKET, BYPASSPROTO_RDS);
```

## Socket Options

RDS supports multiple socket options through the **setsockopt** and **getsockopt** calls. The following options with the SOL\_SOCKET socket level are important.

### SO\_RCVBUF

Specifies the size of the receive buffer. See Congestion Control.

### SO\_SNDBUF

Specifies the size of the send buffer. See Message Transmission.

### SO\_SNDTIMEO

Specifies the send timeout of the socket when you enqueue a message on a socket with a full queue in the blocking mode.

RDS also supports multiple protocol-specific options with the SOL\_RDS socket level .

## Binding

A new RDS has no local address when it is initially returned from the **socket** call. The socket must be bound to a local address by running the **bind** system call before any messages are sent or received. The **bind** call attaches the socket to a specific network transport, which is based on the type of interface the local address is attached to. From the point the call is attached to the socket, the socket can reach the destinations that are available through this network transport.

For instance, when binding to the address of an InfiniBand interface, such as ib0, the socket uses the InfiniBand transport system. If RDS is not able to associate a transport system with the specific address, it returns the EADDRNOTAVAIL value.

An RDS socket can only be bound to one address and only one socket can be bound to a specific address or port pair. If no port is specified in the binding address, an unbound port is selected at random.

RDS does not permit the application to bind a previously bound socket to another address. Binding to the INADDR\_ANY wildcard address is not allowed.

## Connecting

In the default mode of operation RDS uses unconnected sockets, and specifies destination address as an argument to the **sendmsg** subroutine. However, RDS allows sockets to be connected to a remote end point by using the **connect** subroutine. If a socket is connected, you can call the **sendmsg** subroutine without specifying a destination address and the subroutine uses the remote address that was previously provided.

## Congestion Control

RDS does not have an explicit congestion control mechanism like the common streaming protocols such as TCP. The sockets have two queue limits that are the send queue size and the receive queue size. Messages are accounted based on the number of bytes of payload.

The `send_queue_size` limits the data that the local processes can queue on a local socket. If the limit exceeds, the kernel does not accept messages until the queue is free and messages are delivered and acknowledged by the remote host.

The `receive_queue_size` limits the data that RDS stores on the receive queue of a socket before marking the socket as **congested**. When a socket becomes congested, RDS sends a *congestion map* update to the other participating hosts, which are then expected to stop sending more messages to this port.

There is a timing window during which a remote host can continue to send messages to a congested port. RDS resolves the timing window by accepting messages even when the receive queue of the socket exceeds the limit.

When the application receives incoming messages from the receive queue by using the **recvmsg** system call, the number of bytes on the receive queue reduces below the receive queue size and the port is marked as uncongested. A congestion update is sent to all the participating hosts.

The values for the send buffer size and receive buffer size can be tuned by the application through the `SO_SNDBUF` and `SO_RCVBUF` socket options.

## Blocking Behavior

The **sendmsg** and **recvmsg** calls can be blocked in various situations. A call can be blocked or returned with an error depending on the non-blocking setting of the file descriptor and the `MSG_DONTWAIT` message flag. If the file descriptor is set to blocking mode (which is the default), and the `MSG_DONTWAIT` flag is not specified, the call is blocked.

The **SO\_SNDTIMEO** and **SO\_RCVTIMEO** socket options are used to specify a timeout (in seconds) after which the call ends and returns an error. The default timeout is 0, which allows RDS to block indefinitely.

## Message Transmission

Messages can be sent by using the **sendmsg** call after the RDS socket is bound. Message length cannot exceed 4 GB as the wire protocol uses an unsigned 32-bit integer to express the message length.

RDS does not support data that is out-of-band. Applications can send data to unicast addresses only, where broadcast or multicast are not supported.

A successful **sendmsg** call places the message in the transmit queue of the socket where it remains until the destination acknowledges that the message is no longer in the network or the application removes the message from the send queue.

Messages can be removed from the send queue with the **RDS\_CANCEL\_SENT\_TO** socket option.

When a message is in the transmit queue, its payload bytes are considered. If an attempt is made to send a message when the transmit queue is not free, the call blocks or returns the `EAGAIN` value.

When messages are sent to a destination that is marked as congested, the call is blocked or the `ENOBUFS` value is returned.

A message that is sent with no payload bytes does not require any space in the send buffer of the destination but a message receipt is sent to the destination. The receiver cannot get any payload data but the address of the sender can be viewed.

Messages sent to a port to which no socket is bound is discarded by the destination host. No error messages are reported to the sender.

## Message Receipt

Messages can be received with the **recvmsg** call on RDS after it is bound to a source address. RDS returns messages in the same order that the sender sent the messages.

The address of the sender is returned in the `sockaddr_in` structure pointed by the `msg_name` field, if the field is set.

If the MSG\_PEEK flag is set, the first message on the receive queue is returned without removing the message from the queue.

The memory that is used by messages waiting to be delivered does not limit the number of messages that can be queued to be received. RDS attempts to control congestion.

If the length of the message exceeds the size of the buffer that is provided to **recvmsg** call, then the remaining bytes in the message are discarded and the MSG\_TRUNC flag is set in the msg\_flags field. In this case the **recvmsg** call, returns the number of bytes copied. It does not return the length of the entire message. If MSG\_TRUNC is set in the flags argument to **recvmsg**, it returns the number of bytes in the entire message. You can view the size of the next message in the receive queue without providing a zero length buffer and setting the MSG\_PEEK and MSG\_TRUNC options in the flags argument.

The sending address of a zero-length message is provided in the msg\_name field.

## Control Messages

RDS uses control messages that is the ancillary data by using the **msg\_control** and **msg\_controllen** fields in the **sendmsg** and **recvmsg** calls. Control messages that are generated by RDS have a **cmsg\_level** value of `sol_rds`. Most control messages are related to the zerocopy interface added in RDS version 3, and are described in the **rds-rdma** subroutine.

The only exception is the RDS\_CMSG\_CONG\_UPDATE message.

## Polling

Support for the **poll** interface is limited. POLLIN is returned when there is an RDS message, or a control message waiting in the receive queue of the socket. POLLOUT is returned when there is space on the send queue of the socket.

Sending messages to the congested ports requires special handling mechanism. When an application tries to send message to a congested destination, the system call returns the ENOBUFS value. RDS cannot poll for POLLOUT because the transmit queue can still accommodate the messages and the call to the **poll** interface might return immediately, even though the destination is congested.

You can perform one of the method to handle the congestion:

- Poll for the POLLIN option. By default, a process sleeping in the **poll** interface is activated when the congestion map is updated. The application can retry any previously congested send operation.
- Monitor the explicit congestion, which gives the application greater control.

With explicit monitoring, the application polls for POLLIN option as before, and additionally uses the **RDS\_CONG\_MONITOR** socket option to install a 64-bit mask value in the socket, where each bit corresponds to a group of ports. When a congestion update is received, RDS socket checks the set of ports that became uncongested against the bit mask that is installed in the socket. If they overlap, a control message is enqueued on the socket, and the application is activated. When **recvmsg** call is called, RDS gives the control message that contains the bitmap on the socket.

The congestion monitor bitmask can be set and queried by using the **setsockopt** call with the **RDS\_CONG\_MONITOR** option, and a pointer to the 64-bit mask variable.

Congestion updates are delivered to the application through the **RDS\_CMSG\_CONG\_UPDATE** control messages. The control messages are delivered separately, but never with RDS data message. The `cmsg_data` field of the control message is an eight byte data that contains the 64-bit mask value.

Applications can use the following macros to test for and set bits in the bitmask:

```
#define RDS_CONG_MONITOR_SIZE    64
#define RDS_CONG_MONITOR_BIT(port) (((unsigned int) port) % RDS_CONG_MONITOR_SIZE)
#define RDS_CONG_MONITOR_MASK(port) (1 << RDS_CONG_MONITOR_BIT(port))
```

## Canceling Messages

An application can cancel messages from the send queue by using the **RDS\_CANCEL\_SENT\_TO** socket option with the **setsockopt** call. The **setsockopt** call uses an optional `sockaddr_in` address structure as an argument. Only messages to the destination address that is specified by the `sockaddr_in` address are discarded. If no address is provided, all pending messages are discarded.

**Note:** This call affects messages that are not transmitted and messages that are transmitted but no acknowledgment is received from the remote host.

## Reliability

If the **sendmsg** succeeds, RDS guarantees that the message is visible to **recvmsg** on a socket that is bound to the destination address as long as that destination socket remains open.

If there is no socket bound on the destination, the message is dropped. If the RDS that is sending messages is not sure that a socket is bound, it tries to send the message indefinitely until it is sure or the sent message is canceled.

If a socket is closed, the pending sent messages on the socket are canceled and can or cannot be seen by the receiver.

The **RDS\_CANCEL\_SENT\_TO** socket option can be used to cancel all the pending messages to a given destination.

If a receiving socket is closed with pending messages, then the sender considers those messages as having left the network and will not retransmit them.

A message is seen by the **recvmsg** call unless the **MSG\_PEEK** is specified. When the message is delivered it is removed from the transmit queue of the sending socket.

All messages sent from the same socket to the same destination is delivered in the order they are sent. Messages sent from different sockets, or to different destinations, are delivered randomly.

## rds-info Subroutine

### Purpose

Displays information from the kernel extension of the Reliable Datagram Sockets (RDS) .

### Syntax

```
rds-info [-v ] [ -cknrst]
```

### Description

The **rds-info** utility displays various sources of information that the RDS kernel module maintains. When you run the **rds-info** utility without any optional arguments, the output has all the information. When you specify the optional arguments, the information that is associated with those options is displayed.

## Parameters

### Item

-c

### Descriptor

Displays global counters. Each counter increments after the event occurs. You cannot reset the counters. The set of the supported counters can change with time. The list of output fields includes:

#### CounterName

The name of the counter. These names are derived from the kernel and can change based on the capability of the kernel extension.

#### Value

The number of times the counter increments after the kernel module is loaded.

-k

Displays all the RDS sockets in the system. There is one socket that is listed at a time that is not bound to or connected to any address because the **rds-info** utility uses an unbound socket to collect information. The list of output fields includes:

#### BoundAddr, BPort

The IP address and port number to which the socket is bound. The 0.0.0.0 0 address indicates that the socket is not bound.

#### ConnAddr, CPort

The IP address and port number to which the socket is connected. The 0.0.0.0 0 address indicates that the socket is not connected.

#### SndBuf, RcvBuf

The message payload in bytes that can be queued for sending or receiving on the respective socket.

**Item**

-n

**Descriptor**

Displays all the RDS connections. RDS connections are maintained between nodes by the network transports. The list of output fields includes:

**LocalAddr**

The IP address of a node. For connections that originate and terminate on the same node, the local address indicates the address that initiated the connection establishment and

**RemoteAddr**

The IP address of the remote end of the connection.

**NextTX**

The sequence number that is given to the next message that is sent over the connection.

**NextRX**

The expected sequence number of the next message that arrives over the connection. Any incoming messages with sequence numbers less than the expected number is dropped.

**Flg**

Flags that indicate the state of the connection.

**s**

A process is sending a message down the connection.

**c**

The transport is attempting to connect to the remote address.

**C**

The connection to the remote host is active.

-r, -s, -t

Displays the messages in the receive, send, or retransmit queues.

**LocalAddr, LPort**

The local IP address and port number of the node that is associated with the message. For sent messages, this address is the source address. For receive messages, this address is the destination address.

**RemoteAddr, RPort**

The remote IP address and port number that is associated with the message. For sent messages, this address is the destination address. For receive messages, this address is the source address.

**Seq**

The sequence number of the message.

**Bytes**

The message payload in bytes.

-v

Displays verbose output. When this option is specified complete data is displayed.

## rds-ping Subroutine

### Purpose

Tests the reachability of the remote node over Reliable Datagram Sockets (RDS).

### Syntax

```
rds-ping [-ccount] [-iinterval] [-Ilocal_addr] remote_addr
```

### Description

The **rds-ping** utility is used to test whether a remote node is reachable over RDS. The RDS interface is designed to operate like the standard ping utility, with a difference. The **rds-ping** utility opens several RDS sockets and sends packets to port 0 on the specified host. This port is a special port number to which no socket is bound to, and the kernel processes the incoming packets and responds to them.

### Parameters

Item	Description
-c <i>count</i>	Causes the <b>rds-ping</b> utility to exit after the specified number of packets are sent and received.
-I <i>address</i>	Accepts the local source address for the RDS socket that is based on the routing information for the specified destination address. For example, if packets to a specific destination are routed through the ib0 interface, it uses the IP address of ib0 as the source address. By using the -I option, you can override this choice.
-i <i>timeout</i>	Waits for one second between sending packets, by default. Use this option to specify a different interval. The timeout value is given in seconds, and can be a floating point number. Optionally, append the msec or usec parameter to specify the timeout in milliseconds or microseconds.  <b>Note:</b> If you specify a timeout that is considerably smaller than the packet round-trip time, it produces unexpected results.

## rds-rdma Subroutine

### Purpose

Reliable Datagram Sockets (RDS) zerocopy provides an interface for remote direct memory access (RDMA) over RDS.

### Description

The zerocopy interface of RDS was added in RDS Version3. In the RDS zerocopy, the client initiates a direct transfer to or from an area of the memory in its process address space. This memory need not be aligned.

The client obtains a handle for this region of memory, and passes it to the server. This cookie is called the RDMA cookie. To the application, the cookie is an opaque 64-bit data type.

The client sends this handle to the server application, along with other details of the RDMA request such as the data to transfer to the RDMA memory area. This message is called the RDMA request.

The server uses the RDMA cookie to initiate the requested RDMA transfer. The RDMA transfer is combined atomically with a normal RDS message, which is delivered to the client. This message is called the RDMA ACK. Atomic refers to both the RDMA succeeds and the RDMA ACK delivered, or they do not succeed.

When the client receives the RDMA ACK, it means that the RDMA completed successfully. If required, it can then release the RDMA cookie for this memory region.

RDMA operations are not reliable. Unlike normal RDS messages, RDS RDMA operations fail and get dropped.

## Interface

The interface is based on control messages that are sent or received through the **sendmsg** and **recvmsg** system calls. Optionally, a previous interface can be used that is based on the **setsockopt** system call. The control messages must be used as it reduces the number of system calls required.

## Control Message Interface

With the control message interface, the RDMA cookie is passed to the server out-of-band that is included in an extension header that is attached to the RDS message.

Initially, the client sends RDMA requests along with a `RDS_CMSG_RDMA_MAP` control message. The control message contains the address and length of the memory region to obtain a handle, flags, and a pointer to a memory location in the address space of the caller where the kernel stores the RDMA cookie.

If the application has an RDMA cookie for the memory range to or from an RDMA request, it can give this cookie to the kernel by using the `RDS_CMSG_RDMA_DEST` control message.

The kernel includes the resulting RDMA cookie in an extension header that is transmitted as part of the RDMA request to the server.

When the server receives the RDMA request, the kernel delivers the cookie within a `RDS_CMSG_RDMA_DEST` message. The server initiates the data transfer by sending the RDMA ACK message along with a `RDS_CMSG_RDMA_ARGS` control message. This message contains the RDMA cookie, and the local memory that can be copied.

The server process can request a notification when an RDMA operation completes. The notifications are delivered as the `RDS_CMSG_RDMA_STATUS` control messages. When an application calls the **recvmsg** call, it receives a regular RDS message with other RDMA-related control messages, or an empty message with one or more status control messages. When an RDMA operation fails and is discarded, the application can ask notifications for failed messages, regardless of the success notification of an individual message.

To activate the option for receiving failed notification, you must set the `RDS_RECVERR` socket option.

## Setsockopt Interface

A process can register and release memory ranges for RDMA through the **setsockopt** calls with the help of RDS.

### RDS\_GET\_MR

To obtain an RDMA cookie for a memory range, the application can use the **setsockopt** call with the `RDS_GET_MR` option. This cookie operates as the `RDS_CMSG_RDMA_MAP` control message. The argument contains the address and length of the memory range to be registered, and a pointer to an RDMA cookie variable where the system call stores the cookie for the registered range.

### RDS\_FREE\_MR

Memory ranges are released by calling the **setsockopt** call with the `RDS_FREE_MR` option. You can specify the RDMA cookie with flags as arguments.

### RDS\_RECVERR

This is a Boolean option that is set and queried by using the **getsockopt** call. When enabled, RDS sends RDMA notification messages to the application for any RDMA operation that fails. This option by default is set to off.

For all the calls, the level argument to the **setsockopt** call is `SOL_RDS`.

## RDMA Macros and types

### RDMA cookie

```
typedef u_int64_t      rds_rdma_cookie_t
```

This cookie contains a memory location in the client process. The cookie contains the R\_Key of the remote memory region, and the offset into it so that the alignment is not a concern for the application. The RDMA cookie is used in several struct types. The RDS\_CMSG\_RDMA\_DEST control message contains a `rds_rdma_cookie_t` as payload.

### Mapping arguments

The following data type is used with the RDS\_CMSG\_RDMA\_MAP control messages and with the RDS\_GET\_MR socket option:

```
struct rds_iovec {
    u_int64_t      addr;
    u_int64_t      bytes;
};

struct rds_get_mr_args {
    struct rds_iovec vec;
    u_int64_t      cookie_addr;
    u_int64_t      flags;
};
```

The `cookie_addr` parameter specifies a memory location to store the RDMA cookie.

The flags value is a bitwise OR of any of the following flags:

#### **RDS\_RDMA\_USE\_ONCE**

This flag specifies to the kernel that the allocated RDMA cookie must be used one time. When the RDMA ACK message is received, the kernel automatically unbinds the memory area and releases any resources that are associated with the cookie. If this flag is not set, the application must release the memory region by using the RDS\_FREE\_MR socket option.

#### **RDS\_RDMA\_INVALIDATE**

The RDMA memory mappings are not invalidated because it requires synchronization with the HCA, which is not cost effective. However, the server application can access the registered memory for any amount of time. The RDS code invalidates the mapping at the time it is released, and this can happen in two ways:

1. When an RDMA ACK and the RDS\_RDMA\_USE\_ONCE flag is set
2. When the application releases the memory by using the RDS\_FREE\_MR socket option.

### RDMA Operation

RDMA operations are initiated by the server by using the RDS\_CMSG\_RDMA\_ARGS control message, which takes the following data as payload:

```
struct rds_rdma_args {
    rds_rdma_cookie_t cookie;
    struct rds_iovec remote_vec;
    u_int64_t      local_vec_addr;
    u_int64_t      nr_local;
    u_int64_t      flags;
    u_int32_t      user_token;
};
```

The cookie argument contains the RDMA cookie received from the client. The local memory has an array of `rds_iovecs`. The array address is specified in the `local_vec_addr` option, and its number of elements is specified in the `nr_local` option. The struct member `remote_vec` specifies a location relative to the memory area that is identified by the `remote_vec.addr` cookie as an offset into that region, and `remote_vec.bytes` is the length of the memory window that can be copied. This length must match the size of the local memory area that is the sum of bytes in all members of the local `iovec` call. The flags field contains the bitwise or the following flags:

## **RDS\_RDMA\_READWRITE**

Performs an RDMA WRITE from the memory of the server to the client when the flag is set. If not set, RDS does an RDMA READ from the memory of the client to the memory of the server.

## **RDS\_RDMA\_FENCE**

The order of an RDMA READ in reference to the subsequent SEND operations is not decided by InfiniBand. When this flag is set, the RDMA READ is separated from the subsequent RDS ACK message. Setting this flag requires an additional round trip of the InfiniBand. Set this flag by default.

## **RDS\_RDMA\_NOTIFY\_ME**

This flag requests a notification on completion of the RDMA operation whether successful or otherwise. The notification contains the value of the `user_token` field that is passed by the application. This flag allows the application to release resources such as buffers that are associated with the RDMA transfer. The `user_token` can be used to pass an application-specific identifier to the kernel. This token is returned to the application when a status notification is generated.

## RDMA Notification

The RDS kernel code is able to notify the server application when an RDMA operation completes. These notifications are delivered through the `RDS_CMSG_RDMA_STATUS` control messages. By default, no notifications are generated. There are two ways an application can request for the messages. The status notifications can be enabled for every operation by setting the `RDS_RDMA_NOTIFY_ME` flag in the RDMA arguments. The application can request notifications for all RDMA operations that fail by setting the `RDS_RECVERR` socket option. In both cases, the format of the notification is the same and one notification is sent for the completed operation. The format of the message is as shown:

```
struct rds_rdma_notify {
    u_int32_t    user_token;
    int32_t     status;
};
```

The `user_token` field contains the value that was previously stored in the kernel in the `RDS_CMSG_RDMA_ARGS` control message. The status field contains a status value, with 0 indicating success, and non-zero indicating an error. The following status codes are defined:

## **RDS\_RDMA\_SUCCESS**

The RDMA operation succeeded.

## **RDS\_RDMA\_REMOTE\_ERROR**

The RDMA operation failed due to a remote access error. This error is because of an invalid `R_key`, offset, or transfer size.

## **RDS\_RDMA\_CANCELED**

The RDMA operation was canceled by the application.

## **RDS\_RDMA\_DROPPED**

RDMA operations was discarded after the connection failed and was reestablished. The RDMA operation is processed partially.

## **RDS\_RDMA\_OTHER\_ERROR**

Any other failure.

## RDMA setsockopt arguments

When you use the `RDS_GET_MR` socket option to register a memory range, the application passes a pointer to a `rds_get_mr_args` variable. The `RDS_FREE_MR` call accepts an argument of type `rds_free_mr_args` struct:

```
struct rds_free_mr_args {
    rds_rdma_cookie_t cookie;
    u_int64_t         flags;
};
```

Where `cookie` specifies the RDMA cookie to be released. RDMA access to the memory range is not received instantly because the operation is costly. However, if the `flags` argument contains `RDS_RDMA_INVALIDATE`, RDS invalidates the mapping immediately. If the `cookie` argument is 0, and `RDS_RDMA_INVALIDATE` is set, RDS invalidates old memory mappings on all devices.

## Errors

In addition to the usual error codes returned by **sendmsg**, **recvmsg** and **setsockopt** system calls, RDS returns the following error codes:

### EAGAIN

RDS was unable to map a memory range because the limit exceeded (returned by RDS\_CMSG\_RDMA\_MAP and RDS\_GET\_MR) .

### EINVAL

When a message is sent, there were conflicting control messages (For example, two RDMA\_MAP messages, or a RDMA\_MAP and a RDMA\_DEST message). In a RDS\_CMSG\_RDMA\_MAP or RDS\_GET\_MR operation, the application that is specified by the memory range is greater than the maximum size supported. The size of the local memory specified in the `rds_iovec` call does not match the size of the remote memory range when an RDMA operation with the RDS\_CMSG\_RDMA\_ARGS was set up.

### EBUSY

RDS was unable to obtain a DMA mapping for the indicated memory.

## ReadFile Subroutine

### Purpose

Reads data from a socket.

### Syntax

```
#include <iocp.h>
boolean_t ReadFile (FileDescriptor, Buffer, ReadCount, AmountRead, Overlapped)
HANDLE FileDescriptor;
LPVOID Buffer;
DWORD ReadCount;
LPDWORD AmountRead;
LPOVERLAPPED Overlapped;
```

### Description

The **ReadFile** subroutine reads the number of bytes specified by the *ReadCount* parameter from the *FileDescriptor* parameter into the buffer indicated by the *Buffer* parameter. The number of bytes read is saved in the *AmountRead* parameter. The *Overlapped* parameter indicates whether or not the operation can be handled asynchronously.

The **ReadFile** subroutine returns a boolean (an integer) indicating whether or not the request has been completed.

The **ReadFile** subroutine is part of the I/O Completion Port (IOCP) kernel extension.

**Note:** This subroutine only works to a socket file descriptor. It does not work with files or other file descriptors.

### Parameters

Item	Description
<i>FileDescriptor</i>	Specifies a valid file descriptor obtained from a call to the <b>socket</b> or <b>accept</b> subroutines.
<i>Buffer</i>	Specifies the buffer from which the data will be read.
<i>ReadCount</i>	Specifies the maximum number of bytes to read.
<i>AmountRead</i>	Specifies the number of bytes read. The parameter is set by the subroutine.

Item	Description
<i>Overlapped</i>	Specifies an overlapped structure indicating whether or not the request can be handled asynchronously.

## Return Values

Upon successful completion, the **ReadFile** subroutine returns a boolean indicating the request has been completed.

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

- Returns a value of 0 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable, see the link in the Related Information section of this document.

## Error Codes

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

Item	Description
<b>EINPROGRESS</b>	The read request can not be immediately satisfied and will be handled asynchronously. A completion packet will be sent to the associated completion port upon completion.
<b>EAGAIN</b>	The read request cannot be immediately satisfied and cannot be handled asynchronously.
<b>EINVAL</b>	The <i>FileDescriptor</i> parameter is invalid.

## Examples

The following program fragment illustrates the use of the **ReadFile** subroutine to synchronously read data from a socket:

```
void buffer;
int amount_read;
b = ReadFile (34, &buffer, 128, &amount_read, NULL);
```

The following program fragment illustrates the use of the **ReadFile** subroutine to asynchronously read data from a socket:

```
void buffer;
int amount_read;
LPOVERLAPPED overlapped;
b = ReadFile (34, &buffer, 128, &amount_read, overlapped);
```

**Note:** The request will only be handled asynchronously if it cannot be immediately satisfied.

### Related information

[Error Notification Object Class](#)

## recv Subroutine

### Purpose

Receives messages from connected sockets.

### Library

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

## Syntax

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

```
int recv (Socket,  
Buffer, Length, Flags)  
int Socket;  
void * Buffer;  
size_t Length;  
int Flags;
```

## Description

The **recv** subroutine receives messages from a connected socket. The **recvfrom** and **recvmsg** subroutines receive messages from both connected and unconnected sockets. However, they are usually used for unconnected sockets only.

The **recv** subroutine returns the length of the message. If a message is too long to fit in the supplied buffer, excess bytes may be truncated depending on the type of socket that issued the message.

If no messages are available at the socket, the **recv** subroutine waits for a message to arrive, unless the socket is nonblocking. If a socket is nonblocking, the system returns an error.

Use the **select** subroutine to determine when more data arrives.

The socket applications can be compiled with **COMPAT\_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

## Parameters

Item	Description
------	-------------

<i>Socket</i>	Specifies the socket descriptor.
---------------	----------------------------------

<i>Buffer</i>	Specifies an address where the message should be placed.
---------------	--

<i>Length</i>	Specifies the size of the <i>Buffer</i> parameter.
---------------	--

<i>Flags</i>	Points to a value controlling the message reception. The <b>/usr/include/sys/socket.h</b> file defines the <i>Flags</i> parameter. The argument to receive a call is formed by logically ORing one or more of the following values:
--------------	---

### **MSG\_OOB**

Processes out-of-band data. The significance of out-of-band data is protocol-dependent.

### **MSG\_PEEK**

Peeks at incoming data. The data continues to be treated as unread and will be read by the next call to **recv()** or a similar function.

### **MSG\_WAITALL**

Requests that the function not return until the requested number of bytes have been read. The function can return fewer than the requested number of bytes only if a signal is caught, the connection is terminated, or an error is pending for the socket.

## Return Values

Upon successful completion, the **recv** subroutine returns the length of the message in bytes.

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

- Returns a value of -1 to the calling program.
- Returns a 0 if the connection disconnects.
- Moves an error code, indicating the specific error, into the **errno** global variable.

## Error Codes

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

Error	Description
<b>EBADF</b>	The <i>Socket</i> parameter is not valid.
<b>ECONNRESET</b>	The remote peer forces the connection to be closed.
<b>EFAULT</b>	The data was directed to be received into a nonexistent or protected part of the process address space. The <i>Buffer</i> parameter is not valid.
<b>EINTR</b>	A signal interrupted the <b>recv</b> subroutine before any data was available.
<b>EINVAL</b>	The <b>MSG_OOB</b> flag is set and no out-of-band data is available.
<b>ENOBUF</b>	Insufficient resources are available in the system to perform the operation.
<b>ENOTCONN</b>	A receive is attempted on a <b>SOCK_STREAM</b> socket that is not connected.
<b>ENOTSOCK</b>	The <i>Socket</i> parameter refers to a file, not a socket.
<b>EOPNOTSUPP</b>	<b>MSG_OOB</b> flag is set for a <b>SOCK_DGRAM</b> socket, or <b>MSG_OOB</b> flag is set for any <b>AF_UNIX</b> socket.
<b>ETIMEDOUT</b>	The connection timed out during connection establishment, or there was a transmission timeout on an active connection.
<b>EWouldBlock</b>	The socket is marked nonblocking, and no connections are present to be accepted.

### Related reference

[recvmsg Subroutine](#)

[recvfrom Subroutine](#)

[shutdown Subroutine](#)

### Related information

[fgets subroutine](#)

[read subroutine](#)

[Sockets Overview](#)

## recvfrom Subroutine

### Purpose

Receives messages from sockets.

### Library

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

### Syntax

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

```
ssize_t recvfrom  
(Socket, Buffer, Length, Flags, From, FromLength)  
int Socket;  
void * Buffer;  
size_t Length,  
int Flags;  
struct sockaddr * From;  
socklen_t * FromLength;
```

## Description

The **recvfrom** subroutine allows an application program to receive messages from unconnected sockets. The **recvfrom** subroutine is normally applied to unconnected sockets as it includes parameters that allow the calling program to specify the source point of the data to be received.

To return the source address of the message, specify a nonnull value for the *From* parameter. The *FromLength* parameter is a value-result parameter, initialized to the size of the buffer associated with the *From* parameter. On return, the **recvfrom** subroutine modifies the *FromLength* parameter to indicate the actual size of the stored address. The **recvfrom** subroutine returns the length of the message. If a message is too long to fit in the supplied buffer, excess bytes may be truncated depending on the type of socket that issued the message.

If no messages are available at the socket, the **recvfrom** subroutine waits for a message to arrive, unless the socket is nonblocking. If the socket is nonblocking, the system returns an error.

The socket applications can be compiled with **COMPAT\_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

## Parameters

Item	Description
<i>Socket</i>	Specifies the socket descriptor.
<i>Buffer</i>	Specifies an address where the message should be placed.
<i>Length</i>	Specifies the size of the <i>Buffer</i> parameter.
<i>Flags</i>	Points to a value controlling the message reception. The argument to receive a call is formed by logically ORing one or more of the values shown in the following list: <b>MSG_OOB</b> Processes out-of-band data. The significance of out-of-band data is protocol-dependent. <b>MSG_PEEK</b> Peeks at incoming data. The data continues to be treated as unread and will be read by the next call to <b>recv()</b> or a similar function. <b>MSG_WAITALL</b> Requests that the function not return until the requested number of bytes have been read. The function can return fewer than the requested number of bytes only if a signal is caught, the connection is terminated, or an error is pending for the socket.
<i>From</i>	Points to a socket structure, filled in with the source's address.
<i>FromLength</i>	Specifies the length of the sender's or source's address.

## Return Values

If the **recvfrom** subroutine is successful, the subroutine returns the length of the message in bytes.

If the call is unsuccessful, the subroutine handler performs the following functions:

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

## Error Codes

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

Error	Description
<b>EBADF</b>	The <i>Socket</i> parameter is not valid.
<b>ECONNRESET</b>	The remote peer forces the connection to be closed.
<b>EFAULT</b>	The data was directed to be received into a nonexistent or protected part of the process address space. The buffer is not valid.
<b>EINTR</b>	The receive is interrupted by a signal delivery before any data is available.
<b>EINVAL</b>	The <b>MSG_OOB</b> flag is set but no out-of-band data is available.
<b>ENOBUF</b>	Insufficient resources are available in the system to perform the operation.
<b>ENOPROTOPT</b>	The protocol is not 64-bit supported.
<b>ENOTCONN</b>	A receive is attempted on a <b>SOCK_STREAM</b> socket that is not connected.
<b>ENOTSOCK</b>	The <i>Socket</i> parameter refers to a file, not a socket.
<b>EOPNOTSUPP</b>	<b>MSG_OOB</b> flag is set for a <b>SOCK_DGRAM</b> socket, or <b>MSG_OOB</b> flag is set for any <b>AF_UNIX</b> socket.
<b>ETIMEDOUT</b>	The connection timed out during connection establishment, or there was a transmission timeout on an active connection.
<b>EWouldBlock</b>	The socket is marked nonblocking, and no connections are present to be accepted.

#### Related reference

[recv Subroutine](#)

#### Related information

[fgets subroutine](#)

[select subroutine](#)

[Sockets Overview](#)

[Understanding Socket Data Transfer](#)

## recvmsg Subroutine

### Purpose

Receives a message from any socket.

### Library

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

### Syntax

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

```
int recvmsg ( Socket, Message, Flags)
int Socket;
struct msghdr Message [ ];
int Flags;
```

```
int recvmmsg ( Socket, MessageVec, Num_msg, Flags, Timeout)
int Socket;
struct mmsghdr MessageVec [ ];
unsigned int Num_msg ;
int Flags;
struct timespec *Timeout
```

## Description

The **recvmsg** subroutine receives messages from unconnected or connected sockets. The **recvmsg** subroutine returns the length of the message. If a message is too long to fit in the supplied buffer, excess bytes may be truncated depending on the type of socket that issued the message.

If no messages are available at the socket, the **recvmsg** subroutine waits for a message to arrive. If the socket is nonblocking and no messages are available, the **recvmsg** subroutine is unsuccessful.

Use the **select** subroutine to determine when more data arrives.

The **recvmsg** subroutine uses a **msghdr** structure to decrease the number of directly supplied parameters. The **msghdr** structure is defined in the **sys/socket.h** file. In BSD 4.3 Reno, the size and members of the **msghdr** structure have been modified. Applications wanting to start the old structure need to compile with **COMPAT\_43** defined. The default behavior is that of BSD 4.4.

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

The **recvmsg** subroutine is an extension of the **recvmsg** subroutine that receives multiple messages from a socket to the caller socket. This subroutine has performance benefits for some applications. The **recvmsg** subroutine supports timeout for wait during the receive operation.

The **sockfd** argument is the file descriptor of the socket from which data is received. The **msgvec** argument is a pointer to an array of **msgvec** structures. These arguments are defined in the **sys/socket.h** file.

On return from the **recvmsg** subroutine, successive elements of the **msgvec** structure are updated to contain information about each received message. The **msg\_len** field contains the size of the received message. The sub fields of the **msg\_hdr** field are updated as described in the **recvmsg** subroutine. The return value of the **recvmsg** call indicates the number of elements of the **msgvec** field that are updated.

## Parameters

Item	Description
<i>Socket</i>	Specifies the unique name of the socket.
<i>Message</i>	Points to the address of the <b>msghdr</b> structure, which contains both the address for the incoming message and the space for the sender address.
<i>Flags</i>	Permits the subroutine to exercise control over the reception of messages. The <i>Flags</i> parameter that is used to receive a call is formed by logically ORing one or more of the values which are shown in the following list: <b>MSG_OOB</b> Processes out-of-band data. The significance of out-of-band data is protocol-dependent. <b>MSG_PEEK</b> Peeks at incoming data. The data continues to be treated as unread and will be read by the next call to <b>recv()</b> or a similar function. <b>MSG_WAITALL</b> Requests that the function not return until the requested number of bytes have been read. The function can return fewer than the requested number of bytes only if a signal is caught, the connection is terminated, or an error is pending for the socket. <b>MSG_WAITFORONE</b> Turns on the <b>MSG_DONTWAIT</b> flag after the first message is received.

The **/sys/socket.h** file contains the possible values for the *Flags* parameter.

Item	Description
MessageVec	Points to an array of <code>mmsg_hdr</code> structures, which contain <code>msg_hdr</code> structures for incoming messages, space for the sender address and a value that represents the total number of elements in the array.
Num_msg	Defines the number of messages to receive before the control is returned to the calling socket.
Timeout	The <i>timeout</i> argument points to a <code>timespec</code> structure that defines a timeout value (specified in seconds plus nanoseconds) for the receive operation. If the timeout value is NULL a call to the <code>recvmsg</code> subroutine is blocked until the <code>vlen</code> messages are received or until the timeout value expires. A nonblocking call to the <code>recvmsg</code> subroutine reads all messages that are available (the limit is specified by the <code>vlen</code> parameter) at the sender socket and returns from the subroutine to the calling function immediately.

## Return Values

Upon successful completion of `recvmsg` subroutine, the length of the message in bytes is returned and for the `recvmsg` subroutine, the number of received messages is returned.

If the `recvmsg` or the `recvmsg` subroutine is unsuccessful, the subroutine handler performs the following functions:

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

## Error Codes

The `recvmsg` subroutine is unsuccessful if any of the following error codes occurs:

Error	Description
<b>EBADF</b>	The <i>Socket</i> parameter is not valid.
<b>ECONNRESET</b>	The remote peer forces the connection to be closed.
<b>EFAULT</b>	The <i>Address</i> parameter is not in a writable part of the user address space.
<b>EINTR</b>	The <code>recvmsg</code> subroutine was interrupted by delivery of a signal before any data was available for the receive.
<b>EINVAL</b>	The length of the <code>msg_hdr</code> structure is invalid, or the <b>MSG_OOB</b> flag is set and no out-of-band data is available.
<b>EMSGSIZE</b>	The <i>msg_iovlen</i> member of the <code>msg_hdr</code> structure pointed to by <i>Message</i> is less than or equal to 0, or is greater than <b>IOV_MAX</b> .
<b>ENOBUF</b>	Insufficient resources are available in the system to perform the operation.
<b>ENOPROTOPT</b>	The protocol is not 64-bit supported.
<b>ENOTCONN</b>	A receive is attempted on a <b>SOCK_STREAM</b> socket that is not connected.
<b>ENOTSOCK</b>	The <i>Socket</i> parameter refers to a file, not a socket.
<b>EOPNOTSUPP</b>	<b>MSG_OOB</b> flag is set for a <b>SOCK_DGRAM</b> socket, or <b>MSG_OOB</b> flag is set for any <b>AF_UNIX</b> socket.
<b>ETIMEDOUT</b>	The connection timed out during connection establishment, or there was a transmission timeout on an active connection.
<b>EWouldBlock</b>	The socket is marked nonblocking, and no connections are present to be accepted.

## Related reference

[recv Subroutine](#)

## Related information

[no subroutine](#)

[select subroutine](#)

[Sockets Overview](#)

# res\_init Subroutine

## Purpose

Searches for a default domain name and Internet address.

## Library

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

## Syntax

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

void res_init ( )
```

## Description

The **res\_init** subroutine reads the **/etc/resolv.conf** file for the default domain name and the Internet address of the initial hosts running the name server.

**Note:** If the **/etc/resolv.conf** file does not exist, the **res\_init** subroutine attempts name resolution using the local **/etc/hosts** file. If the system is not using a domain name server, the **/etc/resolv.conf** file should not exist. The **/etc/hosts** file should be present on the system even if the system is using a name server. In this instance, the file should contain the host IDs that the system requires to function even if the name server is not functioning.

The **res\_init** subroutine is one of a set of subroutines that form the resolver, a set of functions that translate domain names to Internet addresses. All resolver subroutines use the **/usr/include/resolv.h** file, which defines the **\_res** structure. The **res\_init** subroutine stores domain name information in the **\_res** structure. Three environment variables, **LOCALDOMAIN**, **RES\_TIMEOUT**, and **RES\_RETRY**, affect default values related to the **\_res** structure.

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

For more information on the **\_res** structure, see "Understanding Domain Name Resolution" in *Communications Programming Concepts*.

## Files

Item	Description
<b><u>resolv.conf</u></b>	<b><u>/etc/</u></b> Contains the name server and domain name.

Item	Description
<a href="#">/etc/hosts</a>	Contains host names and their addresses for hosts in a network. This file is used to resolve a host name into an Internet address.

### Related information

[Sockets Overview](#)

[Understanding Domain Name Resolution](#)

## res\_mkquery Subroutine

### Purpose

Makes query messages for name servers.

### Library

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

### Syntax

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

```
int res_mkquery (Operation, DomName, Class, Type, Data, DataLength)
int res_mkquery (Reserved, Buffer, BufferLength)
int Operation;
char * DomName;
int Class, Type;
char * Data;
int DataLength;
struct rrec * Reserved;
char * Buffer;
int BufferLength;
```

### Description

The **res\_mkquery** subroutine creates packets for name servers in the Internet domain. The subroutine also creates a standard query message. The *Buffer* parameter determines the location of this message.

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

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

### Parameters

Item	Description
<i>Operation</i>	Specifies a query type. The usual type is <b>QUERY</b> , but the parameter can be set to any of the query types defined in the <b>arpa/nameser.h</b> file.

<b>Item</b>	<b>Description</b>
<i>DomName</i>	Points to the name of the domain. If the <i>DomName</i> parameter points to a single label and the <b>RES_DEFNAMES</b> structure is set, as it is by default, the subroutine appends the <i>DomName</i> parameter to the current domain name. The current domain name is defined by the name server in use or in the <u><a href="#">/etc/resolv.conf</a></u> file.
<i>Class</i>	Specifies one of the following parameters: <b>C_IN</b> Specifies the ARPA Internet. <b>C_CHAOS</b> Specifies the Chaos network at MIT.
<i>Type</i>	Requires one of the following values: <b>T_A</b> Host address <b>T_NS</b> Authoritative server <b>T_MD</b> Mail destination <b>T_MF</b> Mail forwarder <b>T_CNAME</b> Canonical name <b>T_SOA</b> Start-of-authority zone <b>T_MB</b> Mailbox-domain name <b>T_MG</b> Mail-group member <b>T_MR</b> Mail-rename name <b>T_NULL</b> Null resource record <b>T_WKS</b> Well-known service <b>T_PTR</b> Domain name pointer <b>T_HINFO</b> Host information <b>T_MINFO</b> Mailbox information <b>T_MX</b> Mail-routing information <b>T_UINFO</b> User ( <u><a href="#">finger</a></u> command) information <b>T_UID</b> User ID <b>T_GID</b> Group ID

Item	Description
<i>Data</i>	Points to the data that is sent to the name server as a search key. The data is stored as a character array.
<i>DataLength</i>	Defines the size of the array pointed to by the <i>Data</i> parameter.
<i>Reserved</i>	Specifies a reserved and currently unused parameter.
<i>Buffer</i>	Points to a location containing the query message.
<i>BufferLength</i>	Specifies the length of the message pointed to by the <i>Buffer</i> parameter.

## Return Values

Upon successful completion, the **res\_mkquery** subroutine returns the size of the query. If the query is larger than the value of the *BufferLength* parameter, the subroutine is unsuccessful and returns a value of -1.

## Files

Item	Description
<u><a href="#">/etc/resolv.conf</a></u>	Contains the name server and domain name.

## Related information

[finger subroutine](#)

[Sockets Overview](#)

[Understanding Domain Name Resolution](#)

## res\_ninit Subroutine

### Purpose

Sets the default values for the members of the **\_res** structure.

### Library

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

### Syntax

```
#include <resolv.h>

int res_ninit (statp)
res_state statp;
```

### Description

Reads the **/etc/resolv.conf** configuration file to get the default domain name, search list, and internet address of the local name server(s). It does this in order to re-initialize the resolver context for a given thread in a multi-threaded environment.

The **res\_ninit** subroutine sets the default values for the members of the **\_res** structure (defined in the **/usr/include/resolv.h** file) after reading the **/etc/resolv.conf** configuration file to get default domain name, search list, Internet address of the local name server(s), sort list, and options (for details, please refer to the **/etc/resolv.conf** file). If no name server is configured, the server address is set to INADDR\_ANY and the default domain name is obtained from the **gethostname** subroutine. It also

allows the user to override retrans, retry, and local domain definition using three environment variables RES\_TIMEOUT, RES\_RETRY, and LOCALDOMAIN, respectively.

Using this subroutine, each thread can have unique local resolver context. Since the configuration file is read each time the subroutine is called, it is capable of tracking dynamic changes to the resolver state file. Changes include, addition or removal of the configuration file or any other modifications to this file and reflect the same for a given thread. The **res\_ninit** subroutine can also be used in single-threaded applications to detect dynamic changes to the resolver file even while the program is running (See the example section below). For more information on the **\_res** structure, see [Understanding Domain Name Resolution in AIX Version 6.1 Communications Programming Concepts](#).

## Parameters

Item	Description
<i>statp</i>	Specifies the state to be initialized.

## Examples

```
# cat /etc/resolv.conf
domain in.ibm.com
nameserver 9.184.192.240
```

The following two examples use the **gethostbyname** system call to retrieve the host address of a system (florida.in.ibm.com) continuously. In the first example, **gethostbyname** is called (by a thread 'resolver') in a multi-threaded environment. The second example is not. Before each call to **gethostbyname**, the **res\_ninit** subroutine is called to reflect dynamic changes to the configuration file.

```
1) #include <stdio.h>
#include <netdb.h>
#include <resolv.h>
#include <pthread.h>

void *resolver (void *arg);
main( ) {
    pthread_t thid;
    if ( pthread_create(&thid, NULL, resolver, NULL) ) {
        printf("error in thread creation\n");
        exit( ); }
    pthread_exit(NULL);
}

void *resolver (void *arg) {
    struct hostent *hp;
    struct sockaddr_in client;
    while(1) {
        res_ninit(&_res);          /* res_init() with RES_INIT unset would NOT work
here */

        hp = (struct hostent * ) gethostbyname("florida.in.ibm.com");
        bcopy(hp->h_addr_list[0], &client.sin_addr, sizeof(client.sin_addr));
        printf("hostname: %s\n", inet_ntoa(client.sin_addr));
    }
}
```

If the **/etc/resolv.conf** file is present when the thread 'resolver' is invoked, the hostname will be resolved for that thread (using the nameserver 9.184.192.210) and the output will be hostname: 9.182.21.151.

If **/etc/resolv.conf** is not present, the output will be hostname: 0.0.0.0.

```
2) The changes to /etc/resolv.conf file are reflected even while the program is running

#include <stdio.h>
#include <resolv.h>
#include <sys.h>
#include <netdb.h>
#include <string.h>
```

```

main() {
    struct hostent *hp;
    struct sockaddr_in client;

    while (1) {
        res_ninit(&_res);

        hp = (struct hostent * ) gethostbyname("florida.in.ibm.com");
        bcopy(hp->h_addr_list[0], &client.sin_addr, sizeof(client.sin_addr));
        printf("hostname: %s\n", inet_ntoa(client.sin_addr));
    }
}

```

If **/etc/resolv.conf** is present while the program is running, the hostname will be resolved (using the nameserver 9.184.192.240) and the output will be hostname: 9.182.21.151.

If the **/etc/resolv.conf** file is not present, the output of the program will be hostname: 0.0.0.0.

**Note:** In the second example, the **res\_init** subroutine with `_res.options = ~RES_INIT` can be used instead of the **res\_ninit** subroutine.

## Files

The **/etc/resolv.conf** and **/etc/hosts** files.

## Related information

[Understanding Domain Name Resolution](#)

## res\_query Subroutine

### Purpose

Provides an interface to the server query mechanism.

### Library

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

### Syntax

```

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

```

```

int res_query (DomName, Class, Type, Answer, AnswerLength)
char * DomName;
int Class;
int Type;
u_char * Answer;
int AnswerLength;

```

### Description

The **res\_query** subroutine provides an interface to the server query mechanism. It constructs a query, sends it to the local server, awaits a response, and makes preliminary checks on the reply. The query requests information of the specified type and class for the fully-qualified domain name specified in the *DomName* parameter. The reply message is left in the answer buffer whose size is specified by the *AnswerLength* parameter, which is supplied by the caller.

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

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

## Parameters

Item	Description
<i>DomName</i>	Points to the name of the domain. If the <i>DomName</i> parameter points to a single-component name and the <b>RES_DEFNAMES</b> structure is set, as it is by default, the subroutine appends the default domain name to the single-component name. The current domain name is defined by the name server in use or is specified in the <u><b>/etc/resolv.conf</b></u> file.
<i>Class</i>	Specifies one of the following values: <b>C_IN</b> Specifies the ARPA Internet. <b>C_CHAOS</b> Specifies the Chaos network at MIT.

**Item***Type***Description**

Requires one of the following values:

**T\_A**

Host address

**T\_NS**

Authoritative server

**T\_MD**

Mail destination

**T\_MF**

Mail forwarder

**T\_CNAME**

Canonical name

**T\_SOA**

Start-of-authority zone

**T\_MB**

Mailbox-domain name

**T\_MG**

Mail-group member

**T\_MR**

Mail-rename name

**T\_NULL**

Null resource record

**T\_WKS**

Well-known service

**T\_PTR**

Domain name pointer

**T\_HINFO**

Host information

**T\_MINFO**

Mailbox information

**T\_MX**

Mail-routing information

**T\_UINFO**User (finger command) information**T\_UID**

User ID

**T\_GID**

Group ID

*Answer*

Points to an address where the response is stored.

*AnswerLength*

Specifies the size of the answer buffer.

**Return Values**

Upon successful completion, the **res\_query** subroutine returns the size of the response. Upon unsuccessful completion, the **res\_query** subroutine returns a value of -1 and sets the **h\_errno** value to the appropriate error.

## Files

### Item

[/etc/resolv.conf](#)

### Description

Contains the name server and domain name.

## Related information

[finger subroutine](#)

[Sockets Overview](#)

[Understanding Domain Name Resolution](#)

## res\_search Subroutine

### Purpose

Makes a query and awaits a response.

### Library

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

### Syntax

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

```
int res_search (DomName, Class, Type, Answer, AnswerLength)
char * DomName;
int Class;
int Type;
u_char * Answer;
int AnswerLength;
```

### Description

The **res\_search** subroutine makes a query and awaits a response like the **res\_query** subroutine. However, it also implements the default and search rules controlled by the **RES\_DEFNAMES** and **RES\_DNSRCH** options.

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

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

## Parameters

### Item

*DomName*

### Description

Points to the name of the domain. If the *DomName* parameter points to a single-component name and the **RES\_DEFNAMES** structure is set, as it is by default, the subroutine appends the default domain name to the single-component name. The current domain name is defined by the name server in use or is specified in the [/etc/resolv.conf](#) file.

If the **RES\_DNSRCH** bit is set, as it is by default, the **res\_search** subroutine searches for host names in both the current domain and in parent domains.

*Class*

Specifies one of the following values:

#### **C\_IN**

Specifies the ARPA Internet.

#### **C\_CHAOS**

Specifies the Chaos network at MIT.

**Item***Type***Description**

Requires one of the following values:

**T\_A**

Host address

**T\_NS**

Authoritative server

**T\_MD**

Mail destination

**T\_MF**

Mail forwarder

**T\_CNAME**

Canonical name

**T\_SOA**

Start-of-authority zone

**T\_MB**

Mailbox-domain name

**T\_MG**

Mail-group member

**T\_MR**

Mail-rename name

**T\_NULL**

Null resource record

**T\_WKS**

Well-known service

**T\_PTR**

Domain name pointer

**T\_HINFO**

Host information

**T\_MINFO**

Mailbox information

**T\_MX**

Mail-routing information

**T\_UINFO**User (finger command) information**T\_UID**

User ID

**T\_GID**

Group ID

*Answer*

Points to an address where the response is stored.

*AnswerLength*

Specifies the size of the answer buffer.

**Return Values**

Upon successful completion, the **res\_search** subroutine returns the size of the response. Upon unsuccessful completion, the **res\_search** subroutine returns a value of -1 and sets the **h\_errno** value to the appropriate error.

## Files

### Item

[/etc/resolv.conf](#)

### Description

Contains the name server and domain name.

## Related information

[finger subroutine](#)

[Sockets Overview](#)

[Understanding Domain Name Resolution](#)

## res\_send Subroutine

### Purpose

Sends a query to a name server and retrieves a response.

### Library

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

### Syntax

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

```
int res_send (MessagePtr, MessageLength, Answer, AnswerLength)
char * MsgPtr;
int MsgLength;
char * Answer;
int AnswerLength;
```

### Description

The **res\_send** subroutine sends a query to name servers and calls the **res\_init** subroutine if the **RES\_INIT** option of the **\_res** structure is not set. This subroutine sends the query to the local name server and handles time outs and retries.

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

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

### Parameters

#### Item

#### Description

*MessagePtr*

Points to the beginning of a message.

*MessageLength*

Specifies the length of the message.

*Answer*

Points to an address where the response is stored.

*AnswerLength*

Specifies the size of the answer area.

## Return Values

Upon successful completion, the **res\_send** subroutine returns the length of the message.

If the **res\_send** subroutine is unsuccessful, the subroutine returns a -1.

## Files

Item	Description
<u><a href="#">/etc/resolv.conf</a></u>	Contains general name server and domain name information.

## Related information

[Sockets Overview](#)

[Understanding Domain Name Resolution](#)

## rexec Subroutine

### Purpose

Allows command execution on a remote host.

### Library

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

### Syntax

```
int rexec ( Host, Port, User, Passwd, Command, ErrFileDescParam)
char **Host;
int Port;
char *User, *Passwd,
*Command;
int *ErrFileDescParam;
```

### Description

The **rexec** subroutine allows the calling process to start commands on a remote host.

If the **rexec** connection succeeds, a socket in the Internet domain of type **SOCK\_STREAM** is returned to the calling process and is given to the remote command as standard input and standard output.

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

### Parameters

Item	Description
<i>Host</i>	Contains the name of a remote host that is listed in the <u><a href="#">/etc/hosts</a></u> file or <u><a href="#">/etc/resolv.config</a></u> file. If the name of the host is not found in either file, the <b>rexec</b> subroutine is unsuccessful.

<b>Item</b>	<b>Description</b>
<i>Port</i>	<p>Specifies the well-known DARPA Internet port to use for the connection. A pointer to the structure that contains the necessary port can be obtained by issuing the following library call:</p> <pre style="background-color: #f0f0f0; padding: 5px;">getservbyname("exec", "tcp")</pre>
<i>User and Passwd</i>	<p>Points to a user ID and password valid at the host. If these parameters are not supplied, the <b>rexec</b> subroutine takes the following actions until finding a user ID and password to send to the remote host:</p> <ol style="list-style-type: none"> <li>1. Searches the current environment for the user ID and password on the remote host.</li> <li>2. Searches the user's home directory for a file called <b>\$HOME/.netrc</b> that contains a user ID and password.</li> <li>3. Prompts the user for a user ID and password.</li> </ol>
<i>Command</i>	Points to the name of the command to be executed at the remote host.
<i>ErrFileDescParam</i>	<p>Specifies one of the following values:</p> <p><b>Non-zero</b></p> <p>Indicates an auxiliary channel to a control process is set up, and a descriptor for it is placed in the <i>ErrFileDescParam</i> parameter. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command. This diagnostic information does not include remote authorization failure, since this connection is set up after authorization has been verified.</p> <p><b>0</b></p> <p>Indicates the standard error of the remote command is the same as standard output, and no provision is made for sending arbitrary signals to the remote process. In this case, however, it may be possible to send out-of-band data to the remote command.</p>

## Return Values

Upon successful completion, the system returns a socket to the remote command.

If the **rexec** subroutine is unsuccessful, the system returns a -1 indicating that the specified host name does not exist.

## Files

<b>Item</b>	<b>Description</b>
<u><b>/etc/hosts</b></u>	Contains host names and their addresses for hosts in a network. This file is used to resolve a host name into an Internet address.
<u><b>/etc/resolv.conf</b></u>	Contains the name server and domain name.
<u><b>\$HOME/.netrc</b></u>	Contains automatic login information.

## Related information

[Transmission Control Protocol/Internet Protocol](#)  
[Sockets Overview](#)

## rexec\_af Subroutine

### Purpose

Allows command execution on a remote host.

### Syntax

```
int rexec_af(char **ahost, unsigned short rport, const char *name,  
             const char *pass, const char *cmd, int *fd2p, int af)
```

### Description

The **rexec\_af** subroutine allows the calling process to start commands on a remote host. It behaves the same as the existing **rexec()** function, but instead of creating only an AF\_INET TCP socket, it can also create an AF\_INET6 TCP socket.

The **rexec\_af** subroutine is useful because the existing **rexec()** function cannot transparently use AF\_INET6 sockets. This is because an application would not be prepared to handle AF\_INET6 addresses returned by functions such as **getpeername()** on the file descriptor created by **rexec()**.

If the **rexec\_af** connection succeeds, a socket in the Internet domain of type **SOCK\_STREAM** is returned to the calling process and is given to the remote command as standard input and standard output.

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

### Parameters

Item	Description
<i>ahost</i>	Contains the name of a remote host that is listed in the <b>/etc/hosts</b> file or <b>/etc/resolv.conf</b> file. If the name of the host is not found in either file, the <b>rexec</b> subroutine is unsuccessful.
<i>rport</i>	Specifies the well-known DARPA Internet port to use for the connection. A pointer to the structure that contains the necessary port can be obtained by issuing the following library call: <pre>getservbyname("exec", "tcp")</pre>
<i>name</i> and <i>pass</i>	Points to a valid user ID and password at the host. If these parameters are not supplied, the <b>rexec_af</b> subroutine takes the following actions until it finds a user ID and password to send to the remote host: <ol style="list-style-type: none"><li>1. Searches the current environment for the user ID and password on the remote host.</li><li>2. Searches the user's home directory for a file called <b>\$HOME/.netrc</b> that contains a user ID and password.</li><li>3. Prompts the user for a user ID and password.</li></ol>
<i>cmd</i>	Points to the name of the command to be executed at the remote host.

<b>Item</b>	<b>Description</b>
<i>fd2p</i>	<p>Specifies one of the following values:</p> <p><b>Non-zero</b> Indicates that an auxiliary channel to a control process is set up, and a descriptor for it is placed in the <i>fd2p</i> parameter. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command. This diagnostic information does not include remote authorization failure, since this connection is set up after authorization has been verified.</p> <p><b>0</b> Indicates that the standard error of the remote command is the same as standard output, and no provision is made for sending arbitrary signals to the remote process. In this case, however, it might be possible to send out-of-band data to the remote command.</p>
<i>af</i>	<p>The family argument is AF_INET, AF_INET6, or AF_UNSPEC. When either AF_INET or AF_INET6 is specified, this subroutine will create a socket of the specified address family. When AF_UNSPEC is specified, it will try all possible address families until a connection can be established, and will return the associated socket of the connection.</p>

## Return Values

Upon successful completion, the system returns a socket to the remote command. If the **rexec\_af** subroutine is unsuccessful, the system returns a `-1`, indicating that the specified host name does not exist.

## Files

<b>Item</b>	<b>Description</b>
<u><a href="#">/etc/hosts</a></u>	Contains host names and their addresses for hosts in a network. This file is used to resolve a host name into an Internet address.
<u><a href="#">/etc/resolv.conf</a></u>	Contains the name server and domain name.
<u><a href="#">\$HOME/.netrc</a></u>	Contains automatic login information.

## rresvport Subroutine

### Purpose

Retrieves a socket with a privileged address.

### Library

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

### Syntax

```
int rresvport ( Port)
int *Port;
```

## Description

The **rresvport** subroutine obtains a socket with a privileged address bound to the socket. A privileged Internet port is one that falls in a range between 0 and 1023.

Only processes with an effective user ID of root user can use the **rresvport** subroutine. An authentication scheme based on remote port numbers is used to verify permissions.

If the connection succeeds, a socket in the Internet domain of type **SOCK\_STREAM** is returned to the calling process.

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

## Parameters

### Item Description

*Port* Specifies the port to use for the connection.

## Return Values

Upon successful completion, the **rresvport** subroutine returns a valid, bound socket descriptor.

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

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

## Error Codes

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

Error	Description
<b>EAGAIN</b>	All network ports are in use.
<b>EAFNOSUPPORT</b>	The addresses in the specified address family cannot be used with this socket.
<b>EMFILE</b>	Two hundred file descriptors are currently open.
<b>ENFILE</b>	The system file table is full.
<b>ENOBUFS</b>	Insufficient buffers are available in the system to complete the subroutine.

## Files

Item	Description
<u><a href="#">/etc/services</a></u>	Contains the service names.

## Related information

[Sockets Overview](#)

## rresvport\_af Subroutine

### Purpose

Retrieves a socket with a privileged address.

## Syntax

```
int rresvport_af(int *port, int family);
```

## Description

The **rresvport\_af** subroutine obtains a socket with a privileged address bound to the socket. A privileged Internet port is one that falls in a range between 0 and 1023.

This subroutine is similar to the existing `rresvport()` subroutine, except that `rresvport_af` also takes and address family as an argument. This function is capable of creating either an AF\_INET/TCP or an AF\_INET6/TCP socket.

Only processes with an effective user ID of root user can use the **rresvport** subroutine. An authentication scheme based on remote port numbers is used to verify permissions.

If the connection succeeds, a socket in the Internet domain of type **SOCK\_STREAM** is returned to the calling process.

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

## Parameters

Item	Description
<i>port</i>	Specifies the port to use for the connection.
<i>family</i>	Specifies either AF_INET or AF_INET6 to accommodate the appropriate version.

## Return Values

Upon successful completion, the **rresvport\_af** subroutine returns a valid, bound socket descriptor.

If the **rresvport\_af** subroutine is unsuccessful, the subroutine handler performs the following functions:

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

## Error Codes

Item	Description
<b>EAFNOSUPPORT</b>	The address family is not supported.
<b>EAGAIN</b>	All network ports are in use.
<b>EMFILE</b>	Two hundred file descriptors are currently open.
<b>ENFILE</b>	The system file table is full.
<b>ENOBUFS</b>	Insufficient buffers are available in the system to complete the subroutine.

## Files

Item	Description
<u>/etc/services</u>	Contains the service names.

## ruserok Subroutine

### Purpose

Allows servers to authenticate clients.

### Library

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

### Syntax

```
int ruserok (Host, RootUser, RemoteUser, LocalUser)
char * Host;
int RootUser;
char * RemoteUser,
* LocalUser;
```

### Description

The **ruserok** subroutine allows servers to authenticate clients requesting services.

Always specify the host name. If the local domain and remote domain are the same, specifying the domain parts is optional. To determine the domain of the host, use the **gethostname** subroutine.

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

### Parameters

Item	Description
<i>Host</i>	Specifies the name of a remote host. The <b>ruserok</b> subroutine checks for this host in the <b>/etc/host.equiv</b> file. Then, if necessary, the subroutine checks a file in the user's home directory at the server called <b>/\$HOME/.rhosts</b> for a host and remote user ID.
<i>RootUser</i>	Specifies a value to indicate whether the effective user ID of the calling process is a root user. A value of 0 indicates the process does not have a root user ID. A value of 1 indicates that the process has local root user privileges, and the <b>/etc/hosts.equiv</b> file is not checked.
<i>RemoteUser</i>	Points to a user name that is valid at the remote host. Any valid user name can be specified.
<i>LocalUser</i>	Points to a user name that is valid at the local host. Any valid user name can be specified.

### Return Values

The **ruserok** subroutine returns a 0, if the subroutine successfully locates the name specified by the *Host* parameter in the **/etc/hosts.equiv** file or the IDs specified by the *Host* and *RemoteUser* parameters are found in the **/\$HOME/.rhosts** file.

If the name specified by the *Host* parameter was not found, the **ruserok** subroutine returns a -1.

## Files

Item	Description
<a href="#"><u>/etc/services</u></a>	Contains service names.
<a href="#"><u>/etc/host.equiv</u></a>	Specifies foreign host names.
<a href="#"><u>/\$HOME/.rhosts</u></a>	Specifies the remote users of a local user account.

## Related information

[Sockets Overview](#)

## S

AIX runtime services beginning with the letter s.

## sctp\_opt\_info Subroutine

### Purpose

Passes information both into and out of SCTP stack.

### Library

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

### Syntax

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

int sctp_opt_info(sd, id, opt, *arg_size, *size);
int sd;
sctp_assoc_t id;
int opt;
void *arg_size;
size_t *size;
```

### Description

Applications use the **sctp\_opt\_info** subroutine to get information about various SCTP socket options from the stack. For the sockets with multiple associations, the association ID can be specified to apply the operation on any particular association of a socket. Because an SCTP association supports multihoming, this operation can be used to specify any particular peer address using a **sockaddr\_storage** structure. In this case, the result of the operation will be applied to only that particular peer address.

### Implementation Specifics

The **sctp\_opt\_info** subroutine is part of Base Operating System (BOS) Runtime.

### Parameters

Item	Description
<i>sd</i>	Specifies the UDP style socket descriptor returned from the <b>socket</b> system call.
<i>id</i>	Specifies the identifier of the association to query.

Item	Description
<i>opt</i>	Specifies the socket option to get.
<i>arg_size</i>	Specifies an option specific structure buffer provided by the caller.
<i>size</i>	Specifies the size of the option returned.

## Return Values

Upon successful completion, the **sctp\_opt\_info** subroutine returns 0.

If the **sctp\_opt\_info** subroutine is unsuccessful, the subroutine handler returns a value of -1 to the calling program and sets **errno** to the appropriate error code.

## Error Codes

The **sctp\_opt\_info** subroutine is unsuccessful if any of the following errors occurs:

Item	Description
<b>EFAULT</b>	Indicates that the user has insufficient authority to access the data, or the address specified in the <i>uaddr</i> parameter is not valid.
<b>EIO</b>	Indicates that a permanent I/O error occurred while referencing data.
<b>ENOMEM</b>	Indicates insufficient memory for the required paging operation.
<b>ENOSPC</b>	Indicates insufficient file system or paging space.
<b>ENOBUFS</b>	Insufficient resources were available in the system to complete the call.
<b>ENOPROTOPT</b>	Protocol not available.
<b>ENOTSOCK</b>	Indicates that the user has tried to do a socket operation on a non-socket.

## Related information

[Stream Control Transmission Protocol](#)

## sctp\_peeloff Subroutine

### Purpose

Branches off an association into a separate socket.

### Library

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

### Syntax

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

int sctp_peeloff(sd, *assoc_id);
int sd;
sctp_assoc_t *assoc_id;
```

## Description

An application uses the **sctp\_peeloff** subroutine when it wants to branch-off an existing association into a separate socket/file descriptor. It returns a new socket descriptor, which in turn can be used to send and receive subsequent SCTP packets. After it has been branched off, an association becomes completely independent of the original socket. Any subsequent data or control operations to that association must be passed using the new socket descriptor. Also, a close on the original socket descriptor will not close the new socket descriptor branched out of the association.

All the associations under the same socket share the same socket buffer space of the socket that they belong to. If an association gets branched off to a new socket using **sctp\_peeloff**, then it inherits the socket buffer space associated with the new socket descriptor. This way, the association that got peeled off keeps more buffer space.

## Implementation Specifics

The **sctp\_peeloff** subroutine is part of Base Operating System (BOS) Runtime.

## Parameters

Item	Description
<i>sd</i>	Specifies the UDP style socket descriptor returned from the <b>socket</b> system call.
<i>assoc_id</i>	Specifies the identifier of the association that is to be branched-off to a separate socket descriptor.

## Return Values

Upon successful completion, the **sctp\_peeloff** subroutine returns the nonnegative socket descriptor of the branched-off socket.

If the **sctp\_peeloff** subroutine is unsuccessful, the subroutine handler returns a value of -1 to the calling program and moves an error code to the **errno** global variable.

## Error Codes

The **sctp\_peeloff** subroutine is unsuccessful if any of the following errors occurs:

Item	Description
<b>EINVAL</b>	Invalid argument.
<b>EBADF</b>	Bad file descriptor.
<b>EAFNOSUPPORT</b>	The addresses in the specified address family cannot be used with this socket.
<b>ESOCKTNOSUPPORT</b>	The socket in the specified address family is not supported.
<b>EMFILE</b>	The per-process descriptor table is full.
<b>ENOBUFS</b>	Insufficient resources were available in the system to complete the call.
<b>ECONNABORTED</b>	The client aborted the connection.

## Related information

[Stream Control Transmission Protocol](#)

## send Subroutine

### Purpose

Sends messages from a connected socket.

### Library

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

### Syntax

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

```
int send (Socket,
Message, Length, Flags)
int Socket;
const void * Message;
size_t Length;
int Flags;
```

### Description

The **send** subroutine sends a message only when the socket is connected. This subroutine on a socket is not thread safe. The **sendto** and **sendmsg** subroutines can be used with unconnected or connected sockets.

To broadcast on a socket, first issue a **setsockopt** subroutine using the **SO\_BROADCAST** option to gain broadcast permissions.

Specify the length of the message with the *Length* parameter. If the message is too long to pass through the underlying protocol, the system returns an error and does not transmit the message.

No indication of failure to deliver is implied in a **send** subroutine. A return value of -1 indicates some locally detected errors.

If no space for messages is available at the sending socket to hold the message to be transmitted, the **send** subroutine blocks unless the socket is in a nonblocking I/O mode. Use the **select** subroutine to determine when it is possible to send more data.

The socket applications can be compiled with **COMPAT\_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

### Parameters

Item	Description
<i>Socket</i>	Specifies the unique name for the socket.
<i>Message</i>	Points to the address of the message to send.
<i>Length</i>	Specifies the length of the message in bytes.

Item	Description
<i>Flags</i>	Allows the sender to control the transmission of the message. The <i>Flags</i> parameter used to send a call is formed by logically ORing one or both of the values shown in the following list:
<b>MSG_OOB</b>	Processes out-of-band data on sockets that support <b>SOCK_STREAM</b> communication.
<b>MSG_DONTROUTE</b>	Sends without using routing tables.
<b>MSG_MPEG2</b>	Indicates that this block is a MPEG2 block. This flag is valid <b>SOCK_CONN_DGRAM</b> types of sockets only.

## Return Values

Upon successful completion, the **send** subroutine returns the number of characters sent.

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

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

## Error Codes

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

Error	Description
<b>EACCES</b>	Write access to the named socket is denied, or the socket trying to send a broadcast packet does not have broadcast capability.
<b>EADDRNOTAVAIL</b>	The specified address is not a valid address.
<b>EINVAL</b>	The specified address is not a valid address for the address family of this socket.
<b>EAFNOSUPPORT</b>	The specified address is not a valid address for the address family of this socket.
<b>EBADF</b>	The <i>Socket</i> parameter is not valid.
<b>ECONNRESET</b>	A connection was forcibly closed by a peer.
<b>EDESTADDRREQ</b>	The socket is not in connection-mode and no peer address is set.
<b>EFAULT</b>	The <i>Address</i> parameter is not in a writable part of the user address space.
<b>EHOSTUNREACH</b>	The destination host cannot be reached.
<b>EINTR</b>	A signal interrupted <b>send</b> before any data was transmitted.
<b>EINVAL</b>	The <i>Length</i> parameter is invalid.
<b>EISCONN</b>	A <b>SOCK_DGRAM</b> socket is already connected.
<b>EMSGSIZE</b>	The message is too large to be sent all at once, as the socket requires.
<b>ENETUNREACH</b>	The destination network is not reachable.
<b>ENOBUFS</b>	Insufficient resources were available in the system to perform the operation.
<b>ENOENT</b>	The path name does not name an existing file, or the path name is an empty string.
<b>ENOMEM</b>	The available data space in memory is not large enough to hold group/ACL information.
<b>ENOTSOCK</b>	The <i>Socket</i> parameter refers to a file, not a socket.
<b>EOPNOTSUPP</b>	The socket argument is associated with a socket that does not support one or more of the values set in <i>Flags</i> .

Error	Description
<b>EPIPE</b>	An attempt was made to send on a socket that was connected, but the connection has been shut down either by the remote peer or by this side of the connection. If the socket is of type <b>SOCK_STREAM</b> , the <b>SIGPIPE</b> signal is generated to the calling process.
<b>EWOULDBLOCK</b>	The socket is marked nonblocking, and no connections are present to be accepted.

#### Related reference

[sendmsg Subroutine](#)

[setsockopt Subroutine](#)

#### Related information

[select subroutine](#)

[Sockets Overview](#)

## sendmsg Subroutine

### Purpose

Sends a message from a socket using a message structure.

### Library

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

### Syntax

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

```
int sendmsg ( Socket, Message, Flags)
int Socket;
const struct msghdr Message [ ];
int Flags;
```

### Description

The **sendmsg** subroutine sends messages through connected or unconnected sockets using the **msghdr** message structure. The **/usr/include/sys/socket.h** file contains the **msghdr** structure and defines the structure members. In BSD 4.4, the size and members of the **msghdr** message structure have been modified. Applications wanting to start the old structure need to compile with **COMPAT\_43** defined. The default behaviour is that of BSD 4.4.

To broadcast on a socket, the application program must first issue a **setsockopt** subroutine using the **SO\_BROADCAST** option to gain broadcast permissions.

The **sendmsg** subroutine supports only 15 message elements.

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

The **sendmsg** routine supports IPv6 ancillary data elements as defined in the Advanced Sockets API for IPv6.

## Parameters

Item	Description
<i>Socket</i>	Specifies the socket descriptor.
<i>Message</i>	Points to the <b>msg_hdr</b> message structure containing the message to be sent.
<i>Flags</i>	Allows the sender to control the message transmission. The <b>sys/socket.h</b> file contains the <i>Flags</i> parameter. The <i>Flags</i> parameter used to send a call is formed by logically ORing one or both of the following values: <b>MSG_OOB</b> Processes out-of-band data on sockets that support <b>SOCK_STREAM</b> . <b>Note:</b> The following value is not for general use. It is an administrative tool used for debugging or for routing programs. <b>MSG_DONTROUTE</b> Sends without using routing tables. <b>MSG_MPEG2</b> Indicates that this block is a MPEG2 block. It only applies to <b>SOCK_CONN_DGRAM</b> types of sockets only.

## Return Values

Upon successful completion, the **sendmsg** subroutine returns the number of characters sent.

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

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

## Error Codes

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

Error	Description
<b>EACCES</b>	Write access to the named socket is denied, or the socket trying to send a broadcast packet does not have broadcast capability.
<b>EADDRNOTAVAIL</b>	The specified address is not a valid address.
<b>EAFNOSUPPORT</b>	The specified address is not a valid address for the address family of this socket.
<b>EBADF</b>	The <b>Socket</b> parameter is not valid.
<b>ECONNRESET</b>	A connection was forcibly closed by a peer.
<b>EDESTADDRREQ</b>	The socket is not in connection-mode and does not have its peer address set.
<b>EFAULT</b>	The <b>Address</b> parameter is not in a writable part of the user address space.
<b>EHOSTUNREACH</b>	The destination host cannot be reached.
<b>EINTR</b>	A signal interrupted the <b>sendmsg</b> subroutine before any data was transmitted.
<b>EINVAL</b>	The length of the <b>msg_hdr</b> structure is invalid.
<b>EISCONN</b>	A <b>SOCK_DGRAM</b> socket is already connected.
<b>EMSGSIZE</b>	The message is too large to be sent all at once (as the socket requires), or the <b>msg_iovlen</b> member of the <b>msg_hdr</b> structure pointed to by the <b>Messages</b> parameter is less than or equal to 0 or is greater than <b>IOV_MAX</b> .

Error	Description
<b>ENOENT</b>	The path name does not point an existing file, or the path name is an empty string.
<b>ENETUNREACH</b>	The destination network is not reachable.
<b>ENOBUFS</b>	The system ran out of memory for an internal data structure.
<b>ENOMEM</b>	The available data space in memory is not large enough to hold group or access control list (ACL) information.
<b>ENOPROTOOPT</b>	The protocol is not 64-bit supported.
<b>ENOTCONN</b>	The socket is in connection-mode but is not connected.
<b>ENOTSOCK</b>	The <b>Socket</b> parameter refers to a file, not a socket.
<b>EOPNOTSUPP</b>	The socket argument is associated with a socket that does not support one or more of the values set in flags.
<b>EPIPE</b>	An attempt was made to send on a socket that was connected, but the connection is shut down either by the remote peer or by this side of the connection. If the socket is of type <b>SOCK_STREAM</b> , the <b>SIGPIPE</b> signal is generated to the calling process.
<b>EWouldBlock</b>	The socket is marked nonblocking, and no connections are present to be accepted.

#### Related reference

[send Subroutine](#)

[setsockopt Subroutine](#)

#### Related information

[select subroutine](#)

[Sockets Overview](#)

## sendmmsg Subroutine

### Purpose

Sends multiple messages from a socket by using a message structure.

### Syntax

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

int sendmmsg (Socket, Messages, Flags)

int Socket;

struct mmsgHdr Message [];

int Flags;
```

### Description

The **sendmmsg** subroutine sends messages through the connected or unconnected sockets by using the `mmsgHdr` message structure. The `/usr/include/sys/socket.h` file contains the `mmsgHdr` message structure and defines the structure members. This subroutine is an extension to the **sendmsg** subroutine.

### Parameters

#### Socket

Specifies the socket descriptor.

## Messages

Points to an array of `mmsg_hdr` message structures that contain the messages to be sent.

## Flags

Allows the sender to control the message transmission. The **Flags** parameter that is used to send a call is formed by logically ORing the flag values. The `sendmmsg` subroutine accepts the same flag values as the `sendmsg` subroutine. The `sys/socket.h` file contains the **Flags** parameter.

## Return values

Upon successful completion, the `sendmmsg` subroutine returns the number of sent messages.

The `sendmmsg` subroutine updates the `msg_len` attribute of each `mmsg_hdr` structure to indicate the number of bytes that are sent from the corresponding message.

If the `sendmmsg` subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, which indicates the specific error, into the `errno` global variable.

## Error codes

The `sendmmsg` subroutine is unsuccessful if any of the following errors occur:

Error	Description
<b>EACCES</b>	Write access to the named socket is denied, or the socket that is trying to send a broadcast packet does not have the broadcast capability.
<b>EADDRNOTAVAIL</b>	The specified address is not a valid address.
<b>EAFNOSUPPORT</b>	The specified address is not a valid address for the address family of the socket.
<b>EBADF</b>	The <b>Socket</b> parameter is not valid.
<b>ECONNRESET</b>	A connection was forcibly closed by a peer.
<b>EDESTADDRREQ</b>	The socket is not in connection-mode and does not have its peer address set.
<b>EFAULT</b>	The <b>Address</b> parameter is not in a writable part of the user address space.
<b>EHOSTUNREACH</b>	The destination host cannot be reached.
<b>EINTR</b>	A signal interrupted the <code>sendmmsg</code> subroutine before any data was transmitted.
<b>EINVAL</b>	The length of the <code>mmsg_hdr</code> structure is invalid.
<b>EISCONN</b>	The <code>SOCK_DGRAM</code> socket is already connected.
<b>EMSGSIZE</b>	The message is too large to be sent together (as per the socket requirement), or the <code>msg_iovlen</code> member of the <code>mmsg_hdr</code> structure pointed to by the <b>Messages</b> parameter is less than or equal to 0 or is greater than the <code>IOV_MAX</code> value.
<b>ENOENT</b>	The path name does not point an existing file, or the path name is an empty string.
<b>ENETUNREACH</b>	The destination network is not reachable.
<b>ENOBUFS</b>	The system ran out of memory for an internal data structure.
<b>ENOMEM</b>	The available data space in memory is not large enough to hold group information or access control list (ACL) information.
<b>ENOPROTOOPT</b>	The protocol does not support 64 bits.
<b>ENOTCONN</b>	The socket is in connection-mode but is not connected.
<b>ENOTSOCK</b>	The <b>Socket</b> parameter refers to a file, not a socket.

Error	Description
<b>EOPNOTSUPP</b>	The <b>Socket</b> argument is associated with a socket that does not support one or more of the values that are set in the <b>Flags</b> parameter.
<b>EPIPE</b>	An attempt was made to send on a socket that was connected, but the connection is shut down either by the remote peer or by the socket side of the connection. If the socket is of type <b>SOCK_STREAM</b> , the <b>SIGPIPE</b> signal is generated to the calling process.
<b>EWouldBlock</b>	The socket is marked as nonblocking, and no connections are present to be accepted.

## sendto Subroutine

### Purpose

Sends messages through a socket.

### Library

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

### Syntax

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

```
int sendto
(Socket, Message, Length,
Flags, To, ToLength)
int Socket;
const void * Message;
size_t Length;
int Flags;
const struct sockaddr * To;
socklen_t ToLength;
```

### Description

The **sendto** subroutine allows an application program to send messages through an unconnected socket by specifying a destination address.

To broadcast on a socket, first issue a **setsockopt** subroutine using the **SO\_BROADCAST** option to gain broadcast permissions.

Provide the address of the target using the *To* parameter. Specify the length of the message with the *Length* parameter. If the message is too long to pass through the underlying protocol, the error **EMSGSIZE** is returned and the message is not transmitted.

If the **sending** socket has no space to hold the message to be transmitted, the **sendto** subroutine blocks the message unless the socket is in a nonblocking I/O mode.

Use the **select** subroutine to determine when it is possible to send more data.

The socket applications can be compiled with **COMPAT\_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

## Parameters

Item	Description
<i>Socket</i>	Specifies the unique name for the socket.
<i>Message</i>	Specifies the address containing the message to be sent.
<i>Length</i>	Specifies the size of the message in bytes.
<i>Flags</i>	Allows the sender to control the message transmission. The <i>Flags</i> parameter used to send a call is formed by logically ORing one or both of the following values: <b>MSG_OOB</b> Processes out-of-band data on sockets that support <b>SOCK_STREAM</b> . <b>Note:</b> <b>MSG_DONTRROUTE</b> Sends without using routing tables. The <b>/usr/include/sys/socket.h</b> file defines the <i>Flags</i> parameter.
<i>To</i>	Specifies the destination address for the message. The destination address is a <b>sockaddr</b> structure defined in the <b>/usr/include/sys/socket.h</b> file.
<i>ToLength</i>	Specifies the size of the destination address.

## Return Values

Upon successful completion, the **sendto** subroutine returns the number of characters sent.

If the **sendto** subroutine is unsuccessful, the system returns a value of -1, and the **errno** global variable is set to indicate the error.

## Error Codes

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

Error	Description
<b>EACCES</b>	Write access to the named socket is denied, or the socket trying to send a broadcast packet does not have broadcast capability.
<b>EADDRNOTAVAIL</b>	The specified address is not a valid address.
<b>EAFNOSUPPORT</b>	The specified address is not a valid address for the address family of this socket.
<b>EBADF</b>	The <i>Socket</i> parameter is not valid.
<b>ECONNRESET</b>	A connection was forcibly closed by a peer.
<b>EDESTADDRREQ</b>	The socket is not in connection-mode and no peer address is set.
<b>EFAULT</b>	The <i>Address</i> parameter is not in a writable part of the user address space.
<b>EHOSTUNREACH</b>	The destination host cannot be reached.
<b>EINTR</b>	A signal interrupted <b>sendto</b> before any data was transmitted.
<b>EINVAL</b>	The <i>Length</i> or <i>ToLength</i> parameter is invalid.
<b>EISCONN</b>	A <b>SOCK_DGRAM</b> socket is already connected.
<b>EMSGSIZE</b>	The message is too large to be sent all at once as the socket requires.
<b>ENETUNREACH</b>	The destination network is not reachable.
<b>ENOBUFS</b>	The system ran out of memory for an internal data structure.

Error	Description
<b>ENOENT</b>	The path name does not name an existing file, or the path name is an empty string.
<b>ENOMEM</b>	The available data space in memory is not large enough to hold group/ACL information.
<b>ENOPROTOOPT</b>	The protocol is not 64-bit supported.
<b>ENOTCONN</b>	The socket is in connection-mode but is not connected.
<b>ENOTSOCK</b>	The <i>Socket</i> parameter refers to a file, not a socket.
<b>EOPNOTSUPP</b>	The socket argument is associated with a socket that does not support one or more of the values set in <i>Flags</i> .
<b>EPIPE</b>	An attempt was made to send on a socket that was connected, but the connection has been shut down either by the remote peer or by this side of the connection. If the socket is of type <b>SOCK_STREAM</b> , the <b>SIGPIPE</b> signal is generated to the calling process.
<b>EWOULDBLOCK</b>	The socket is marked nonblocking, and no connections are present to be accepted.

#### Related reference

[setsockopt Subroutine](#)

#### Related information

[select subroutine](#)

[Sending Datagrams Example Program](#)

## send\_file Subroutine

### Purpose

Sends the contents of a file through a socket.

### Library

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

### Syntax

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

```
ssize_t send_file(Socket_p, sf_iobuf, flags)
```

```
int * Socket_p;
```

```
struct sf_parms * sf_iobuf;
```

```
uint_t flags;
```

### Description

The **send\_file** subroutine sends data from the opened file specified in the *sf\_iobuf* parameter, over the connected socket pointed to by the *Socket\_p* parameter.

**Note:** Currently, the **send\_file** only supports the TCP/IP protocol (SOCK\_STREAM socket in AF\_INET). An error will be returned when this function is used on any other types of sockets.

### Parameters

**Item**      **Description**

*Socket\_p* Points to the socket descriptor of the socket which the file will be sent to.

**Note:** This is different from most of the socket functions.

*sf\_iobuf* Points to a *sf\_parms* structure defined as follows:

```
/*
 * Structure for the send_file system call
 */
#ifdef __64BIT__
#define SF_INT64(x)      int64_t x;
#define SF_UINT64(x)    uint64_t x;
#else
#ifdef _LONG_LONG
#define SF_INT64(x)      int64_t x;
#define SF_UINT64(x)    uint64_t x;
#else
#define SF_INT64(x)      int filler_##x; int x;
#define SF_UINT64(x)    int filler_##x; uint_t x;
#endif
#endif

struct sf_parms {
    /* ----- header parms ----- */
    void      *header_data;          /* Input/Output. Points to header buf */
    uint_t    header_length;        /* Input/Output. Length of the header */
    /* ----- file parms ----- */
    int       file_descriptor;       /* Input. File descriptor of the file */
    SF_UINT64(file_size)            /* Output. Size of the file */
    SF_UINT64(file_offset)         /* Input/Output. Starting offset */
    SF_INT64(file_bytes)           /* Input/Output. number of bytes to send */
    /* ----- trailer parms ----- */
    void      *trailer_data;        /* Input/Output. Points to trailer buf */
    uint_t    trailer_length;       /* Input/Output. Length of the trailer */
    /* ----- return info ----- */
    SF_UINT64(bytes_sent)          /* Output. number of bytes sent */
};
```

**header\_data**

Points to a buffer that contains header data which is to be sent before the file data. May be a NULL pointer if *header\_length* is 0. This field will be updated by **send\_file** when header is transmitted - that is, *header\_data* + number of bytes of the header sent.

**header\_length**

Specifies the number of bytes in the *header\_data*. This field must be set to 0 to indicate that header data is not to be sent. This field will be updated by **send\_file** when header is transmitted - that is, *header\_length* - number of bytes of the header sent.

**file\_descriptor**

Specifies the file descriptor for a file that has been opened and is readable. This is the descriptor for the file that contains the data to be transmitted. The *file\_descriptor* is ignored when *file\_bytes* = 0. This field is not updated by **send\_file**.

**file\_size**

Contains the byte size of the file specified by *file\_descriptor*. This field is filled in by the kernel.

**file\_offset**

Specifies the byte offset into the file from which to start sending data. This field is updated by the **send\_file** when file data is transmitted - that is, *file\_offset* + number of bytes of the file data sent.

Item	Description
------	-------------

***file\_bytes***

Specifies the number of bytes from the file to be transmitted. Setting *file\_bytes* to -1 transmits the entire file from the *file\_offset*. When this field is not set to -1, it is updated by **send\_file** when file data is transmitted - that is, *file\_bytes* - number of bytes of the file data sent.

***trailer\_data***

Points to a buffer that contains trailer data which is to be sent after the file data. May be a NULL pointer if *trailer\_length* is 0. This field will be updated by **send\_file** when trailer is transmitted - that is, *trailer\_data* + number of bytes of the trailer sent.

***trailer\_length***

Specifies the number of bytes in the *trailer\_data*. This field must be set to 0 to indicate that trailer data is not to be sent. This field will be updated by **send\_file** when trailer is transmitted - that is, *trailer\_length* - number of bytes of the trailer sent.

***bytes\_sent***

Contains number of bytes that were actually sent in this call to **send\_file**. This field is filled in by the kernel.

All fields marked with Input in the *sf\_parms* structure requires setup by an application prior to the **send\_file** calls. All fields marked with Output in the *sf\_parms* structure adjusts by **send\_file** when it successfully transmitted data, that is, either the specified data transmission is partially or completely done.

The **send\_file** subroutine attempts to write *header\_length* bytes from the buffer pointed to by *header\_data*, followed by *file\_bytes* from the file associated with *file\_descriptor*, followed by *trailer\_length* bytes from the buffer pointed to by *trailer\_data*, over the connection associated with the socket pointed to by *Socket\_p*.

As the data is sent, the kernel updates the parameters pointed by *sf\_iobuf* so that if the **send\_file** has to be called multiple times (either due to interruptions by signals, or due to non-blocking I/O mode) in order to complete a file data transmission, the application can reissue the **send\_file** command without setting or re-adjusting the parameters over and over again.

If the application sets *file\_offset* greater than the actual file size, or *file\_bytes* greater than (the actual file size - *file\_offset*), the return value will be -1 with errno EINVAL.

Item	Description
------	-------------

*flags* Specifies the following attributes:

**SF\_CLOSE**

Closes the socket pointed to by *Socket\_p* after the data has been successfully sent or queued for transmission.

**SF\_REUSE**

Prepares the socket for reuse after the data has been successfully sent or queued for transmission and the existing connection closed.

**Note:** This option is currently not supported on this operating system.

**SF\_DONT\_CACHE**

Does not put the specified file in the Network Buffer Cache.

**SF\_SYNC\_CACHE**

Verifies/Updates the Network Buffer Cache for the specified file before transmission.

When the *SF\_CLOSE* flag is set, the connected socket specified by *Socket\_p* will be disconnected and closed by **send\_file** after the requested transmission has been successfully done. The socket descriptor pointed to by *Socket\_p* will be set to -1. This flag won't take effect if **send\_file** returns non-0.

The flag *SF\_REUSE* currently is not supported by AIX. When this flag is specified, the socket pointed by *Socket\_p* will be closed and returned as -1. A new socket needs to be created for the next connection.

**send\_file** will take advantage of a Network Buffer Cache in kernel memory to dynamically cache the output file data. This will help to improve the **send\_file** performance for files which are:

1. accessed repetitively through network and
2. not changed frequently.

Applications can exclude the specified file from being cached by using the *SF\_DONT\_CACHE* flag. **send\_file** will update the cache every so often to make sure that the file data in cache is valid for a certain time period. The network option parameter "send\_file\_duration" controlled by the **no** command can be modified to configure the interval of the **send\_file** cache validation, the default is 300 (in seconds). Applications can use the *SF\_SYNC\_CACHE* flag to ensure that a cache validation of the specified file will occur before the file is sent by **send\_file**, regardless the value of the "send\_file\_duration". Other Network Buffer Cache related parameters are "nbc\_limit", "nbc\_max\_cache", and "nbc\_min\_cache". For additional information, see the **no** command.

## Return Value

There are three possible return values from **send\_file**:

Val	Description
-----	-------------

- |    |   |
|----|---|
| -1 | an error has occurred, errno contains the error code.   |
| 0  | the command has completed successfully.   |
| 1  | the command was completed partially, some data has been transmitted but the command has to return for some reason, for example, the command was interrupted by signals. |

The fields marked with Output in the *sf\_parms* structure (pointed to by *sf\_iobuf*) is updated by **send\_file** when the return value is either 0 or 1. The *bytes\_sent* field contains the total number of bytes that were sent in this call. It is always true that *bytes\_sent* (Output) <= *header\_length*(Input) + *file\_bytes*(Input) + *trailer\_length* (Input).

The **send\_file** supports the blocking I/O mode and the non-blocking I/O mode. In the blocking I/O mode, **send\_file** blocks until all file data (plus the header and the trailer) is sent. It adjusts the *sf\_iobuf* to reflect the transmission results, and return 0. It is possible that **send\_file** can be interrupted before the request is fully done, in that case, it adjusts the *sf\_iobuf* to reflect the transmission progress, and return 1.

In the non-blocking I/O mode, the **send\_file** transmits as much as the socket space allows, adjusts the *sf\_iobuf* to reflect the transmission progress, and returns either 0 or 1. When there is no socket space in the system to buffer any of the data, the **send\_file** returns -1 and sets *errno* to EWOULDBLOCK. **select** or **poll** can be used to determine when it is possible to send more data.

**Possible errno returned:**

EBADF	Either the socket or the file descriptor parameter is not valid.
ENOTSOCK	The socket parameter refers to a file, not a socket.
EPROTONOSUPPORT	Protocol not supported.
EFAULT	The addresses specified in the HeaderTrailer parameter is not in a writable part of the user-address space.
EINTR	The operation was interrupted by a signal before any data was sent. (If some data was sent, <b>send_file</b> returns the number of bytes sent before the signal, and EINTR is not set).
EINVAL	The offset, length of the HeaderTrailer, or flags parameter is invalid.
ENOTCONN	A <b>send_file</b> on a socket that is not connected, a <b>send_file</b> on a socket that has not completed the connect sequence with its peer, or is no longer connected to its peer.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.
ENOMEM	No memory is available in the system to perform the operation.

**PerformanceNote**

By taking advantage of the Network Buffer Cache, **send\_file** provides better performance and network throughput for file transmission. It is recommended for files bigger than 4K bytes.

**Related information**

[select subroutine](#)

[Sockets Overview](#)

[Understanding Socket Data Transfer](#)

## set\_auth\_method Subroutine

### Purpose

Sets the authentication methods for the rcmds for this system.

### Library

Authentication Methods Library (**libauthm.a**)

## Syntax

```
int set_auth_method (uint ** authm)
```

## Description

This method configures the authentication methods for the system. The authentication methods should be passed to the function in the order in which they should be attempted in the unsigned integer pointer in which the user passed.

The list is an array of unsigned integers terminated by a zero. Each integer identifies an authentication method. The order that a client should attempt to authenticate is defined by the order of the list.

The flags identifying the authentication methods are defined in the `/usr/include/authm.h` file.

Any undefined bits in the input parameter invalidate the entire command. If the same authentication method is specified twice or if any authentication method is specified after Standard AIX, the command fails.

The user must have root authority or this method fails.

## Parameter

Item	Description
<i>authm</i>	Points to an array of unsigned integers. The list of authentication methods to be set is terminated by a zero.

## Return Values

Upon successful completion, the `set_auth_method` subroutine returns a zero.

Upon unsuccessful completion, the `set_auth_method` subroutine returns an **errno**.

## Related information

[Communications and networks](#)

[Authentication and the secure rcmds](#)

## setdomainname Subroutine

### Purpose

Sets the name of the current domain.

### Library

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

## Syntax

```
int setdomainname ( Name, NameLen )  
char *Name;  
int NameLen;
```

## Description

The `setdomainname` subroutine sets the name of the domain for the host machine. It is normally used when the system is bootstrapped. You must have root user authority to run this subroutine.

The purpose of domains is to enable two distinct networks that may have host names in common to merge. Each network would be distinguished by having a different domain name. At the current time, only Network Information Service (NIS) makes use of domains set by this subroutine.

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

**Note:** Domain names are restricted to 256 characters.

## Parameters

Item	Description
<i>Name</i>	Specifies the domain name to be set.
<i>Namelen</i>	Specifies the size of the array pointed to by the <i>Name</i> parameter.

## Return Values

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

## Error Codes

The following errors may be returned by this subroutine:

Error	Description
<b>EFAULT</b>	The <i>Name</i> parameter gave an invalid address.
<b>EPERM</b>	The caller was not the root user.

## Related reference

[getdomainname Subroutine](#)

## Related information

[Sockets Overview](#)

## sethostent Subroutine

### Purpose

Opens network host file.

### Library

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

### Syntax

```
#include <netdb.h>
```

```
sethostent ( StayOpen )  
int StayOpen;
```

## Description

When using the **sethostent** subroutine in DNS/BIND name service resolution, **sethostent** allows a request for the use of a connected socket using TCP for queries. If the *StayOpen* parameter is non-zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to **gethostbyname** or **gethostbyaddr**.

When using the **sethostent subroutine** to search the **/etc/hosts** file, **sethostent** opens and rewinds the **/etc/hosts** file. If the *StayOpen* parameter is non-zero, the hosts database is not closed after each call to **gethostbyname** or **gethostbyaddr**.

## Parameters

Item	Description
<i>StayOpen</i>	When used in NIS name resolution and to search the local <b>/etc/hosts</b> file, it contains a value used to indicate whether to close the host file after each call to <b>gethostbyname</b> and <b>gethostbyaddr</b> . A non-zero value indicates not to close the host file after each call and a zero value allows the file to be closed.  When used in DNS/BIND name resolution, a non-zero value retains the TCP connection after each call to <b>gethostbyname</b> and <b>gethostbyaddr</b> . A value of zero allows the connection to be closed.

## Files

Item	Description
<b><u>/etc/hosts</u></b>	Contains the host name database.
<b><u>/etc/netsvc.conf</u></b>	Contains the name services ordering.
<b><u>/etc/include/netdb.h</u></b>	Contains the network database structure.

## Related reference

[endhostent Subroutine](#)

## Related information

[Sockets Overview](#)

[Network Address Translation](#)

## sethostent\_r Subroutine

### Purpose

Opens network host file.

### Library

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

### Syntax

```
#include <netdb.h>  
sethostent_r (StayOpenflag, ht_data)  
  
int StayOpenflag;  
struct hostent_data *ht_data;
```

## Description

When using the **sethostent\_r** subroutine in DNS/BIND name service resolution, **sethostent\_r** allows a request for the use of a connected socket using TCP for queries. If the *StayOpen* parameter is non-zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to **gethostbyname\_r** or **gethostbyaddr\_r**.

When using the **sethostent\_r** subroutine to search the **/etc/hosts** file, **sethostent\_r** opens and rewinds the **/etc/hosts** file. If the *StayOpen* parameter is non-zero, the hosts database is not closed after each call to **gethostbyname\_r** or **gethostbyaddr\_r**. It internally runs the **sethostent** command.

## Parameters

Item	Description
<i>StayOpenflag</i>	When used in NIS name resolution and to search the local <b>/etc/hosts</b> file, it contains a value used to indicate whether to close the host file after each call to the <b>gethostbyname</b> and <b>gethostbyaddr</b> subroutines. A non-zero value indicates not to close the host file after each call, and a zero value allows the file to be closed.  When used in DNS/BIND name resolution, a non-zero value retains the TCP connection after each call to <b>gethostbyname</b> and <b>gethostbyaddr</b> . A value of zero allows the connection to be closed.
<i>ht_data</i>	Points to the <b>hostent_data</b> structure.

## Files

Item	Description
<b>/etc/hosts</b>	Contains the host name database.
<b>/etc/netsvc.conf</b>	Contains the name services ordering.
<b>/etc/include/netdb.h</b>	Contains the network database structure.

## sethostid Subroutine

### Purpose

Sets the unique identifier of the current host.

### Library

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

### Syntax

```
int sethostid ( HostID )
int HostID;
```

### Description

The **sethostid** subroutine allows a calling process with a root user ID to set a new 32-bit identifier for the current host. The **sethostid** subroutine enables an application program to reset the host ID.

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

## Parameters

Item	Description
<i>HostID</i>	Specifies the unique 32-bit identifier for the current host.

## Return Values

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

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

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

## Error Codes

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

Error	Description
<b>EPERM</b>	The calling process did not have an effective user ID of root user.

## Related information

[Sockets Overview](#)

## sethostname Subroutine

### Purpose

Sets the name of the current host.

### Library

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

### Syntax

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

### Description

The **sethostname** subroutine sets the name of a host machine. Only programs with a root user ID can use this subroutine.

The **sethostname** subroutine allows a calling process with root user authority to set the internal host name of a machine on a network.

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

## Parameters

Item	Description
<i>Name</i>	Specifies the name of the host machine.
<i>NameLength</i>	Specifies the length of the <i>Name</i> array.

## Return Values

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

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

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable see [Error Notification Object Class](#) in *General Programming Concepts: Writing and Debugging Programs*.

## Error Codes

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

Error	Description
<b>EFAULT</b>	The <i>Name</i> parameter or <i>NameLength</i> parameter gives an address that is not valid.
<b>EPERM</b>	The calling process did not have an effective root user ID.

### Related reference

[gethostname Subroutine](#)

[gethostid Subroutine](#)

### Related information

[Sockets Overview](#)

[Understanding Network Address Translation](#)

## setnetent Subroutine

### Purpose

Opens the **/etc/networks** file and sets the file marker.

### Library

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

### Syntax

```
#include <netdb.h>
void setnetent (StayOpen)
int StayOpen;
```

### Description

The **setnetent** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **setnetent** subroutine opens the **/etc/networks** file and sets the file marker at the beginning of the file.

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

## Parameters

Item	Description
<i>StayOpen</i>	Contains a value used to indicate when to close the <b>/etc/networks</b> file. Specifying a value of 0 closes the <b>/etc/networks</b> file after each call to the <b>getnetent</b> subroutine. Specifying a nonzero value leaves the <b>/etc/networks</b> file open after each call.

## Return Values

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

## Files

Item	Description
<b><u>/etc/networks</u></b>	Contains official network names.

### Related reference

[endnetent Subroutine](#)

[getnetent Subroutine](#)

[getnetbyaddr Subroutine](#)

### Related information

[Sockets Overview](#)

## setnetent\_r Subroutine

### Purpose

Opens the **/etc/networks** file and sets the file marker.

### Library

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

### Syntax

```
#include <netdb.h>
int setnetent_r(StayOpenflag, net_data)
struct netent_data *net_data;
int StayOpenflag;
```

### Description

The **setnetent\_r** subroutine opens the **/etc/networks** file and sets the file marker at the beginning of the file.

## Parameters

Item	Description
<i>StayOpenflag</i>	Contains a value used to indicate when to close the <b>/etc/networks</b> file. Specifying a value of 0 closes the <b>/etc/networks</b> file after each call to the <b>getnetent</b> subroutine. Specifying a nonzero value leaves the <b>/etc/networks</b> file open after each call.
<i>net_data</i>	Points to the <b>netent_data</b> structure.

## Files

Item	Description
<b>/etc/networks</b>	Contains official network names.

## setnetgrent\_r Subroutine

### Purpose

Handles the group network entries.

### Library

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

### Syntax

```
#include <netdb.h>
int setnetgrent_r(NetGroup, ptr)
char *NetGroup;
void **ptr;
```

### Description

The **setnetgrent\_r** subroutine functions the same as the **setnetgrent** subroutine.

The **setnetgrent\_r** subroutine establishes the network group from which the **getnetgrent\_r** subroutine will obtain members. This subroutine also restarts calls to the **getnetgrent\_r** subroutine from the beginning of the list. If the previous **setnetgrent\_r** call was to a different network group, an **endnetgrent\_r** call is implied. The **endnetgrent\_r** subroutine frees the space allocated during the **getnetgrent\_r** calls.

### Parameters

Item	Description
<i>NetGroup</i>	Points to a network group.
<i>ptr</i>	Keeps the function threadsafe.

### Return Values

The **setnetgrent\_r** subroutine returns a 0 if successful and a -1 if unsuccessful.

## Files

Item	Description
<code>/etc/netgroup</code>	Contains network groups recognized by the system.
<code>/usr/include/netdb.h</code>	Contains the network database structures.

## setprotoent Subroutine

### Purpose

Opens the `/etc/protocols` file and sets the file marker.

### Library

Standard C Library (`libc.a`)

### Syntax

```
#include <netdb.h>
```

```
void setprotoent (StayOpen)  
int StayOpen;
```

### Description

The `setprotoent` subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The `setprotoent` subroutine opens the `/etc/protocols` file and sets the file marker to the beginning of the file.

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

### Parameters

Item	Description
<code>StayOpen</code>	Indicates when to close the <code>/etc/protocols</code> file.  Specifying a value of 0 closes the file after each call to <code>getprotoent</code> .  Specifying a nonzero value allows the <code>/etc/protocols</code> file to remain open after each subroutine.

### Return Values

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

## Files

Item	Description
<code>/etc/protocols</code>	Contains the protocol names.

### Related reference

[endprotoent Subroutine](#)

[getprotobyname Subroutine](#)

## Related information

[Sockets Overview](#)

## setprotoent\_r Subroutine

### Purpose

Opens the **/etc/protocols** file and sets the file marker.

### Library

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

### Syntax

```
#include <netdb.h>

int setprotoent_r(StayOpenflag, proto_data);
int StayOpenflag;
struct protoent_data *proto_data;
```

### Description

The **setprotoent\_r** subroutine opens the **/etc/protocols** file and sets the file marker to the beginning of the file.

### Parameters

Item	Description
<i>StayOpenflag</i>	Indicates when to close the <b>/etc/protocols</b> file.  Specifying a value of 0 closes the file after each call to <b>getprotoent</b> . Specifying a nonzero value allows the <b>/etc/protocols</b> file to remain open after each subroutine.

### Files

Item	Description
<b>/etc/protocols</b>	Contains the protocol names.

## setservent Subroutine

### Purpose

Opens **/etc/services** file and sets the file marker.

### Library

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

### Syntax

```
#include <netdb.h>

void setservent ( StayOpen)
int StayOpen;
```

## Description

The **setservernt** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **setservernt** subroutine opens the [/etc/services](#) file and sets the file marker at the beginning of the file.

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

## Parameters

Item	Description
<i>StayOpen</i>	Indicates when to close the <a href="#">/etc/services</a> file. Specifying a value of 0 closes the file after each call to the <a href="#">getservernt</a> subroutine. Specifying a nonzero value allows the file to remain open after each call.

## Return Values

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

## Files

Item	Description
<a href="#">/etc/services</a>	Contains service names.

### Related reference

[endprotoent](#) Subroutine

[getprotobyname](#) Subroutine

### Related information

[Sockets Overview](#)

## setservernt\_r Subroutine

### Purpose

Opens [/etc/services](#) file and sets the file marker.

### Library

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

### Syntax

```
#include <netdb.h>

int setservernt_r(StayOpenflag, serv_data)
int StayOpenflag;
struct servent_data serv_data;
```

## Description

The **setservernt\_r** subroutine opens the [/etc/services](#) file and sets the file marker at the beginning of the file.

## Parameters

Item	Description
<i>StayOpenflag</i>	Indicates when to close the <b>/etc/services</b> file.  Specifying a value of 0 closes the file after each call to the <b>getservent</b> subroutine. Specifying a nonzero value allows the file to remain open after each call.
<i>serv_data</i>	Points to the <b>servent_data</b> structure.

## Files

Item	Description
<b>/etc/services</b>	Contains service names.

## setsockopt Subroutine

### Purpose

Sets the socket options.

### Library

Standard C Library (libc.a)

### Syntax

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

```
int setsockopt ( Socket, Level, OptionName, OptionValue, OptionLength )
int Socket, Level, OptionName;
const void* OptionValue;
socklen_t OptionLength;
```

### Description

The **setsockopt** subroutine sets the options that are associated with a socket. The options can exist at a multiple protocol levels. The options are at the uppermost socket level.

The **setsockopt** subroutine provides an application program with the means to control a socket communication. An application program can use the **setsockopt** subroutine to enable debugging at the protocol level, allocate buffer space, control timeouts, or to allow socket data broadcasts. The `/usr/include/sys/socket.h` file defines the options that are available to the **setsockopt** subroutine.

Specify the protocol level at which the option is located and the name of the option when you set socket options.

Use the *OptionValue* and *OptionLength* parameters to access option values for the **setsockopt** subroutine. The *OptionValue* and *OptionLength* parameters identify a buffer in which the value for the requested option or is returned.

The applications that contain the **setsockopt** subroutine must be compiled with the `_BSD` macro set to a specific value. The acceptable values are **43** and **44**. In addition, all socket applications must include the Berkeley Software Distribution (BSD) `libbsd.a` library.

# Parameters

Table 1. Parameters

Item	Description
<i>Socket</i>	Specifies the unique name of the socket.
<i>Level</i>	<p>Specifies the protocol level at which the option is located. You can set the socket options at the following levels:</p> <p><b>Socket level</b> Specifies the <i>Level</i> parameter as the <b>SOL_SOCKET</b> option.</p> <p><b>Other levels</b> Supplies the appropriate protocol number for the protocol that controls the option. For example, to indicate that the TCP protocol interprets the option, set the <i>Level</i> parameter to the protocol number of TCP as defined in the <code>netinet/in.h</code> file. Similarly, to indicate that the ATM protocol interprets the option, set the <i>Level</i> parameter to the <b>NDDPROTO_ATM</b> option as defined in <code>sys/atmsock.h</code> file.</p>
<i>OptionName</i>	<p>Specifies the option to set. The <i>OptionName</i> parameter and the specified options are passed uninterpreted to the appropriate protocol module for interpretation. The <code>sys/socket.h</code> file defines the socket protocol-level options. The <code>netinet/tcp.h</code> file defines the TCP protocol level options. You can enable or disable the socket level options and they operate in a toggle fashion.</p> <p>The following list defines the socket protocol level options that are available in the <code>sys/socket.h</code> file:</p> <p><b>SO_DEBUG</b> Enables or disables the debugging in the underlying protocol modules. The <b>SO_DEBUG</b> turns on the recording of the debugging information. Set this option in one of the following ways at the command level:</p> <ul style="list-style-type: none"><li>• To turn on or turn OFF this option for the existing sockets by using the <b>sodebug</b> command.</li><li>• To turn on this option for the specific service, you must specify the <code> DEBUG[=<i>level</i>]</code> in the <code>wait/nwait</code> field of a service in the <code>inetd.conf</code> file.</li><li>• To turn on or turn OFF this option for all the subsequent sockets that are created by the process, you must set the <b>sodebug_env</b> parameter to <b>no</b>. Specify the <code>SODEBUG=<i>level</i></code> in the process environment.</li></ul> <p>The value for the <i>level</i> parameter can be either <b>min</b>, <b>normal</b>, or <b>detail</b>.</p> <p><b>SO_REUSEADDR</b> Specifies that the rules used in validating addresses that are supplied by a <b>bind</b> subroutine must allow reuse of a local port.</p> <p><b>SO_REUSEADDR</b> option allows an application to explicitly deny subsequent <b>bind</b> subroutine to the port or address of the socket with the <b>SO_REUSEADDR</b> socket option. This allows an application to block other applications from binding with the <b>bind</b> subroutine.</p> <p><b>SO_REUSEPORT</b> Specifies that the rules used in validating addresses that are supplied by a <b>bind</b> subroutine must allow reuse of a local port or address combination. Each binding of the port or address combination must specify the <b>SO_REUSEPORT</b> socket option</p> <p><b>SO_CKSUMREV</b> Enables the performance enhancements in the protocol layers. If the protocol supports the <b>SO_CKSUMREV</b> option, enabling causes the protocol to defer checksum verification until the data of the user is moved into the buffer of the user (on <code>recv</code>, <code>recvfrom</code>, <code>read</code>, or <code>recvmsg</code> thread). This can cause applications to be awakened when no data is available, in the case of a checksum error and the <b>EAGAIN</b> is returned. The applications that set this option must handle the <b>EAGAIN</b> error code returned from a receive call.</p> <p><b>SO_KEEPAIVE</b> Enables or disables the periodic transmission of ACK messages on a connected socket to monitor the activity of a connection. The idle interval time can be designated by using the Transmission Control Protocol/Internet Protocol (TCP/IP) <b>no</b> command. For more information on broken connections, see <a href="#">Understanding Socket Types and Protocols</a> in <i>Communications Programming Concepts</i>.</p>

Table 1. Parameters (continued)

Item	Description
OptionName	<p data-bbox="565 216 711 237"><b>SO_DONTROUTE</b></p> <p data-bbox="609 243 1474 306">Does not apply routing on outgoing messages. Indicates that outgoing messages must bypass the standard routing facilities. Instead, they are directed to the appropriate network interface according to the network portion of the destination address.</p> <p data-bbox="565 321 711 342"><b>SO_BROADCAST</b></p> <p data-bbox="609 348 976 369">Permits sending of the broadcast messages.</p> <p data-bbox="565 384 667 405"><b>SO_LINGER</b></p> <p data-bbox="609 411 1474 531">Controls the action that is taken when an unsent messages queue exists for a socket, and a process performs a <b>close</b> subroutine on the socket. Lingers on a <b>close</b> subroutine if data is present. If <b>SO_LINGER</b> option is set, the system blocks the process during the <b>close</b> subroutine until it can transmit the data or until the time expires. If you do not specify the <b>SO_LINGER</b> option and a <b>close</b> subroutine is issued, the system handles the call in a way that allows the process to continue as quickly as possible.</p> <p data-bbox="609 546 1474 678">The <code>sys/socket.h</code> file defines the <code>linger</code> structure that contains the <code>l_linger</code> member to specify the <code>linger</code> time interval. If the <code>linger</code> time is set to anything but <b>0</b>, the system tries to send any messages that are queued on the socket. The maximum value that the <code>l_linger</code> member can be set to is <b>65535</b>. If the application has requested the SPEC1170 compliant behavior by exporting the <code>XPG_SUS_ENV</code> environment variable, the <code>linger</code> time is <i>n</i> seconds. Otherwise, the <code>linger</code> time is <i>n</i> per 100 seconds (ticks), where <i>n</i> is the value of the <code>l_linger</code> member.</p> <p data-bbox="565 693 711 714"><b>SO_OOBINLINE</b></p> <p data-bbox="609 720 1159 741">Leaves the received out-of-band data (data marked urgent) in line.</p> <p data-bbox="565 756 667 777"><b>SO_SNDBUF</b></p> <p data-bbox="609 783 818 804">Sets the send buffer size.</p> <p data-bbox="565 819 667 840"><b>SO_RCVBUF</b></p> <p data-bbox="609 846 837 867">Sets the receive buffer size.</p> <p data-bbox="565 882 711 903"><b>SO_SNDLOWAT</b></p> <p data-bbox="609 909 862 930">Sets the send low-water mark.</p> <p data-bbox="565 945 711 966"><b>SO_RCVLOWAT</b></p> <p data-bbox="609 972 898 993">Sets the receive a low-water mark.</p> <p data-bbox="565 1008 711 1029"><b>SO_SNDTIMEO</b></p> <p data-bbox="609 1035 1474 1119">Sets the maximum length of time that a send-type function that is called with <code>socket</code>, for example <code>send</code> or <code>write</code> subroutines, can remain blocked before it completes. The <b>SO_SNDTIMEO</b> option requires a <code>TIMEVAL</code> structure. The <code>TIMEVAL</code> structure contains the number of seconds and microseconds. You can specify the <code>TIMEVAL</code> value in microseconds. However, the internal TCP/IP timers that are used to implement this function have a granularity of 10 milliseconds. If the timeout value is set to a value less than 10 milliseconds, it results in the same behavior as if the timeout value is not set.</p> <p data-bbox="565 1134 711 1155"><b>SO_RCVTIMEO</b></p> <p data-bbox="609 1161 1474 1287">Sets the maximum length of time that a receive-type function that is called with <code>socket</code>, for example <code>recv</code> or <code>read</code> subroutines, can wait before it completes. The <b>SO_RCVTIMEO</b> option requires a <code>TIMEVAL</code> structure. The <code>TIMEVAL</code> structure contains the number of seconds and microseconds. You can specify the <code>TIMEVAL</code> value in microseconds. However, the internal TCP/IP timers that are used to implement this function have a granularity of 10 milliseconds. If the timeout value is set to a value less than 10 milliseconds, it results in the same behavior as if the timeout value is not set.</p> <p data-bbox="565 1302 667 1323"><b>SO_ERROR</b></p> <p data-bbox="609 1329 959 1350">Sets the retrieval of error status and clear.</p> <p data-bbox="565 1365 667 1386"><b>SO_TYPE</b></p> <p data-bbox="609 1392 891 1413">Sets the retrieval of a socket type.</p>

Table 1. Parameters (continued)

Item	Description
OptionName	The following list defines the TCP protocol level options that are available in the <code>netinet/tcp.h</code> file:
	<b>TCP_CWND_IF</b> Increases the factor of the TCP congestion window (cwnd) during the congestion avoidance. The value must be in the range <b>0</b> to <b>100</b> ( <b>0</b> is disable). Enable the <b>tcp_cwnd_modified</b> network tunable option.
	<b>TCP_CWND_DF</b> Decreases the factor of the TCP cwnd during the congestion avoidance. The value must be in the range <b>0</b> to <b>100</b> ( <b>0</b> is disable). Enable the <b>tcp_cwnd_modified</b> network tunable option.
	<b>TCP_NOTENTER_SSTART</b> Avoids reentering the slow start after the retransmit timeout that might reset the cwnd to the initial window size, instead of the size of the current slow-start threshold ( <code>ss_threshold</code> ) value or half of the maximum cwnd ( <code>max_cwnd/2</code> ). The values are <b>1</b> for enable and <b>0</b> for disable. Enable the <b>tcp_cwnd_modified</b> network tunable option.
	<b>TCP_NOREDUCE_CWND_IN_FRXMT</b> Does not decrease the cwnd size in the fast retransmit phrase. The values are <b>1</b> for enable and <b>0</b> for disable. Enable the <b>tcp_cwnd_modified</b> network tunable option.
	<b>TCP_NOREDUCE_CWND_EXIT_FRXMT</b> Does not decrease the cwnd size when exits the fast retransmit phrase. The values are <b>1</b> for enable and <b>0</b> for disable. Enable the <b>tcp_cwnd_modified</b> network tunable option.
	<b>TCP_KEEPCNT</b> Specifies the maximum number of keepalive packets to be sent to validate a connection. This socket option value is inherited from the parent socket. The default value is <b>8</b> .
	<b>TCP_KEEPIDLE</b> Specifies the number of seconds of idle time on a connection after which the TCP sends a keepalive packet. This socket option value is inherited from the parent socket from the <code>accept</code> system call. The default value is 7200 seconds (14400 half seconds).
	<b>TCP_KEEPINTVL</b> Specifies the interval of time between each keepalive packets. The interval of time is measured in seconds. This socket option value is inherited from the parent socket from the <code>accept</code> system call. The default value is 75 seconds (150 half seconds).
	<b>TCP_NODELAY</b> Specifies whether the TCP must follow the Nagle algorithm for deciding when to send the data. By default, TCP follows the Nagle algorithm. To disable this behavior, applications can enable the <b>TCP_NODELAY</b> option to force to TCP to send the data immediately. For example, <b>TCP_NODELAY</b> option must be used when there is an application by using the TCP for a request or response.

Table 1. Parameters (continued)

Item	Description
OptionName	<p><b>TCP_RFC1323</b> Enables or disables the RFC 1323 enhancements on the specified TCP socket. An application might contain the following lines to enable the RFC 1323:</p> <pre data-bbox="623 310 1081 380">int on=1; setsockopt(s, IPPROTO_TCP, TCP_RFC1323, &amp;on, sizeof(on));</pre>
	<p><b>TCP_STDURG</b> Enables or disables the RFC 1122 compliant urgent pointer handling. By default, the TCP implements urgent pointer behavior compliant with the 4.2 BSD operating system, that is, this option defaults to <b>0</b>.</p>
	<p><b>TCP_NODELAYACK</b> Specifies whether the TCP needs to send immediate acknowledgment packets to the sender. If this option is not set, TCP delays sending the acknowledgment packets by up to 200 ms. This allows the acknowledgements to be sent along with the data on a response and minimizes the system overhead. If this TCP option is set, it might cause a slight increase in the system overhead, but can result in a higher performance for the network transfers if the sender is waiting on the acknowledgements of the receiver.</p> <p>The TCP protocol level socket options are inherited from listening sockets to new sockets.</p> <p>The following list defines the ATM protocol-level options that are available in the <code>sys/atmsock.h</code> file:</p>
	<p><b>SO_ATM_PARAM</b> Sets the ATM parameters. You can use the <b>SO_ATM_PARAM</b> socket option instead of using the individual sockets options described below. It uses the <code>connect_ie</code> structure that is defined in the <code>sys/call_ie.h</code> file.</p>
	<p><b>SO_ATM_AAL_PARM</b> Sets ATM AAL (Adaptation Layer) parameters. It uses the <code>aal_parm</code> structure that is defined in the <code>sys/call_ie.h</code> file.</p>
	<p><b>SO_ATM_TRAFFIC_DES</b> Sets the ATM Traffic Descriptor values. It uses the <code>traffic</code> structure that is defined in the <code>sys/call_ie.h</code> file.</p>
	<p><b>SO_ATM_BEARER</b> Sets the ATM bearer capability. It uses the <code>bearer</code> structure that is defined in the <code>sys/call_ie.h</code> file.</p>
	<p><b>SO_ATM_BHLI</b> Sets the ATM broadband high layer information. It uses the <code>bhli</code> structure that is defined in the <code>sys/call_ie.h</code> file.</p>
	<p><b>SO_ATM_BLLI</b> Sets the ATM broadband low layer information. It uses the <code>blli</code> structure that is defined in the <code>sys/call_ie.h</code> file.</p>
	<p><b>SO_ATM_QOS</b> Sets the ATM quality of service values. It uses the <code>qos_parm</code> structure that is defined in the <code>sys/call_ie.h</code> file.</p>
	<p><b>SO_ATM_TRANSIT_SEL</b> Sets the ATM Transit Selector Carrier. It uses the <code>transit_sel</code> structure that is defined in the <code>sys/call_ie.h</code> file.</p>
OptionName	<p><b>SO_ATM_ACCEPT</b> Indicates the acceptance of an incoming ATM call that is indicated to the application through the <code>ACCEPT</code> system call. This must be issues for the incoming connection to be fully established. This allows negotiation of the ATM parameters.</p>
	<p><b>SO_ATM_MAX_PEND</b> Sets the number of outstanding transmit buffers that are permitted before an error indication is returned to applications as a result of a transmit operation. This option is only valid for non best effort types of virtual circuits. The <code>OptionValue</code> or <code>OptionLength</code> parameters points to a byte that contains the value that this parameter is set to.</p> <p>The following list defines the <code>IPPROTO_TCP</code> protocol-level options that are available in the <code>netinet/sctp.h</code> file:</p>
	<p><b>SCTP_PEER_ADDR_PARAMS</b> Enables or disables heartbeats for an association and modifies the heartbeat interval of the association. This option uses the <code>sctp_paddrparams</code> structure that is defined in the <code>netinet/sctp.h</code> file. For <code>spp_address</code> field, the AIX supports only wildcard address now. The <b>SPP_HB_ENABLE</b>, <b>SPP_HB_DISABLE</b>, and <b>SPP_HB_TIME_ISZERO</b> flags are supported for the <code>spp_flags</code> field. You can set the <code>spp_hbinterval</code> field to a minimum value of 50 milliseconds.</p>
	<p><b>SCTP_MAXSEG</b> Sets the maximum size of any outgoing <b>SCTP DATA</b> chunk. If the message is larger than the specified size, the SCTP fragments the message into the specified size. It uses the <code>sctp_assoc_value</code> structure that is defined in the <code>netinet/sctp.h</code> file.</p>

Table 1. Parameters (continued)

Item	Description
<i>OptionValue</i>	<p>The <i>OptionValue</i> parameter takes an <b>Int</b> parameter. To enable a Boolean option, you must set the <i>OptionValue</i> parameter to a nonzero value. To disable a Boolean option, set the <i>OptionValue</i> parameter to <b>0</b>.</p> <p>The following options are enabled and disabled in the same manner:</p> <ul style="list-style-type: none"> <li>• <b>SO_DEBUG</b></li> <li>• <b>SO_REUSEADDR</b></li> <li>• <b>SO_KEEPAIVE</b></li> <li>• <b>SO_DONTROUTE</b></li> <li>• <b>SO_BROADCAST</b></li> <li>• <b>SO_OOINLINE</b></li> <li>• <b>SO_LINGER</b></li> <li>• <b>TCP_RFC1323</b></li> </ul>
<i>OptionLength</i>	The <i>OptionLength</i> parameter contains the size of the buffer pointed to by the <i>OptionValue</i> parameter.

Options at other protocol levels vary in format and name.

Table 2. IP level options

Item	Description
<b>IP_DONTFRAG</b>	Sets DF bit from now on for every packet in the IP header. Use the <b>IP_DONTFRAG</b> option to detect decreases in Path MTU (PMTU), UDP applications.
<b>IP_FINDPMTU</b>	Sets enable or disable PMTU discovery for this path. Enable the protocol-level path to MTU discovery for the discovery to happen.
<b>IP_PMTUAGE</b>	Sets the age of the PMTU. Specifies the frequency of the PMT reductions discovery for the session. Setting the PMTU age to <b>0</b> implies an infinite age and PMTU reduction discovery is not attempted and replaces the previously set PMTU age. The new PMTU age is effective after the currently set timer expires. This option is unused because UDP applications must set the <b>IP_DONTFRAG</b> socket to detect decreases in PMTU immediately.
<b>IP_TTL</b>	Sets the time-to-live (TTL) field in the IP header for every packet. However, for raw sockets, the default MAXTTL value is used while sending the messages irrespective of the value set by using the <b>setsockopt</b> subroutine.
<b>IP_HDRINCL</b>	Allows you to build your own IP header. It indicates that the complete IP header is included with the data and can be used only for raw sockets.
<b>IP_ADD_MEMBERSHIP</b>	Joins a multicast group as specified in the <i>OptionValue</i> parameter of the <code>ip_mreq</code> structure type.
<b>IP_DROP_MEMBERSHIP</b>	Leaves a multicast group as specified in the <i>OptionValue</i> parameter of the <code>ip_mreq</code> structure type.
<b>IP_MULTICAST_IF</b>	Permits sending of multicast messages on an interface as specified in the <i>OptionValue</i> parameter of the <code>ip_addr</code> structure type. An address of <code>INADDR_ANY</code> (0x00000000) removes the previous selection of an interface in the multicast options. If you do not specify the interface, the interface leading to the default route is used.
<b>IP_MULTICAST_LOOP</b>	Sets a multicast loopback, determining whether the transmitted messages are delivered to the sending host. An <i>OptionValue</i> parameter of the <code>char</code> type controls the loopback to be ON or OFF.
<b>IP_MULTICAST_TTL</b>	Sets the TTL for multicast packets. An <i>OptionValue</i> parameter of the <code>char</code> type sets the value of TTL that ranges from <b>0</b> to <b>255</b> .
<b>IP_BLOCK_SOURCE</b>	Blocks the data from a given source to a given group.
<b>IP_UNBLOCK_SOURCE</b>	Unblocks a blocked source to undo the <b>IP_BLOCK_SOURCE</b> operation.
<b>IP_ADD_SOURCE_MEMBERSHIP</b>	Joins a source-specific multicast group. If the host is a member of the group, accept data from the source. Otherwise, join the group and accept data from the given source.

Table 2. IP level options (continued)

Item	Description
<b>IP_DROP_SOURCE_MEMBERSHIP</b>	Leaves a source-specific multicast group. Drops the source from the given multicast group list. To drop all sources of a given group, you must use the <b>IP_DROP_MEMBERSHIP</b> socket option.

Table 3. IPv6 level options

Item	Description	Value
<b>IPPROTO_IPV6</b>	Restricts the AF_INET6 sockets to IPv6 communications only.	<b>Option Type</b> int (Boolean interpretation)
	Allows you to set the outgoing hop limit for unicast IPv6 packets.	<b>Option Type</b> int (x) <b>Option Value</b> x < -1 Error <b>EINVAL</b> x == -1 Use kernel default 0 <= x <= 255 Use x x >= 256 Error <b>EINVAL</b>
	Allows you to set the outgoing hop limit for multicast IPv6 packets.	<b>Option Type</b> int (x) <b>Option Value</b> The interpretation is the same as <b>IPV6_UNICAST_HOPS</b> (listed before).
	Allows you to specify the interface that is used for outgoing multicast packets. If specified as <b>0</b> , the system selects the outgoing interface.	<b>Option Type</b> Unsigned int (index of interface to use)
	If a multicast datagram is sent to a group that the sending host belongs to, a copy of the datagram is looped back by the IP layer for local delivery, if the option is set to <b>1</b> . If the option is set to <b>0</b> , a copy is not looped back.	<b>Option Type</b> Unsigned int
	Joins a multicast group on a specified local interface. If the interface index is specified as <b>0</b> , the kernel chooses the local interface.	<b>Option Type</b> struct ipv6_mreq as defined in the netinet/in.h file.
	Leaves a multicast group on a specified interface.	<b>Option Type</b> struct ipv6_mreq as defined in the netinet/in.h file.
	Specifies that the kernel computes a checksum over the data and the pseudo-IPv6 header for a raw socket. The kernel computes the checksum for outgoing packets and verifies the checksum for incoming packets on that socket. Incoming packets with incorrect checksum are discarded. This option is disabled by default.	<b>Option Type</b> int <b>Option Value</b> Offsets into the user data where the checksum result must be stored. This must be a positive even value. If the value is set to <b>-1</b> disables the option.
	Causes the destination IPv6 address and arriving interface index of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.	<b>Option Type</b> int (Boolean interpretation)
	Causes the hop limit of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.	<b>Option Type</b> int (Boolean interpretation)
	Causes the traffic class of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.	<b>Option Type</b> int (Boolean interpretation)
	Causes the routing header of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.	<b>Option Type</b> int (Boolean interpretation)
	Causes the hop-by-hop options header of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.	<b>Option Type</b> int (Boolean interpretation)
	Causes the destination options header of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.	<b>Option Type</b> int (Boolean interpretation)
	Sets the source IPv6 address and outgoing interface index for all IPv6 packets that are sent on this socket. You can clear this option by doing a regular <b>setsockopt</b> with <b>ip6_addr</b> being <b>in6addr_any</b> and <b>ip6_ifindex</b> being <b>0</b> .	<b>Option Type</b> struct in6_pktinfo defined in the netinet/in.h file.
	Sets the next hop for outgoing IPv6 datagrams on this socket. You can clear this option by doing a regular <b>setsockopt</b> with a <b>0</b> length. A memory pointer must be supplied for the option value.	<b>Option Type</b> struct sockaddr_in6 defined in the netinet/in.h file.

Table 3. IPv6 level options (continued)

Item	Description	Value
	Sets the traffic class for outgoing IPv6 datagrams on this socket. To clear this option, the application can specify the value as <b>-1</b> .	<p><b>Option Type</b> int (x)</p> <p><b>Option Value</b> x &lt; -1 Error <b>EINVAL</b> x == -1 Use kernel default 0 &lt;= x &lt;= 255 Use x x &gt;= 256 Error <b>EINVAL</b></p>
	Sets the routing header to be used for outgoing IPv6 datagrams on this socket. This option can be cleared by doing a regular <b>setsockopt</b> with a <b>0</b> length. A memory pointer must still be supplied for the option value in this case.	<p><b>Option Type</b> struct ip6_rthdr defined in the netinet/ip6.h file.</p>
	Sets the hop-by-hop options header to be used for outgoing IPv6 datagrams on this socket. This option can be cleared by doing a regular <b>setsockopt</b> with a <b>0</b> length. A memory pointer must still be supplied for the option value in this case.	<p><b>Option Type</b> struct ip6_hbh defined in the netinet/ip6.h file.</p>
	Sets the destination options header to be used for outgoing IPv6 datagrams on this socket. This header follows a routing header and is used when the routing header is not specified. This option can be cleared by doing a regular <b>setsockopt</b> with a <b>0</b> length. A memory pointer must still be supplied for the option value in this case.	<p><b>Option Type</b> struct ip6_dest defined in the netinet/ip6.h file.</p>
	Sets the destination options header to be used for outgoing IPv6 datagrams on this socket. This header precedes a routing header. If no routing header is specified, this option is silently ignored. This option can be cleared by doing a regular <b>setsockopt</b> with a <b>0</b> length. A memory pointer must still be supplied for the option value in this case.	<p><b>Option Type</b> struct ip6_dest defined in the netinet/ip6.h file.</p>
	Sets this option to control IPv6 path MTU discovery.	<p><b>Option Type</b> int</p> <p><b>Option Value</b> <b>-1</b> Performs path MTU discovery for unicast destinations, but does not perform it for multicast destinations. <b>0</b> performs path MTU discovery. <b>1</b> disables path MTU discovery and sends packets at the minimum MTU.</p>
	Set this option to prevent fragmentation of outgoing IPv6 packets on this socket. If a packet is being sent that is larger than the outgoing interface MTU, the packet is discarded.	<p><b>Option Type</b> int (Boolean interpretation)</p>
	Enables the receipt of the <b>IPV6_PATHMTU</b> ancillary data items by setting this option.	<p><b>Option Type</b> int (Boolean interpretation)</p>
	Sets the address selection preferences for this socket.	<p><b>Option Type</b> int</p> <p><b>Option Value</b> Combination of the <b>IPV6_PREFER_SRC_*</b> flags defined in the netinet/in.h file.</p>
	Joins the multicast group as specified in the <i>OptionValue</i> parameter of the <i>group_req</i> structure. If the specified interface index is <b>0</b> , the kernel chooses the default interface.	<p><b>Option Type</b> struct group_req as defined in the netinet/in.h file.</p>
	Leaves the multicast group as specified in the <i>OptionValue</i> parameter of the <i>group_req</i> structure.	<p><b>Option Type</b> struct group_req as defined in the netinet/in.h file.</p>
	Blocks data from the specified source to the specified multicast group.	<p><b>Option Type</b> struct group_source_req as defined in the netinet/in.h file.</p>
	Unblocks data from the specified source to the specified multicast group. The option is used to undo the <b>MCAST_BLOCK_SOURCE</b> operation.	<p><b>Option Type</b> struct group_source_req as defined in the netinet/in.h file.</p>
	Joins a source-specific multicast group. If the host is already a member of the group, accept data from the specified source. Otherwise, join the <i>group</i> and accept data from the specified source.	<p><b>Option Type</b> struct group_source_req as defined in the netinet/in.h file.</p>

Table 3. IPv6 level options (continued)

Item	Description	Value
	Leaves a source-specific multicast group. Leaves the specified source from the specified multicast group. To leave all sources of the multicast group, you must use the <b>IPV6_LEAVE_GROUP</b> or <b>MCAST_LEAVE_GROUP</b> socket option.	<b>Option Type</b> struct <code>group_source_req</code> as defined in the <code>netinet/in.h</code> file.

Table 4. ICMPV6 level options

Item	Description	Value
<b>IPPROTO_ICMPV6</b>	Filters the ICMPV6 messages by the ICMPV6 type field. Issue a <b>setsockopt</b> call with zero length to clear an existing filter.	<b>Option Type</b> The <code>icmp6_filter</code> structure defined in the <code>netinet/icmp6.h</code> file.

The following values that are defined in the `/usr/include/netinet/tcp.h` file are used by the **setsockopt** subroutine to configure the `dacinet` functions.

```
tcp.h:#define TCP_ACLFLUSH    0x21    /* clear all DACinet ACLs */
tcp.h:#define TCP_ACLCLEAR  0x22    /* clear DACinet ACL */
tcp.h:#define TCP_ACLADD    0x23    /* Add to DACinet ACL */
tcp.h:#define TCP_ACLDEL    0x24    /* Delete from DACinet ACL */
tcp.h:#define TCP_ACLLS     0x25    /* List DACinet ACL */
tcp.h:#define TCP_ACLBIND   0x26    /* Set port number for TCP_ACLLS */
tcp.h:#define TCP_ACLGID    0x01    /* ID being added to ACL is a GID */
tcp.h:#define TCP_ACLUID    0x02    /* ID being added to ACL is a GID */
tcp.h:#define TCP_ACLSUBNET 0x04    /* address being added to ACL is a subnet */
tcp.h:#define TCP_ACLDENY   0x08    /* this ACL entry is for denying access */
```

**Note:** The DACinet facility is available only in a CAPP/EAL4+ configured AIX system.

## Return Values

Upon successful completion, a value of **0** is returned.

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

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

## Error Codes

Table 5. Error codes

Item	Description
<b>EBADF</b>	The <i>Socket</i> parameter is not valid.
<b>EFAULT</b>	The <i>Address</i> parameter is not in a writable part of the user address space.
<b>EINVAL</b>	The <i>OptionValue</i> parameter or the <i>OptionLength</i> parameter is invalid or the socket is shutdown.
<b>ENOBUFS</b>	Insufficient memory for an internal data structure.
<b>ENOTSOCK</b>	The <i>Socket</i> parameter refers to a file, not to a socket.
<b>ENOPROTOOPT</b>	The option is unknown.
<b>EOPNOTSUPP</b>	The socket family or the socket type does not support the <b>EOPNOTSUPP</b> option.
<b>EPERM</b>	The user application does not have the permission to get or to set this socket option. Check the network tunable option.

## Examples

- To mark a socket for broadcasting, enter the following command:

```
int on=1;
setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
```

- To turn on the **TCP\_NODELAYACK** option, enter the following command:

```
int on=1;
setsockopt(s, IPPROTO_TCP, TCP_NODELAYACK, &on, sizeof(on));
```

### Related reference

[sendto Subroutine](#)

[bind Subroutine](#)

### Related information

[no subroutine](#)

## setsourcefilter, getsourcefilter, setipv4sourcefilter, getipv4sourcefilter Subroutine

### Purpose

Manage IP multicast source filters.

### Library

Library (**libc.a**)

### Syntax

```
#include <netinet/in.h>
int setsourcefilter(int socket, uint32_t interface,
                   struct sockaddr *group, socklen_t grouplen,
                   uint32_t fmode, uint_t numsrc,
                   struct sockaddr_storage *slist);

int getsourcefilter(int socket, uint32_t interface,
                   struct sockaddr *group, socklen_t grouplen,
                   uint32_t *fmode, uint_t *numsrc,
                   struct sockaddr_storage *slist);

int setipv4sourcefilter(int socket, struct in_addr interface,
                       struct in_addr group, uint32_t fmode,
                       uint32_t numsrc, struct in_addr *slist);

int getipv4sourcefilter(int socket, struct in_addr interface,
                       struct in_addr group, uint32_t *fmode,
                       uint32_t *numsrc, struct in_addr *slist);
```

### Description

The **setsourcefilter** and **setipv4sourcefilter** subroutines allow a socket to join a multicast group on an interface while excluding (fmode = MCAST\_EXCLUDE) messages or accepting (fmode = MCAST\_INCLUDE) messages from a number of senders listed in the slist table. The number of elements in the slist is specified by numsrc.

The **getsourcefilter** and **getipv4sourcefilter** subroutines provide information on existing source filter for a socket on a given interface and for a given multicast group. fmode, numsrc and slist are pointers to parameters which will contain the information returned by the subroutine. fmode will point to the type of filter returned: MCAST\_EXCLUDE or MCAST\_INCLUDE. On input, numsrc points to the maximum number of senders that the application is expecting. If there are more sources than requested, the subroutine returns only the first numsrc sources in slist and numsrc is set to indicate the total number of sources.

slist contains the table of excluded or included senders depending on the type of the filter. Memory pointed by fmode, numsrc and slist must be allocated by the application. In particular, slist must point to a memory zone able to contain numsrc elements.

The **setipv4sourcefilter** and **getipv4sourcefilter** can only be used for AF\_INET sockets.

The **setsourcefilter** and **getsourcefilter** can be used for AF\_INET and AF\_INET6 sockets.

## Parameters

For setsourcefilter and setipv4sourcefilter:

Item	Description
<b>socket</b>	Specifies the unique socket name
<b>interface</b>	Specifies the local interface. For <b>setipv4sourcefilter</b> and <b>getipv4sourcefilter</b> an address configured on the interface must be specified. For <b>setsourcefilter</b> and <b>getsourcefilter</b> , the interface must be specified by its interface index.
<b>group</b>	Specifies the multicast group
<b>fmode</b>	Specifies if the elements contained in the slist must be excluded (MCAST_EXCLUDE) or included (MCAST_INCLUDE)
<b>numsrc</b>	Specifies the number of elements in slist
<b>slist</b>	Specifies the list of elements to exclude or include.

For getsourcefilter and getipv4sourcefilter:

Item	Description
<b>socket</b>	Specifies the unique socket name
<b>interface</b>	Specifies the local interface. For <b>setipv4sourcefilter</b> and <b>getipv4sourcefilter</b> an address configured on the interface must be specified. For <b>setsourcefilter</b> and <b>getsourcefilter</b> the interface must be specified by its interface index.
<b>group</b>	Specifies the multicast group
<b>fmode</b>	Specifies a pointer to the type of element returned in slist. MCAST_EXCLUDE for a list of excluded elements MCAST_INCLUDE for a list of included elements.
<b>numsrc</b>	On input, specifies the number of elements that can be returned in slist. On output, contains the total number of sources for this filter
<b>slist</b>	Contains the list of elements returned.

## Return Values

Upon successful completion, the subroutine returns 0.

If unsuccessful, the subroutine returns -1 and errno is set accordingly.

## shutdown Subroutine

### Purpose

Shuts down all socket send and receive operations.

### Library

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

## Syntax

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

```
int shutdown ( Socket, How)  
int Socket, How;
```

## Description

The **shutdown** subroutine disables all [receive](#) and [send](#) operations on the specified socket.

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

## Parameters

Item	Description
<i>Socket</i>	Specifies the unique name of the socket.
<i>How</i>	Specifies the type of subroutine shutdown. Use the following values: <b>0</b> Disables further receive operations. <b>1</b> Disables further send operations. <b>2</b> Disables further send operations and receive operations.

## Return Values

Upon successful completion, a value of 0 is returned.

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

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable see [Error Notification Object Class](#) in *General Programming Concepts: Writing and Debugging Programs*.

## Error Codes

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

Error	Description
<b>EBADF</b>	The <i>Socket</i> parameter is not valid.
<b>EINVAL</b>	The <i>How</i> parameter is invalid.
<b>ENOTCONN</b>	The socket is not connected.
<b>ENOTSOCK</b>	The <i>Socket</i> parameter refers to a file, not a socket.

## Files

Item	Description
<u><a href="#">/usr/include/sys/socket.h</a></u>	Contains socket definitions.
<u><a href="#">/usr/include/sys/types.h</a></u>	Contains definitions of unsigned data types.

## Related reference

[getsockopt Subroutine](#)

[recv Subroutine](#)

## Related information

[read subroutine](#)

[Sockets Overview](#)

# SLPAttrCallback Subroutine

## Purpose

Returns the same callback type as the **SLPFindAttrs()** function.

## Syntax

```
typedef SLPBoolean SLPAttrCallback(SLPHandle hSLP,  
                                   const char* pcAttrList,  
                                   SLPErrCode errCode,  
                                   void *pvCookie);
```

## Description

The **SLPAttrCallback** type is the type of the callback function parameter to the **SLPFindAttrs()** function.

The *pcAttrList* parameter contains the requested attributes as a comma-separated list (or is empty if no attributes matched the original tag list).

## Parameters

Item	Description
<i>hSLP</i>	The <b>SLPHandle</b> used to initiate the operation.
<i>pcAttrList</i>	A character buffer containing a comma-separated, null-terminated list of attribute ID/value assignments, in SLP wire format: "(attr-id=attr-value-list)"
<i>errCode</i>	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 <b>SLP_OK</b> , then the API library can choose to terminate the outstanding operation.
<i>pvCookie</i>	Memory passed down from the client code that called the original API function, starting the operation. Can be NULL.

## Return Values

The client code should return **SLP\_TRUE** if more data is desired; otherwise **SLP\_FALSE** is returned.

## Related information

[/etc/slp.conf File](#)

[Service Location Protocol \(SLP\) API](#)

## SLPClose Subroutine

## Purpose

Frees all resources associated with the handle.

## Syntax

```
void SLPClose(SLPHandle hSLP);
```

## Description

The **SLPClose** subroutine frees all resources associated with the handle. If the handle was 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

Item	Description
<i>hSLP</i>	The <b>SLPHandle</b> handle returned from a call to <b>SLPOpen()</b> .

## Related information

[/etc/slp.conf File](#)

[Service Location Protocol \(SLP\) API](#)

## SLPDereg Subroutine

### Purpose

Deregisters the advertisement for URL in all scopes and locales.

### Syntax

```
SLPError SLPDereg(hSLP, pcURL, callback, pvCookie)  
SLPHandle hSLP;  
const char *pcURL;  
SLPRegReport callback;  
void *pvCookie;
```

## Description

The **SLPDereg** subroutine deregisters the advertisement for the URL specified by the *pcURL* parameter in all scopes where the service is registered and in all language locales. The deregistration is not confined to the **SLPHandle** locale. Deregistration takes place in all locales.

## Parameters

Item	Description
<i>hSLP</i>	The language-specific <b>SLPHandle</b> handle used for deregistration of services.
<i>pcURL</i>	The URL that needs to be deregistered.
<i>callback</i>	A callback function through which the results of the operation are reported.
<i>pvCookie</i>	The memory passed to callback code from the client. The parameter can be set to NULL.

## Return Values

Item	Description
<b>SLP_OK</b>	The subroutine has run successfully.
<b>SLPError</b>	An error occurred.

## Related information

[/etc/slp.conf subroutine](#)

[Service Location Protocol \(SLP\) APIs](#)

## SLPEscape Subroutine

### Purpose

Processes an input string and escapes any characters reserved for SLP.

### Syntax

```
SLPError SLPEscape(const char* pcInbuf,  
                  char** ppcOutBuf,  
                  SLPBoolean isTag);
```

### Description

The **SLPEscape** subroutine processes the input string in *pcInbuf* and escapes any characters reserved for SLP. If the *isTag* parameter is **SLPTrue**, **SLPEscape** 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()** when the memory is no longer needed.

### Parameters

Item	Description
<i>pcInbuf</i>	Pointer to the input buffer to process for escape characters.
<i>ppcOutBuf</i>	Pointer to a pointer for the output buffer with the characters reserved for SLP escaped. Must be freed using <b>SLPFree()</b> when the memory is no longer needed.
<i>isTag</i>	When true, the input buffer is checked for bad tag characters.

### Return Values

The **SLPEscape** subroutine returns **SLP\_PARSE\_ERROR** if any characters are bad tag characters and the *isTag* flag is true; otherwise, it returns **SLP\_OK**, or the appropriate error code if another error occurs.

## Related information

[/etc/slp.conf File](#)

[Service Location Protocol \(SLP\) API](#)

## SLPFindAttrs Subroutine

### Purpose

Returns service attributes that match the attribute IDs for the indicated service URL or service type.

## Syntax

```
SLPError SLPFindAttrs(SLPHandle hSLP,
                     const char *pcURLOrServiceType,
                     const char *pcScopeList,
                     const char *pcAttrIds,
                     SLPAttrCallback callback,
                     void *pvCookie);
```

## Description

The **SLPFindAttrs** subroutine returns service attributes matching the attribute IDs for the indicated service URL or service type. If **pcURLOrServiceType** is a service URL, the attribute information returned is for that particular advertisement in the language locale of the *SLPHandle*.

If **pcURLOrServiceType** is a service type name (including naming authority if any), then the attributes for all advertisements of that service type are returned regardless of the language of registration. Results are returned through the *callback*.

The result is filtered with an SLP attribute request filter string parameter. If the filter string is the empty string (""), all attributes are returned.

## Parameters

Item	Description
<i>hSLP</i>	The <b>SLPHandle</b> on which to search for attributes.
<i>pcURLOrServiceType</i>	The service URL or service type. Cannot be the empty string.
<i>pcScopeList</i>	A pointer to a char containing a comma-separated list of scope names. Cannot be the empty string, "".
<i>pcAttrIds</i>	The filter string indicating which attribute values to return. Use the empty string ("") to indicate all values. Wildcards matching all attribute IDs having a particular prefix or suffix are also possible.
<i>callback</i>	A callback function through which the results of the operation are reported.
<i>pvCookie</i>	Memory passed to the callback code from the client. Can be NULL.

## Return Values

If **SLPFindAttrs** is successful, it returns **SLP\_OK**. If an error occurs in starting the operation, one of the **SLPError** codes is returned.

## Related information

[/etc/slp.conf File](#)

[Service Location Protocol \(SLP\) API](#)

## SLPFindScopes Subroutine

### Purpose

Sets the *ppcScopeList* parameter to point to a comma-separated list that includes all available scope values.

### Syntax

```
SLPError SLPFindScopes(SLPHandle hSLP,
                      char** ppcScopeList);
```

## Description

The **SLPFindScopes** subroutine sets the *ppcScopeList* parameter to point to a comma-separated list that includes all available scope values. If there is any order to the scopes, preferred scopes are listed before less desirable scopes. There is always at least one name in the list, the default scope, **DEFAULT**.

## Parameters

Item	Description
<i>hSLP</i>	The <b>SLPHandle</b> on which to search for scopes.
<i>ppcScopeList</i>	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 <b>SLPFree()</b> .

## Return Values

If no error occurs, **SLPFindScopes** returns **SLP\_OK**; otherwise, it returns the appropriate error code.

## Related information

[/etc/slp.conf File](#)

[Service Location Protocol \(SLP\) API](#)

## SLPFindSrvs Subroutine

### Purpose

Issues the query for services on the language-specific **SLPHandle** and returns the results through the *callback*.

### Syntax

```
SLPError SLPFindSrvs(SLPHandle hSLP,  
                    const char *pcServiceType,  
                    const char *pcScopeList,  
                    const char *pcSearchFilter,  
                    SLPsrvURLCallback callback,  
                    void *pvCookie);
```

## Description

The **SLPFindSrvs** subroutine issues the query for services on the language-specific **SLPHandle** and returns the results through the callback. The parameters determine the results

## Parameters

Item	Description
<i>hSLP</i>	The language-specific <b>SLPHandle</b> on which to search for services.
<i>pcServiceType</i>	The Service Type String, including authority string if any, for the request, which can be discovered using <b>SLPSrvTypes()</b> . This could be, for example, "service:printer:lpr" or "service:nfs". This cannot be the empty string ("").
<i>pcScopeList</i>	A pointer to a char containing a comma-separated list of scope names. This cannot be the empty string ("").

Item	Description
<i>pcSearchFilter</i>	A query formulated of attribute pattern matching expressions in the form of a LDAPv3 Search Filter. If this filter is empty (""), all services of the requested type in the specified scopes are returned.
<i>callback</i>	A callback function through which the results of the operation are reported.
<i>pvCookie</i>	Memory passed to the callback code from the client. Can be NULL.

## Return Values

If **SLPFindSrvs** is successful, it returns **SLP\_OK**. If an error occurs in starting the operation, one of the **SLPError** codes is returned.

### Related information

[/etc/slp.conf File](#)

[Service Location Protocol \(SLP\) API](#)

## SLPFindSrvTypes Subroutine

### Purpose

Issues an SLP service type request.

### Syntax

```
SLPError SLPFindSrvTypes(SLPHandle hSLP,
                        const char *pcNamingAuthority,
                        const char *pcScopeList,
                        SLPsrvTypeCallback callback,
                        void *pvCookie);
```

### Description

The **SLPFindSrvType()** subroutine 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 naming authority indicated by *pcNamingAuthority*.

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, and it returns a **PARAMETER\_BAD** error when it is included explicitly.

The service type names are returned with the naming authority intact. If the naming authority is the default (that is, the empty string), then it is omitted, as is the separating ".". Service type names from URLs of the **service:** scheme are returned with the "service:" prefix intact.

### Parameters

Item	Description
<i>hSLP</i>	The <b>SLPHandle</b> on which to search for types.
<i>pcNamingAuthority</i>	The naming authority to search. Use "*" for all naming authorities and the empty string, "", for the default naming authority.
<i>pcScopeList</i>	A pointer to a char containing a comma-separated list of scope names to search for service types. Cannot be the empty string, "".

Item	Description
<i>callback</i>	A callback function through which the results of the operation are reported.
<i>pvCookie</i>	Memory passed to the callback code from the client. Can be NULL.

## Return Values

If **SLPFindSrvTypes** is successful, it returns **SLP\_OK**. If an error occurs in starting the operation, one of the **SLPError** codes is returned.

### Related information

[/etc/slp.conf File](#)

[Service Location Protocol \(SLP\) API](#)

## SLPFree Subroutine

### Purpose

Frees memory returned from **SLPParseSrvURL()**, **SLPFindScopes()**, **SLPEscape()**, and **SLPUnescape()**.

### Syntax

```
void SLPFree(void* pvMem);
```

### Description

The **SLPFree** subroutine frees memory returned from **SLPParseSrvURL()**, **SLPFindScopes()**, **SLPEscape()**, and **SLPUnescape()**.

### Parameters

Item	Description
<i>pvMem</i>	A pointer to the storage allocated by the <b>SLPParseSrvURL()</b> , <b>SLPEscape()</b> , <b>SLPUnescape()</b> , or <b>SLPFindScopes()</b> function. Ignored if NULL.

### Related information

[/etc/slp.conf File](#)

[Service Location Protocol \(SLP\) API](#)

## SLPGetProperty Subroutine

### Purpose

Returns the value of the corresponding SLP property name.

### Syntax

```
const char* SLPGetProperty(const char* pcName);
```

## Description

The **SLPGetProperty** subroutine returns the value of the corresponding SLP property name. The returned string is owned by the library and *must not* be freed.

## Parameters

Item	Description
<i>pcName</i>	Null-terminated string with the property name.

## Return Values

If no error, the **SLPGetProperty** subroutine returns a pointer to a character buffer containing the property value. If the property was not set, the subroutine returns the default value. If an error occurs, it returns NULL. The returned string *must not* be freed.

## Related information

[/etc/slp.conf File](#)

[Service Location Protocol \(SLP\) API](#)

## SLPOpen Subroutine

### Purpose

Returns an **SLPHandle** handle that encapsulates the language locale for SLP requests.

### Syntax

```
SLPError SLPOpen(const char *pcLang, SLPBoolean isAsync, SLPHandle  
*phSLP);
```

## Description

The **SLPOpen** subroutine returns an **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. However, 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.

## Implementation Specifics

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()** API function terminates any outstanding calls on the handle. If an implementation is unable to support an asynchronous (resp. synchronous) operation, because of memory constraints or lack of threading support, the **SLP\_NOT\_IMPLEMENTED** flag might be returned when the *isAsync* flag is **SLP\_TRUE** (resp. **SLP\_FALSE**).

## Parameters

Item	Description
<i>pcLang</i>	A pointer to an array of characters (AIX supports "en" only).

Item	Description
<i>isAsync</i>	An <b>SLPBoolean</b> indicating whether the <b>SLPHandle</b> should be opened for asynchronous operation or not. AIX supports synchronous operation only.
<i>phSLP</i>	A pointer to an <b>SLPHandle</b> , in which the open <b>SLPHandle</b> is returned. If an error occurs, the value upon return is NULL.

## Return Values

If **SLPOpen** is successful, it returns **SLP\_OK** and an **SLPHandle** handle in the *phSLP* parameter for the language locale passed in as the *pcLang* parameter.

## Error Codes

Item	Description
<b>SLPError</b>	Indicates the status of the operation

## Related information

[/etc/slp.conf File](#)

[Service Location Protocol \(SLP\) API](#)

## SLPParseSrvURL Subroutine

### Purpose

Parses the URL passed in as the argument into a service URL structure and returns it in the *ppSrvURL* pointer.

### Syntax

```
SLPError SLPParseSrvURL(char *pcSrvURL
                        SLPSrvURL** ppSrvURL);
```

### Description

The **SLPParseSrvURL** subroutine parses the URL passed in as the argument into a service URL structure and returns it in the *ppSrvURL* pointer. If a parse error occurs, returns **SLP\_PARSE\_ERROR**. The input buffer *pcSrvURL* is destructively modified during the parse and used to fill in the fields of the return structure. The structure returned in *ppSrvURL* should be freed with **SLPFreeURL()**. If the URL has no service part, the **s\_pcSrvPart** string is the empty string (""), 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\_pcTransport** field is the empty string. If the transport is not IP or there is no port number, the **s\_iPort** field is 0.

### Parameters

Item	Description
<i>pcSrvURL</i>	A pointer to a character buffer containing the null-terminated URL string to parse. It is destructively modified to produce the output structure.
<i>ppSrvURL</i>	A pointer to a pointer for the <b>SLPSrvURL</b> structure to receive the parsed URL. The memory should be freed by a call to <b>SLPFree()</b> when no longer needed.

## Return Values

If no error occurs, the return value is **SLP\_OK**. Otherwise, the appropriate error code is returned.

## Related information

[/etc/slp.conf File](#)

[Service Location Protocol \(SLP\) API](#)

## SLPReg Subroutine

### Purpose

Registers the services on the language-specific **SLPHandle** handle and returns the results through the callback.

### Syntax

```
SLPError SLPReg (hSLP, pcSrvURL,
usLifetime, pcSrvType,
pcAttrs, fresh,
callback, pvCookie)
SLPHandle hSLP;
const char *pcSrvURL;
const unsigned short usLifetime;
const char *pcSrvType;
const char *pcAttrs;
SLPBoolean fresh;
SLPRegReport callback;
void *pvCookie;
```

### Description

The **SLPReg** subroutine registers the URL specified by the *pcSrvURL* parameter having the *usLifeTime* lifetime with the attribute list specified by the *pcAttrs* parameter. The attribute list is a comma-separated list of attributes. The *pcSrvType* parameter is the service type name and can be included in the **scheme** service URL that are not in the service. In the case of the **scheme** service URL with service, the *pcSrvType* parameter is ignored. The *fresh* flag specifies that this registration is a new or an update-only registration. If the *fresh* parameter is set to **SLP\_TRUE**, the registration replaces existing registrations. If the *fresh* parameter is set to **SLP\_FALSE**, the registration only updates existing registrations. The *usLifeTime* parameter must be nonzero and less than or equal to **SLP\_LIFETIME\_MAXIMUM**. The registration takes place in the language locale of **hhSLP** handle.

### Parameters

Item	Description
<i>hSLP</i>	The language-specific <b>SLPHandle</b> handle on which to register the services.
<i>pcSrvURL</i>	The URL that needs to be registered.
<i>usLifetime</i>	The time after which the registered URL will expire.
<i>pcSrvType</i>	Specifies the service type name that can be included in the service URL, which is not in the <b>scheme</b> service.
<i>pcAttrs</i>	The comma-separated list of attributes to be registered along with the service URL.
<i>fresh</i>	If the <i>fresh</i> parameter is set to <b>SLP_TRUE</b> , the registration is new; if the <i>fresh</i> parameter is set to <b>SLP_FALSE</b> , this registration updates an existing registration.
<i>callback</i>	A callback function through which the results of the operation are reported.

Item	Description
<i>pvCookie</i>	The memory passed to callback code from the client. The parameter can be set to NULL.

## Return Values

Item	Description
<b>SLP_OK</b>	The subroutine has run successfully.
<b>SLPError</b>	An error occurred.

## Related information

[/etc/slp.conf subroutine](#)

[Service Location Protocol \(SLP\) APIs](#)

## SLPRegReport Callback Subroutine

### Introduction

Returns the same callback type as the **SLPReg** and **SLPDereg** subroutines.

### Syntax

```
typedef void SLPRegReport (hSLP, errCode, pvCookie)
SLPHandle hSLP;
SLPError errCode;
void *pvCookie;
```

### Description

The **SLPSrvURLCallback** type is the type of the callback subroutine parameter to the **SLPFindSrvs** subroutine.

### Parameters

Item	Description
<i>hSLP</i>	The <b>SLPHandle</b> handle used to initiate the operation.
<i>errCode</i>	An error code indicating that an error occurred during the operation. The callback must check this error code before processing the parameters. If the error code is not SLP_OK, the API library can choose to terminate the outstanding operation.
<i>pvCookie</i>	The memory passed down from the client code that calls the original API function at the start of the operation. The parameter can be set to NULL.

### Return Values

Item	Description
<b>SLP_TRUE</b>	More data is necessary.
<b>SLP_FALSE</b>	No additional data is necessary.

## SLPSrvTypeCallback Subroutine

## Purpose

Returns the same callback type as the **SLPFindSrvTypes()** function.

## Syntax

```
typedef SLPBoolean SLPsrvTypeCallback(SLPHandle hSLP,  
                                     const char* pcSrvTypes,  
                                     SLPErrCode errCode,  
                                     void *pvCookie);
```

## Description

The **SLPsrvTypeCallback** type is the type of the callback function parameter to the **SLPFindSrvTypes()** function.

## Parameters

Item	Description
<i>hSLP</i>	The <b>SLPHandle</b> used to initiate the operation.
<i>pcSrvTypes</i>	A character buffer containing a comma-separated, null-terminated list of service types.
<i>errCode</i>	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 <b>SLP_OK</b> , then the API library can choose to terminate the outstanding operation.
<i>pvCookie</i>	Memory passed down from the client code that called the original API function, starting the operation. Can be NULL.

## Return Values

The client code should return **SLP\_TRUE** if more data is desired; otherwise **SLP\_FALSE** is returned.

## SLPsrvURLCallback Subroutine

### Purpose

Returns the same callback type as the **SLPFindSrvs()** function.

### Syntax

```
typedef SLPBoolean SLPsrvURLCallback(SLPHandle hSLP,  
                                     const char* pcSrvURL,  
                                     unsigned short sLifetime,  
                                     SLPErrCode errCode,  
                                     void *pvCookie);
```

### Description

The **SLPsrvURLCallback** type is the type of the callback function parameter to the **SLPFindSrvs()** function.

## Parameters

Item	Description
<i>hSLP</i>	The <b>SLPHandle</b> used to initiate the operation.
<i>pcSrvURL</i>	A character buffer containing the returned service URL.
<i>sLifetime</i>	An unsigned short giving the lifetime of the service advertisement, in seconds. The value must be an unsigned integer less than or equal to <b>SLP_LIFETIME_MAXIMUM</b> .
<i>errCode</i>	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 <b>SLP_OK</b> , then the API library can choose to terminate the outstanding operation.
<i>pvCookie</i>	Memory passed down from the client code that called the original API function, starting the operation. Can be NULL.

## Return Values

The client code should return **SLP\_TRUE** if more data is desired; otherwise **SLP\_FALSE** is returned.

### Related information

[/etc/slp.conf File](#)

[Service Location Protocol \(SLP\) API](#)

## SLPUnescape Subroutine

### Purpose

Processes an input string and unescapes any characters reserved for SLP.

### Syntax

```
SLPError SLPUnescape(const char* pcInbuf,  
                    char** ppcOutBuf,  
                    SLPBoolean isTag);
```

### Description

The **SLPUnescape** subroutine processes the input string in *pcInbuf* and unescapes any characters reserved for SLP. If the *isTag* parameter is **SLPTrue**, **SLPUnescape** looks for bad tag characters and signals an error if any are found by returning the **SLP\_PARSE\_ERROR** code. No transformation is performed if the input string is 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()** when the memory is no longer needed.

## Parameters

Item	Description
<i>pcInbuf</i>	Pointer to the input buffer to process for escape characters.
<i>ppcOutBuf</i>	Pointer to a pointer for the output buffer with the characters reserved for SLP escaped. Must be freed using <b>SLPFree()</b> when the memory is no longer needed.
<i>isTag</i>	When true, the input buffer is checked for bad tag characters.

## Return Values

The **SLPUnescape** subroutine returns **SLP\_PARSE\_ERROR** if any characters are bad tag characters and the *isTag* flag is true; otherwise, it returns **SLP\_OK**, or the appropriate error code if another error occurs.

## Related information

[/etc/slp.conf File](#)

[Service Location Protocol \(SLP\) API](#)

## socket Subroutine

### Purpose

Creates an end point for communication and returns a descriptor.

### Library

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

### Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
int socket ( AddressFamily, Type, Protocol )
int AddressFamily, Type, Protocol;
```

### Description

The **socket** subroutine creates a socket in the specified *AddressFamily* and of the specified type. A protocol can be specified or assigned by the system. If the protocol is left unspecified (a value of 0), the system selects an appropriate protocol from those protocols in the address family that can be used to support the requested socket type.

The **socket** subroutine returns a descriptor (an integer) that can be used in later subroutines that operate on sockets.

Socket level options control socket operations. The **getsockopt** and **setsockopt** subroutines are used to get and set these options, which are defined in the **/usr/include/sys/socket.h** file.

## Parameters

Item	Description
<i>AddressFamily</i>	<p>Specifies an address family with which addresses specified in later socket operations should be interpreted. The <b>/usr/include/sys/socket.h</b> file contains the definitions of the address families. Commonly used families are:</p> <p><b>AF_UNIX</b> Denotes the operating system path names.</p> <p><b>AF_INET</b> Denotes the ARPA Internet addresses.</p> <p><b>AF_INET6</b> Denotes the <b>IPv6</b> and <b>IPv4</b> addresses.</p> <p><b>AF_NS</b> Denotes the XEROX Network Systems protocol.</p> <p><b>AF_BYPASS</b> Denotes the kernel-bypass protocol domain (for example, the protocols that operate on the InfiniBand domain).</p>
<i>Type</i>	<p>Specifies the semantics of communication. The <b>/usr/include/sys/socket.h</b> file defines the socket types. The operating system supports the following types:</p> <p><b>SOCK_STREAM</b> Provides sequenced, two-way byte streams with a transmission mechanism for out-of-band data.</p> <p><b>SOCK_DGRAM</b> Provides datagrams, which are connectionless messages of a fixed maximum length (usually short).</p> <p><b>SOCK_RAW</b> Provides access to internal network protocols and interfaces. This type of socket is available only to the root user, or to non-root users who have the CAP_NUMA_ATTACH capability. (For non-root raw socket access, the CAP_NUMA_ATTACH capability, along with CAP_PROPAGATE, is assigned using the <a href="#">chuser Command</a>).</p> <p><b>SOCK_SEQPACKET</b> Provides sequenced, reliable, and unduplicated flow of information. This type of socket is used for UDP-style socket creation in case of Stream Control Transmission Protocol and Reliable Datagram Sockets (RDS) Protocol.</p>
<i>Protocol</i>	<p>Specifies a particular protocol to be used with the socket. Specifying the <i>Protocol</i> parameter of 0 causes the <b>socket</b> subroutine to default to the typical protocol for the requested type of returned socket. For SCTP sockets, the protocol parameter is IPPROTO_SCTP. For RDS sockets, the <i>Protocol</i> parameter is BYPASSPROTO_RDS.</p>

## Return Values

Upon successful completion, the **socket** subroutine returns an integer (the socket descriptor).

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

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable see [Error Notification Object Class](#).

## Error Codes

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

Error	Description
<b>EAFNOSUPPORT</b>	The addresses in the specified address family cannot be used with this socket.
<b>EMFILE</b>	The per-process descriptor table is full.
<b>ENOBUFS</b>	Insufficient resources were available in the system to complete the call.
<b>ESOCKTNOSUPPORT</b>	The socket in the specified address family is not supported.

## Examples

The following program fragment illustrates the use of the **socket** subroutine to create a datagram socket for on-machine use:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

## Implementation Specifics

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

The socket applications can be compiled with **COMPAT\_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

### Related reference

[accept Subroutine](#)

[bind Subroutine](#)

[getsockname Subroutine](#)

### Related information

[ioctl subroutine](#)

[Initiating Internet Stream Connections Example Program](#)

## socketpair Subroutine

### Purpose

Creates a pair of connected sockets.

### Library

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

### Syntax

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

```
int socketpair (Domain, Type, Protocol, SocketVector[0])  
int Domain, Type, Protocol;  
int SocketVector[2];
```

### Description

The **socketpair** subroutine creates an unnamed pair of connected sockets in a specified domain, of a specified type, and using the optionally specified protocol. The two sockets are identical.

**Note:** Create sockets with this subroutine only in the **AF\_UNIX** protocol family.

The descriptors used in referencing the new sockets are returned in the *SocketVector*[0] and *SocketVector*[1] parameters.

The */usr/include/sys/socket.h* file contains the definitions for socket domains, types, and protocols.

All applications containing the **socketpair** subroutine must be compiled with the **\_BSD** macro set to a value of 43 or 44. Socket applications must include the BSD **libbsd.a** library.

## Parameters

Item	Description
<i>Domain</i>	Specifies the communications domain within which the sockets are created. This subroutine does not create sockets in the Internet domain.
<i>Type</i>	Specifies the communications method, whether <b>SOCK_DGRAM</b> or <b>SOCK_STREAM</b> , that the socket uses.
<i>Protocol</i>	Points to an optional identifier used to specify which standard set of rules (such as <u>UDP/IP</u> and <u>TCP/IP</u> ) governs the transfer of data.
<i>SocketVector</i>	Points to a two-element vector that contains the integer descriptors of a pair of created sockets.

## Return Values

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

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

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

## Error Codes

If the **socketpair** subroutine is unsuccessful, it returns one of the following errors codes:

Error	Description
<b>EAFNOSUPPORT</b>	The addresses in the specified address family cannot be used with this socket.
<b>EFAULT</b>	The <i>SocketVector</i> parameter is not in a writable part of the user address space.
<b>EMFILE</b>	This process has too many descriptors in use.
<b>ENFILE</b>	The maximum number of files allowed are currently open.
<b>ENOBUFS</b>	Insufficient resources were available in the system to perform the operation.
<b>EOPNOTSUPP</b>	The specified protocol does not allow the creation of socket pairs.
<b>EPROTONOSUPPORT</b>	The specified protocol cannot be used on this system.
<b>EPROTOTYPE</b>	The socket type is not supported by the protocol.

## Related information

[Socketpair Communication Example Program](#),  
[Sockets Overview](#),

## socks5\_getserv Subroutine

## Purpose

Return the address of the SOCKSv5 server (if any) to use when connecting to a given destination.

## Library

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

## Syntax

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

```
struct sockaddr * socks5_getserv (Dst, DstLen)
struct sockaddr *Dst;
size_t DstLen;
```

## Description

The **socks5\_getserv** subroutine determines which (if any) SOCKSv5 server should be used as an intermediary when connecting to the address specified in *Dst*.

The address returned in *Dst* may be IPv4 or IPv6 or some other family. The user should check the address family before using the returned data.

The socket applications can be compiled with **COMPAT\_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

## Parameters

Item	Description
<i>Dst</i>	Specifies the external address of the target socket to use as a key for looking up the appropriate SOCKSv5 server.
<i>DstLength</i>	Specifies the length of the address structure in <i>Dst</i> .

## Return Values

- Upon successful lookup, the **socks\_getserv** subroutine returns a reference to a **sockaddr** struct.
- If the **socks5tcp\_connect** subroutine is unsuccessful in finding a server, for any reason, a value of **NULL** is returned. If an error occurred, an error code, indicating the generic system error, is moved into the **errno** global variable.

## Error Codes (placed in errno)

The **socks5\_getserv** subroutine is unsuccessful if no server is indicated or if any of the following errors occurs:

Error	Description
<b>EAFNOSUPPORT</b>	The addresses in the specified address family cannot be used with this socket.
<b>EFAULT</b>	The <i>Dst</i> parameter is not in a writable part of the user address space.
<b>EINVAL</b>	One or more of the specified arguments is invalid.
<b>ENOMEM</b>	The <i>Dst</i> parameter is not large enough to hold the server address.

## Examples

The following program fragment illustrates the use of the `socks5_getserv` subroutine by a client to request a connection from a server's socket.

```
struct sockaddr_in6 dst;
struct sockaddr *srv;
.
.
.
srv = socks5_getserv((struct sockaddr*)&dst, sizeof(dst));
if (srv !=NULL) {
    /* Success: srv should be used as the socks5 server */
} else {
    /* Failure: no server could be returned.  check errno */
}
}
```

### Related reference

[connect Subroutine](#)

### Related information

[Sockets Overview](#)

[SOCKS5C\\_CONFIG Environment Variable](#)

## /etc/socks5c.conf File

### Purpose

Contains mappings between network destinations and SOCKSv5 servers.

### Description

The `/etc/socks5c.conf` file contains basic mappings between network destinations (hosts or networks) and SOCKSv5 servers to use when accessing those destinations. This is an ASCII file that contains records for server mappings. Text following a pound character ('#') is ignored until the end of line. Each record appears on a single line and is the following format:

```
<destination>[/<prefixlength>] <server>[:<port>]
```

You must separate fields with whitespace. Records are separated by new-line characters. The fields and modifiers in a record have the following values:

Item	Description
<i>destination</i>	Specifies a network destination; <i>destination</i> may be either a name fragment or a numeric address (with optional <i>prefixlength</i> ). If <i>destination</i> is an address, it may be either IPv4 or IPv6.
<i>prefixlength</i>	If specified, indicates the number of leftmost (network order) bits of an address to use when comparing to this record. Only valid if <i>destination</i> is an address. If not specified, all bits are used in comparisons.
<i>server</i>	Specifies the SOCKSv5 server associated with <i>destination</i> . If <i>server</i> is "NONE" (must be all uppercase), this record indicates that target addresses matching <i>destination</i> should not use any SOCKSv5 server, but rather be contacted directly.
<i>port</i>	If specified, indicates the port to use when contacting <i>server</i> . If not specified, the default of 1080 is assumed.

**Note:** Server address in IPv6 format **must** be followed by a port number.

If a name fragment *destination* is present in **/etc/socks5c.conf**, all target addresses is SOCKSv5 operations will be converted into hostnames for name comparison (in addition to numeric comparisons with numeric records). The resulting hostname is considered to match if the last characters in the hostname match the specified name fragment.

When using this configuration information to determine the address of the appropriate SOCKSv5 server for a target destination, the "best" match is used. The "best" match is defined as:

Item	Description
<i>destination</i> is numeric	Most bits in comparison (i.e. largest <i>prefixlength</i> )
<i>destination</i> is a name fragment	Most characters in name fragment.

When both name fragment and numeric addresses are present, all name fragment entries are "better" than numeric address entries.

Two implicit records:

```
0.0.0.0/0 NONE #All IPv4 destinations; no associated server.
```

```
::/0 NONE #All IPv6 destinations; no associated server.
```

are assumed as defaults for all destinations not specified in **/etc/socks5c.conf**.

## Security

Access Control: This file should grant read (r) access to all users and grant write (w) access only to the root user.

## Examples

```
#Sample socks5c.conf file
```

```
9.0.0.0/8 NONE #Direct communication with all hosts in the 9 network.  
129.35.0.0/16 sox1.austin.ibm.com  
ibm.com NONE #Direct communication will all hosts matching "ibm.com" (e.g.  
"aguila.austin.ibm.com")
```

## Related reference

[connect Subroutine](#)

## socks5tcp\_accept Subroutine

### Purpose

Awaits an incoming connection to a socket from a previous `socks5tcp_bind()` call.

### Library

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

### Syntax

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

```

int socks5tcp_accept(Socket, Dst, DstLen, Svr, SvrLen)
int Socket;
struct sockaddr *Dst;
size_t DstLen;
struct sockaddr *Svr;
size_t SvrLen;

```

## Description

The **socks5tcp\_accept** subroutine blocks until an incoming connection is established on a listening socket that was requested in a previous call to **socks5tcp\_bind**. Upon success, subsequent writes to and reads from *Socket* will be relayed through *Svr*.

*Socket* must be an open socket descriptor of type SOCK\_STREAM.

The socket applications can be compiled with **COMPAT\_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

## Parameters

Item	Description
<i>Socket</i>	Specifies the unique name of the socket.
<i>Dst</i>	If non-NULL, buffer for receiving the address of the remote client which initiated an incoming connection
<i>DstLength</i>	Specifies the length of the address structure in <i>Dst</i> .
<i>Svr</i>	If non-NULL, specifies the address of the SOCKSv5 server to use to request the relayed connection; on success, this space will be overwritten with the server-side address of the incoming connection.
<i>SvrLength</i>	Specifies the length of the address structure in <i>Svr</i> .

## Return Values

Upon successful completion, the **socks5tcp\_accept** subroutine returns a value of 0, and modifies *Dst* and *Svr* to reflect the actual endpoints of the incoming external socket.

If the **socks5tcp\_accept** subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the generic system error, into the **errno** global variable.
- Moves an error code, indicating the specific SOCKSv5 error, into the **socks5\_errno** global variable.

## Error Codes (placed in errno; inherited from underlying call to connect())

The **socks5tcp\_bindaccept** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
<b>EBADF</b>	The <i>Socket</i> parameter is not valid.
<b>EAFNOSUPPORT</b>	The addresses in the specified address family cannot be used with this socket.
<b>ENETUNREACH</b>	No route to the network or host is present.
<b>EFAULT</b>	The <i>Dst</i> or <i>Svr</i> parameter is not in a writable part of the user address space.
<b>EINVAL</b>	One or more of the specified arguments is invalid.
<b>ENETDOWN</b>	The specified physical network is down.
<b>ENOSPC</b>	There is no space left on a device or system table.

Error	Description
<b>ENOTCONN</b>	The socket could not be connected.

## Error Codes (placed in `socks5_errno`; SOCKSv5-specific errors)

The `socks5tcp_connect` subroutine is unsuccessful if any of the following errors occurs:

Error	Description
<b>S5_ESRVFAIL</b>	General SOCKSv5 server failure.
<b>S5_EPERM</b>	SOCKSv5 server ruleset rejection.
<b>S5_ENETUNREACH</b>	SOCKSv5 server could not reach target network.
<b>S5_EHOSTUNREACH</b>	SOCKSv5 server could not reach target host.
<b>S5_ECONNREFUSED</b>	SOCKSv5 server connection request refused by target host.
<b>S5_ETIMEDOUT</b>	SOCKSv5 server connection failure due to TTL expiry.
<b>S5_EOPNOTSUPP</b>	Command not supported by SOCKSv5 server.
<b>S5_EAFNOSUPPORT</b>	Address family not supported by SOCKSv5 server.
<b>S5_EADDRINUSE</b>	Requested bind address is already in use (at the SOCKSv5 server).
<b>S5_ENOSERV</b>	No server found.

## Examples

The following program fragment illustrates the use of the `socks5tcp_accept` and `socks5tcp_bind` subroutines by a client to request a listening socket from a server and wait for an incoming connection on the server side.

```

struct sockaddr_in svr;
struct sockaddr_in dst;
.
.
.
socks5tcp_bind(s, (struct sockaddr*)&dst, sizeof(dst), (struct sockaddr *)&svr, sizeof(svr), &res,
sizeof(svr));
.
.
.
socks5tcp_accept(s, (struct sockaddr *)&dst, sizeof(dst), (struct sockaddr *)&svr, sizeof(svr));

```

## Related information

[Initiating Stream Connections Example Program](#)

[Sockets Overview](#)

## socks5tcp\_bind Subroutine

### Purpose

Connect to a SOCKSv5 server and request a listening socket for incoming remote connections.

### Library

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

## Syntax

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

```
int socks5tcp_bind(Socket, Dst, DstLen, Svr, SvrLen)
Int Socket;
struct sockaddr *Dst;
size_t DstLen;
struct sockaddr *Svr;
size_t SvrLen;
```

## Description

The **socks5tcp\_bind** subroutine requests a listening socket on the SOCKSv5 server specified in *Svr*, in preparation for an incoming connection from a remote destination, specified by *Dst*. Upon success, *Svr* will be overwritten with the actual address of the newly bound listening socket, and *Socket* may be used in a subsequent call to **socks5tcp\_accept**.

*Socket* must be an open socket descriptor of type SOCK\_STREAM.

The socket applications can be compiled with **COMPAT\_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

## Parameters

Item	Description
<i>Socket</i>	Specifies the unique name of the socket.
<i>Dst</i>	Specifies the address of the SOCKSv5 server to use to request the relayed connection; on success, this space will be overwritten with the actual bound address on the server.
<i>DstLength</i>	Specifies the length of the address structure in <i>Dst</i> .
<i>Svr</i>	If non-NULL, specifies the address of the SOCKSv5 server to use to request the relayed connection; on success, this space will be overwritten with the actual bound address on the server.
<i>SvrLength</i>	Specifies the length of the address structure in <i>Svr</i> .

## Return Values

Upon successful completion, the **socks5tcp\_bind** subroutine returns a value of 0, and modifies *Svr* to reflect the actual address of the newly bound listener socket.

If the **socks5tcp\_bind** subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the generic system error, into the **errno** global variable.
- Moves an error code, indicating the specific SOCKSv5 error, into the **socks5\_errno** global variable.

## Error Codes (placed in errno; inherited from underlying call to connect())

The **socks5tcp\_bindaccept** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
<b>EBADF</b>	The <i>Socket</i> parameter is not valid.
<b>ENOTSOCK</b>	The <i>Socket</i> parameter refers to a file, not a socket.
<b>EADDRNOTAVAIL</b>	The specified address is not available from the local machine.

<b>Error</b>	<b>Description</b>
<b>EAFNOSUPPORT</b>	The addresses in the specified address family cannot be used with this socket.
<b>EISCONN</b>	The socket is already connected.
<b>ETIMEDOUT</b>	The establishment of a connection timed out before a connection was made.
<b>ECONNREFUSED</b>	The attempt to connect was rejected.
<b>ENETUNREACH</b>	No route to the network or host is present.
<b>EADDRINUSE</b>	The specified address is already in use.
<b>EFAULT</b>	The <i>Address</i> parameter is not in a writable part of the user address space.
<b>EINPROGRESS</b>	The socket is marked as nonblocking. The connection cannot be immediately completed. The application program can select the socket for writing during the connection process.
<b>EINVAL</b>	One or more of the specified arguments is invalid.
<b>ENETDOWN</b>	The specified physical network is down.
<b>ENOSPC</b>	There is no space left on a device or system table.
<b>ENOTCONN</b>	The socket could not be connected.

### **Error Codes (placed in `socks5_errno`; SOCKSv5-specific errors)**

The `socks5tcp_connect` subroutine is unsuccessful if any of the following errors occurs:

<b>Error</b>	<b>Description</b>
<b>S5_ESRVFAIL</b>	General SOCKSv5 server failure.
<b>S5_EPERM</b>	SOCKSv5 server ruleset rejection.
<b>S5_ENETUNREACH</b>	SOCKSv5 server could not reach target network.
<b>S5_EHOSTUNREACH</b>	SOCKSv5 server could not reach target host.
<b>S5_ECONNREFUSED</b>	SOCKSv5 server connection request refused by target host.
<b>S5_ETIMEDOUT</b>	SOCKSv5 server connection failure due to TTL expiry.
<b>S5_EOPNOTSUPP</b>	Command not supported by SOCKSv5 server.
<b>S5_EAFNOSUPPORT</b>	Address family not supported by SOCKSv5 server.
<b>S5_EADDRINUSE</b>	Requested bind address is already in use (at the SOCKSv5 server).
<b>S5_ENOERV</b>	No server found.

### **Examples**

The following program fragment illustrates the use of the `socks5tcp_bind` subroutine by a client to request a listening socket from a server.

```
struct sockaddr_in svr;
struct sockaddr_in dst;
.
.
.
socks5tcp_bind(s, (struct sockaddr *)&dst, sizeof(dst), (structsockaddr *)&svr, sizeof(svr));
```

### **Related reference**

[getsockname Subroutine](#)

## Related information

[Sockets Overview](#)

# socks5tcp\_connect Subroutine

## Purpose

Connect to a SOCKSv5 server and request a connection to an external destination.

## Library

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

## Syntax

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

```
int socks5tcp_connect (Socket, Dst, DstLen, Svr, SvrLen)
int Socket;
struct sockaddr *Dst;
size_t DstLen;
struct sockaddr *Svr;
size_t SvrLen;
```

## Description

The **socks5tcp\_connect** subroutine requests a connection to *Dst* from the SOCKSv5 server specified in *Svr*. If successful, *Dst* and *Svr* will be overwritten with the actual addresses of the external connection and subsequent writes to and reads from *Socket* will be relayed through *Svr*.

*Socket* must be an open socket descriptor of type SOCK\_STREAM; *Dst* and *Svr* may be either IPv4 or IPv6 addresses.

The socket applications can be compiled with **COMPAT\_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

## Parameters

Item	Description
<i>Socket</i>	Specifies the unique name of the socket.
<i>Dst</i>	Specifies the external address of the target socket to which the SOCKSv5 server will attempt to connect.
<i>DstLength</i>	Specifies the length of the address structure in <i>Dst</i> .
<i>Svr</i>	If non-NULL, specifies the address of the SOCKSv5 server to use to request the relayed connection.
<i>SvrLength</i>	Specifies the length of the address structure in <i>Svr</i> .

## Return Values

Upon successful completion, the **socks5tcp\_connect** subroutine returns a value of 0, and modifies *Dst* and *Svr* to reflect the actual endpoints of the created external socket.

If the **socks5tcp\_connect** subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the generic system error, into the **errno** global variable.
- Moves an error code, indicating the specific SOCKSv5 error, into the **socks5\_errno** global variable.
- *Dst* and *Svr* are left unmodified.

### Error Codes (placed in **errno**; inherited from underlying call to **connect()**)

The **socks5tcp\_connect** subroutine is unsuccessful if any of the following errors occurs:

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

### Error Codes (placed in **socks5\_errno**; SOCKSv5-specific errors)

The **socks5tcp\_connect** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
<b>S5_ESRVFAIL</b>	General SOCKSv5 server failure.
<b>S5_EPERM</b>	SOCKSv5 server ruleset rejection.
<b>S5_ENETUNREACH</b>	SOCKSv5 server could not reach target network.
<b>S5_EHOSTUNREACH</b>	SOCKSv5 server could not reach target host.
<b>S5_ECONNREFUSED</b>	SOCKSv5 server connection request refused by target host.
<b>S5_ETIMEDOUT</b>	SOCKSv5 server connection failure due to TTL expiry.
<b>S5_EOPNOTSUPP</b>	Command not supported by SOCKSv5 server.
<b>S5_EAFNOSUPPORT</b>	Address family not supported by SOCKSv5 server.
<b>S5_ENOSERV</b>	No server found.

## Examples

The following program fragment illustrates the use of the **socks5tcp\_connect** subroutine by a client to request a connection from a server's socket.

```
struct sockaddr_in svr;
struct sockaddr_in6 dst;
.
.
.
socks5tcp_connect(s, (struct sockaddr*)&dst, sizeof(dst), (struct sockaddr *)&svr, sizeof(svr));
```

### Related reference

[getsockname Subroutine](#)

### Related information

[Initiating Stream Connections Example Program](#)

## socks5udp\_associate Subroutine

### Purpose

Connects to a SOCKSv5 server, and requests a UDP association for subsequent UDP socket communications.

### Library

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

### Syntax

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

```
int socks5udp_associate (Socket, Dst, DstLen, Svr, SvrLen)
int Socket;
const struct sockaddr *Dst;
size_t DstLen;
const struct sockaddr *Svr;
size_t SvrLen;
```

### Description

The **socks5udp\_associate** subroutine requests a UDP association for *Dst* on the SOCKSv5 server specified in *Svr*. Upon success, *Dst* is overwritten with a rendezvous address to which subsequent UDP packets should be sent for relay by *Svr*.

*Socket* must be an open socket descriptor of type SOCK\_STREAM; *Dst* and *Svr* may be either IPv4 or IPv6 addresses.

Note that *Socket* cannot be used to send subsequent UDP packets (a second socket of type SOCK\_DGRAM must be created).

The socket applications can be compiled with **COMPAT\_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

### Parameters

Item	Description
<i>Socket</i>	Specifies the unique name of the socket.

Item	Description
<i>Dst</i>	Specifies the external address of the target socket to which the SOCKSv5 client expects to send UDP packets.
<i>DstLength</i>	Specifies the length of the address structure in <i>Dst</i> .
<i>Svr</i>	Specifies the address of the SOCKSv5 server to use to request the association.
<i>SvrLength</i>	Specifies the length of the address structure in <i>Svr</i> .

## Return Values

Upon successful completion, the **socks5udp\_associate** subroutine returns a value of 0 and overwrites *Dst* with the rendezvous address.

If the **socks5udp\_associate** subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the generic system error, into the **errno** global variable.
- Moves an error code, indicating the specific SOCKSv5 error, into the **socks5\_errno** global variable.

## Error Codes (placed in **errno**; inherited from underlying call to **connect()**)

The **socks5udp\_associate** subroutine is unsuccessful if any of the following errors occurs:

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

## Error Codes (placed in **socks5\_errno**; SOCKSv5-specific errors)

The **socks5tcp\_connect** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
<b>S5_ESRVFAIL</b>	General SOCKSv5 server failure.
<b>S5_EPERM</b>	SOCKSv5 server ruleset rejection.
<b>S5_ENETUNREACH</b>	SOCKSv5 server could not reach target network.
<b>S5_EHOSTUNREACH</b>	SOCKSv5 server could not reach target host.
<b>S5_ECONNREFUSED</b>	SOCKSv5 server connection request refused by target host.
<b>S5_ETIMEDOUT</b>	SOCKSv5 server connection failure due to TTL expiry.
<b>S5_EOPNOTSUPP</b>	Command not supported by SOCKSv5 server.
<b>S5_EAFNOSUPPORT</b>	Address family not supported by SOCKSv5 server.
<b>S5_ENOSERV</b>	No server found.

## Examples

The following program fragment illustrates the use of the **socks5udp\_associate** subroutine by a client to request an association on a server.

```
struct sockaddr_in svr;
struct sockaddr_in6 dst;
.
.
.
socks5udp_associate(s, (struct sockaddr*)&dst, sizeof(dst), (struct sockaddr *)&svr,
sizeof(svr));
```

### Related information

[Initiating Stream Connections Example Program](#)  
[SOCKS5C\\_CONFIG Environment Variable](#)

## socks5udp\_sendto Subroutine

### Purpose

Send UDP packets through a SOCKSv5 server.

### Library

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

### Syntax

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

```
int socks5udp_sendto (Socket, Message, MsgLen, Flags, Dst, DstLen, Svr, SvrLen)
int Socket;
void *Message;
size_t MsgLen;
int Flags;
struct sockaddr *Dst;
size_t DstLen;
struct sockaddr *Svr;
size_t SvrLen;
```

## Description

The **socks5udp\_sendto** subroutine sends a UDP packet to *Svr* for relay to *Dst*. *Svr* must be the rendezvous address returned from a previous call to **socks5udp\_associate**.

*Socket* must be an open socket descriptor of type SOCK\_DGRAM; *Dst* and *Svr* may be either IPv4 or IPv6 addresses.

The socket applications can be compiled with **COMPAT\_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

## Parameters

Item	Description
<i>Socket</i>	Specifies the unique name of the socket.
<i>Message</i>	Specifies the address containing the message to be sent.
<i>MsgLen</i>	Specifies the size of the message in bytes.
<i>Flags</i>	Allows the sender to control the message transmission. See the description in the <b>sendto</b> subroutine for more specific details.
<i>Dst</i>	Specifies the external address to which the SOCKSv5 server will attempt to relay the UDP packet.
<i>DstLength</i>	Specifies the length of the address structure in <i>Dst</i> .
<i>Svr</i>	Specifies the address of the SOCKSv5 server to send the UDP packet for relay.
<i>SvrLength</i>	Specifies the length of the address structure in <i>Svr</i> .

## Return Values

Upon successful completion, the **socks5udp\_sendto** subroutine returns a value of 0.

If the **socks5udp\_sendto** subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the generic system error, into the **errno** global variable.
- Moves an error code, indicating the specific SOCKSv5 error, into the **socks5\_errno** global variable.

## Error Codes (placed in **errno**; inherited from underlying call to **sendto()**)

The **socks5tcp\_connect** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
<b>EBADF</b>	The <i>Socket</i> parameter is not valid.
<b>ENOTSOCK</b>	The <i>Socket</i> parameter refers to a file, not a socket.
<b>EAFNOSUPPORT</b>	The addresses in the specified address family cannot be used with this socket.
<b>ENETUNREACH</b>	No route to the network or host is present.
<b>EINVAL</b>	One or more of the specified arguments is invalid.
<b>ENETDOWN</b>	The specified physical network is down.
<b>ENOSPC</b>	There is no space left on a device or system table.

## Error Codes (placed in `socks5_errno`; SOCKSv5-specific errors)

The `socks5tcp_connect` subroutine is unsuccessful if any of the following errors occurs:

Error	Description
<b>S5_ESRVFAIL</b>	General SOCKSv5 server failure.
<b>S5_EPERM</b>	SOCKSv5 server ruleset rejection.
<b>S5_ENETUNREACH</b>	SOCKSv5 server could not reach target network.
<b>S5_EHOSTUNREACH</b>	SOCKSv5 server could not reach target host.
<b>S5_ECONNREFUSED</b>	SOCKSv5 server connection request refused by target host.
<b>S5_ETIMEDOUT</b>	SOCKSv5 server connection failure due to TTL expiry.
<b>S5_EOPNOTSUPP</b>	Command not supported by SOCKSv5 server.
<b>S5_EAFNOSUPPORT</b>	Address family not supported by SOCKSv5 server.
<b>S5_ENOSERV</b>	No server found.

## Examples

The following program fragment illustrates the use of the `socks5udp_sendto` subroutine by a client to request a connection from a server's socket.

```
void *message;
size_t msglen;
int flags;
struct sockaddr_in svr;
struct sockaddr_in6 dst;
.
.
socks5udp_associate(s, (struct sockaddr*)&dst, sizeof(dst), (struct sockaddr *)&svr, sizeof(svr));
.
.
socks5udp_sendto(s, message, msglen, flags (struct sockaddr*)&dst, sizeof(dst), (struct sockaddr *)&svr, sizeof(svr));
```

## Related information

[Sockets Overview](#)

## splice Subroutine

### Purpose

Lets the protocol stack manage two sockets that use TCP.

### Syntax

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

```
int splice(socket1, socket2, flags)
    int socket1, socket2;
    int flags;
```

### Description

The **splice** subroutine will let TCP manage two sockets that are in connected state thus relieving the caller from moving data from one socket to another. After the **splice** subroutine returns successfully, the caller needs to close the two sockets.

The two sockets should be of type **SOCK\_STREAM** and protocol **IPPROTO\_TCP**. Specifying a protocol of zero will also work.

## Parameters

Item	Description
<i>socket1, socket2</i>	Specifies a socket that had gone through a successful connect() or accept().
<i>flags</i>	Set to zero. Currently ignored.

## Return Values

Item	Description
0	Indicates a successful completion.
-1	Indicates an error. The specific error is indicated by errno.

## Error Codes

Item	Description
<b>EBADF</b>	<i>socket1</i> or <i>socket2</i> is not valid.
<b>ENOTSOCK</b>	<i>socket1</i> or <i>socket2</i> refers to a file, not a socket.
<b>EOPNOTSUPP</b>	<i>socket1</i> or <i>socket2</i> is not of type <b>SOCK_STREAM</b> .
<b>EINVAL</b>	The parameters are invalid.
<b>EEXIST</b>	<i>socket1</i> or <i>socket2</i> is already spliced.
<b>ENOTCONN</b>	<i>socket1</i> or <i>socket2</i> is not in connected state.
<b>EAFNOSUPPORT</b>	<i>socket1</i> or <i>socket2</i> address family is not supported for this subroutine.

## WriteFile Subroutine

### Purpose

Writes data to a socket.

### Syntax

```
#include <iocp.h>
boolean_t WriteFile (FileDescriptor, Buffer, WriteCount, AmountWritten, Overlapped)
HANDLE FileDescriptor;
LPVOID Buffer;
DWORD WriteCount;
LPDWORD AmountWritten;
LPOVERLAPPED Overlapped;
```

### Description

The **WriteFile** subroutine writes the number of bytes specified by the *WriteCount* parameter from the buffer indicated by the *Buffer* parameter to the *FileDescriptor* parameter. The number of bytes written is saved in the *AmountWritten* parameter. The *Overlapped* parameter indicates whether or not the operation can be handled asynchronously.

The **WriteFile** subroutine returns a boolean (an integer) indicating whether or not the request has been completed.

The **WriteFile** subroutine is part of the I/O Completion Port (IOCP) kernel extension.

**Note:** This subroutine only works to a socket file descriptor. It does not work with files or other file descriptors.

## Parameters

Item	Description
<i>FileDescriptor</i>	Specifies a valid file descriptor obtained from a call to the <b>socket</b> or <b>accept</b> subroutines.
<i>Buffer</i>	Specifies the buffer from which the data will be written.
<i>WriteCount</i>	Specifies the maximum number of bytes to write.
<i>AmountWritten</i>	Specifies the number of bytes written. The parameter is set by the subroutine.
<i>Overlapped</i>	Specifies an overlapped structure indicating whether or not the request can be handled asynchronously.

## Return Values

Upon successful completion, the **WriteFile** subroutine returns a boolean indicating the request has been completed.

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

- Returns a value of 0 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable, see the link in the Related Information section of this document.

## Error Codes

Item	Description
<b>EINPROGRESS</b>	The write request can not be immediately satisfied and will be handled asynchronously. A completion packet will be sent to the associated completion port upon completion.
<b>EAGAIN</b>	The write request cannot be immediately satisfied and cannot be handled asynchronously.
<b>EINVAL</b>	The <i>FileDescriptor</i> is invalid.

## Examples

The following program fragment illustrates the use of the **WriteFile** subroutine to synchronously write data to a socket:

```
void buffer;  
int amount_written;  
b=WriteFile (34, &buffer, 128, &amount_written, NULL);
```

The following program fragment illustrates the use of the **WriteFile** subroutine to asynchronously write data to a socket:

```
void buffer;  
int amount_written;  
LPOVERLAPPED overlapped;  
b = ReadFile (34, &buffer, 128, &amount_written, overlapped);
```

**Note:** The request will only be handled asynchronously if it cannot be immediately satisfied.

## Related information

[Error Notification Object Class](#)

# Packet Capture

---

The packet capture library contains subroutines that allow users to communicate with the packet capture facility provided by the operating system to read unprocessed network traffic. Applications using these subroutines must be run as root. These subroutines are maintained in the **libpcap.a** library:

## Related information

[pcap\\_close](#)

[pcap\\_strerror](#)

## ioctl BPF Control Operations

### Purpose

Performs packet-capture-related control operations.

### Syntax

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

```
int ioctl ( int fd, int cmd [, arg ] )
```

### Description

The Berkeley Packet Filter (BPF) ioctl commands perform a variety of packet-capture-related control. The *fd* argument is a BPF device descriptor. For non-packet-capture descriptors, functions performed by this call are unspecified.

The *cmd* parameter and an optional third parameter (with varying types) are passed to and interpreted by the BPF ioctl function to perform an appropriate control operation that is specified by the user.

### Parameters

Item	Description
<i>fd</i>	Specifies an open file descriptor that refers to a BPF device created using the open call.
<i>cmd</i>	Selects the control function to be performed.
<i>arg</i>	Represents additional information that is needed to perform the requested function. The type of the <i>arg</i> parameter is either an integer or a pointer to a BPF-specific data structure, depending on the particular control request.

### BPF Control Operations

In addition to the **FIONREAD** ioctl command, the following commands can be applied to any open BPF device. The *arg* parameter is a pointer to the indicated type.

ioctl command	Type of the <i>arg</i> parameter	Description
<b>BIOCGBLEN</b>	<b>u_int</b>	Returns the buffer length for reads on BPF devices.

<b>ioctl command</b>	<b>Type of the <i>arg</i> parameter</b>	<b>Description</b>
<b>BIOCSBLEN</b>	<b>u_int</b>	Sets the buffer length for reads on BPF devices. The <i>buffer</i> parameter must be set before the device is attached to an interface with the <b>BIOCSETIF</b> command. If the requested buffer size cannot be accommodated, the closest allowable size is set and returned in the <i>arg</i> parameter.
<b>BIOCGLDT</b>	<b>u_int</b>	Returns the type of the data link layer underlying the attached interface.
<b>BIOCPRMISC</b>	N/A	Forces the interface into promiscuous mode. All packets, not just those destined for the local host, are processed. A listener that opened its interface nonpromiscuously can receive packets promiscuously, because more than one device can be listening on a given interface. The problem can be remedied with an appropriate filter.
<b>BIOCFLUSH</b>	N/A	Flushes the buffer of incoming packets, and resets the statistics that are returned by the <b>BIOCGSTATS</b> command.
<b>BIOCGETIF</b>	<b>struct ifreq</b>	Returns the name of the hardware interface that the device is listening on. The name is returned in the <b>ifr_name</b> field of the <b>ifreq</b> structure. All other fields are undefined.
<b>BIOCSETIF</b>	<b>struct ifreq</b>	Sets the hardware interface associate with the device. This command must be performed before any pack-packets can be read. The device is indicated by the name using the <b>ifr_name</b> field of the <b>ifreq</b> structure. Additionally, the command performs the actions of the <b>BIOCFLUSH</b> command.
<b>BIOCGRTIMEOUT</b>	<b>struct timeval</b>	Gets the read timeout value. The <i>arg</i> parameter specifies the length of time to wait before a read request times out. This parameter is initialized to zero by an open, indicating no timeout.
<b>BIOCSRTIMEOUT</b>	<b>struct timeval</b>	Sets the read timeout value described in the <b>BIOCGRTIMEOUT</b> command.
<b>BIOCGSTATS</b>	<b>struct bpf_stat</b>	Returns the a structure of packet statistics. The structure is defined in the <b>net/bpf.h</b> file.
<b>BIOCIMMEDIATE</b>	<b>u_int</b>	Enables or disables the <b>immediate</b> mode, based on the truth value of the <i>arg</i> parameter. When the <b>immediate</b> mode is enabled, reads return immediately upon packet reception. Otherwise, a read will be blocked until either the kernel buffer becomes full or a timeout occurs.
<b>BIOCSETF</b>	<b>struct bpf_program</b>	Sets the filter program used by the kernel to discard uninteresting packets. The <b>bpf_program</b> structure is defined in the <b>net/bpf.h</b> file.

ioctl command	Type of the <i>arg</i> parameter	Description
<b>BIOCVERSION</b>	<b>struct bpf_version</b>	Returns the major and minor version numbers of the filter language currently recognized by the kernel. Before installing a filter, applications must check that the current version is compatible with the running kernel. The current version numbers are given by the <code>BPF_MAJOR_VERSION</code> and <code>BPF_MINOR_VERSION</code> variables from the <code>net/bpf.h</code> file. An incompatible filter might result in undefined behavior.

## Return Values

Upon successful completion, `ioctl` returns a value of 0. Otherwise, it returns a value of -1 and sets `errno` to indicate the error.

## Error Codes

The `ioctl` commands fail under the following general conditions:

Item	Description
<b>EINVAL</b>	A command or argument, which is not valid, was specified.
<b>ENETDOWN</b>	The underlying interface or network is down.
<b>ENXIO</b>	The underlying interface is not found.
<b>ENOBUFS</b>	Insufficient memory was available to process the request.
<b>EEXIST</b>	The BPF device already exists.
<b>ENODEV</b>	The BPF device could not be set up.
<b>EINTR</b>	A signal was caught during an <code>ioctl</code> operation.
<b>EACCES</b>	The permission was denied for the specified operation.
<b>EADDRNOTAVAIL</b>	The specified address is not available for interface.
<b>ENOMEM</b>	The available memory is not enough.
<b>ESRCH</b>	Such a process does not exist.

### Related information

[Packet Capture Library Overview](#)

## Librdmacm Library

The `librdmacm` library provides the connection management (CM) functionality and the CM interfaces for remote direct memory access (RDMA).

The API user space is described in the `/usr/include/rdma/rdma_cma.h` file.

The manual pages are created to describe the various interfaces and test programs that are available. For a full list of interfaces and test programs, refer to the `rdma_cm` manual page.

## Returned error rules

The `librdmacm` functions return 0 to indicate success, and a negative value to indicate failure.

If a function operates asynchronously, a return value of 0 means that the operation was successfully started. The operation might still return an error. You must check the status of the related event. If the return value is -1, the `errno` can be examined for additional information of the failure.

Item	Description
=0	Success
= -1	Error . See the <b>errno</b> for details of the error message.

## Supported verbs

You can find a list of verbs supported by the librdmacm library.

### Event channel operations

Lists the event channel operations that are handled for the library verbs.

#### ***rdma\_create\_event\_channel***

Opens a channel that is used to report communication events.

### Syntax

```
#include <rdma/rdma_cma.h>
struct rdma_event_channel *rdma_create_event_channel(void);
```

### Description

The **rdma\_create\_event\_channel** function reports the asynchronous events through event channels. Each event channel maps to a file descriptor.

#### Note:

- Event channels are used to direct all events on an **rdma\_cm\_id** identifier. You might require multiple event channels when you are managing a large number of connections or connection manager (CM) ID's.
- All event channels that are created must be destroyed by calling the **rdma\_destroy\_event\_channel** function. You must call the **rdma\_get\_cm\_event** function to retrieve events on an event channel.

### Parameters

Item	Description
<i>void</i>	No arguments.

### Return Value

The **rdma\_create\_event\_channel** function returns 0 on success, and NULL if the request fails. On failure, **errno** indicates the reason for failure.

#### ***rdma\_destroy\_event\_channel***

Closes an event communication channel.

### Syntax

```
#include <rdma/rdma_cma.h>
void rdma_destroy_event_channel(struct rdma_event_channel *channel);
```

### Description

The **rdma\_destroy\_event\_channel** function releases all resources that are associated with an event channel and closes the associated file descriptor.

**Note:** The `rdma_cm_id` identifiers that are associated with the event channel must be destroyed, and all returned events must be acknowledged before calling the `rdma_destroy_event_channel` function.

## Parameters

Item	Description
<code>channel</code>	Specifies the communication channel to be destroyed.

## Return Value

The `rdma_destroy_event_channel` function returns 0 on success, or -1 on error. If an error occurs, `errno` indicates the reason for failure.

## Connection Manager (CM) ID operations

The Connection Manager (CM) ID operation is used for ID related operations such as to create, destroy, migrate, resolve address, establish connection, listen to the request, reject request, and to provide the address information.

### *rdma\_cm*

Establishes communication over RDMA transports.

## Syntax

```
#include <rdma/rdma_cma.h>
```

## Description

Establishes communication over RDMA transports.

### Notes:

- The RDMA CM is a communication manager (CM) used to set up reliable, connected, and unreliable datagram data transfers. It provides an RDMA transport neutral interface for establishing connections. The API concepts are based on sockets, but adapted for queue pair (QP) based semantics. The communication for QP must be over a specific RDMA device, and data transfers are message-based.
- The RDMA CM can control both the QP and communication management (that is connection setup or teardown) functions of an RDMA API, or only the communication management. It works in conjunction with the verbs API that is defined by the libibverbs library. The libibverbs library provides the underlying interfaces needed to send and receive data.
- The RDMA CM can operate asynchronously or synchronously. The mode of operation is controlled by using the `rdma_cm` event channel parameter in specific calls. If an event channel is provided, an `rdma_cm` identifier reports its event data (that is results of establishing a connection, for example), on that channel. If a channel is not provided, then all `rdma_cm` operation for the selected `rdma_cm` identifier is blocked until the channel completes.

## RDMA verbs

The `rdma_cm` manager supports the verbs that are available in the libibverbs library and interfaces. However, it also provides wrapper functions for the commonly used verbs. The set of abstracted verb call are:

### `rdma_reg_msgs`

Registers an array of buffers for sending and receiving.

### `rdma_reg_read`

Registers a buffer for RDMA read operations.

**rdma\_reg\_write**

Registers a buffer for RDMA write operations.

**rdma\_dereg\_m**

Reregisters a memory region.

**rdma\_post\_rcv**

Posts a buffer to receive a message.

**rdma\_post\_send**

Posts a buffer to send a message.

**rdma\_post\_read**

Posts an RDMA to read data into a buffer.

**rdma\_post\_write**

Posts an RDMA to send data from a buffer.

**rdma\_post\_rcvv**

Posts a vector of buffers to receive a message.

**rdma\_post\_sendv**

Posts a vector of buffers to send a message.

**rdma\_post\_readv**

Posts a vector of buffers to receive an RDMA read.

**rdma\_post\_writev**

Posts a vector of buffers to send an RDMA write.

**rdma\_post\_ud\_send**

Posts a buffer to send a message on a UD QP.

**rdma\_get\_send\_comp**

Gets completion status for a send or RDMA operation.

**rdma\_get\_rcv\_comp**

Gets information about a completed receive.

**Examples****1. CLIENT operation**

An overview of the basic operation for the active, or client, side of communication is described in this section. This flow is for asynchronous operation with low-level call details. For synchronous operation, calls to **rdma\_create\_event\_channel**, **rdma\_get\_cm\_event**, **rdma\_ack\_cm\_event**, and **rdma\_destroy\_event\_channel** is eliminated. Abstracted calls, such as **rdma\_create\_ep** contains several calls under a single API. A general connection flow includes the following calls:

**rdma\_getaddrinfo**

Retrieves address information of the destination.

**rdma\_create\_event\_channel**

Creates channel to receive events.

**rdma\_create\_id**

Allocates an **rdma\_cm\_id** identifier, this call is similar in function to a socket.

**rdma\_resolve\_addr**

Obtains a local RDMA device to reach the remote address.

**rdma\_get\_cm\_event**

Waits for RDMA\_CM\_EVENT\_ADDR\_RESOLVED event.

**rdma\_ack\_cm\_event**

Acknowledges an event.

**rdma\_create\_qp**

Allocates a queue pair (QP) for the communication.

**rdma\_resolve\_route**

Determines the route to the remote address.

**rdma\_get\_cm\_event**

Waits for the RDMA\_CM\_EVENT\_ROUTE\_RESOLVED event.

**rdma\_ack\_cm\_event**

Acknowledges an event.

**rdma\_connect**

Connects to the remote server.

**rdma\_get\_cm\_event**

Waits for the RDMA\_CM\_EVENT\_ESTABLISHED event

**rdma\_ack\_cm\_event**

Acknowledges an event.

To perform data transfers over connection, follow these steps:

**rdma\_disconnect**

Tears-down a connection.

**rdma\_get\_cm\_event**

Waits for an RDMA\_CM\_EVENT\_DISCONNECTED event.

**rdma\_ack\_cm\_event**

Acknowledges an event.

**rdma\_destroy\_qp**

Destroys the QP.

**rdma\_destroy\_id**

Releases the **rdma\_cm\_id** identifier.

**rdma\_destroy\_event\_channel**

Releases the event channel.

An identical process is used to set up unreliable datagram (UD) communication between nodes. No actual connection is formed between the queue pairs, so disconnection is not required. This example shows initiating the client for disconnect, either side of a connection can initiate the disconnect.

## 2. Server connection

A general overview of the basic operation for the passive, or server, side of communication is explained. A general connection flow includes the following events:

**rdma\_create\_event\_channel**

Creates channel to receive events.

**rdma\_create\_id**

Allocates an **rdma\_cm\_id** identifier, this call is similar in function to a socket.

**rdma\_bind\_addr**

Sets the local port number to listen.

**rdma\_listen**

Begins to listen for connection requests.

**rdma\_get\_cm\_event**

Waits for RDMA\_CM\_EVENT\_CONNECT\_REQUEST event with a new **rdma\_cm\_id** identifier.

**rdma\_create\_qp**

Allocates a QP for the communication on the new **rdma\_cm\_id** identifier.

**rdma\_accept**

Accepts the connection request.

**rdma\_ack\_cm\_event**

Acknowledges an event.

**rdma\_get\_cm\_event**

Waits for RDMA\_CM\_EVENT\_ESTABLISHED event.

**rdma\_ack\_cm\_event**

Acknowledges an event.

To perform data transfers over connection, follow these steps:

**rdma\_get\_cm\_event**

Waits for an RDMA\_CM\_EVENT\_DISCONNECTED event.

**rdma\_ack\_cm\_event**

Acknowledges an event.

**rdma\_disconnect**

Tears-down a connection.

**rdma\_destroy\_qp**

Destroys the QP.

**rdma\_destroy\_id**

Releases the connected **rdma\_cm\_id** identifier.

**rdma\_destroy\_id**

Releases the listening **rdma\_cm\_id** identifier.

**rdma\_destroy\_event\_channel**

Releases the event channel.

**Exit Status**

**= 0**

Success

**= -1**

Error. See **errno** for more details.

Most librdmacm functions return 0 to indicate success, and a -1 return value to indicate failure. If a function operates asynchronously, a return value of 0 means that the operation started successfully. The operation can complete in error, and you must check the status of the related event. If the return value is -1, then **errno** contains additional information for the failure.

**Note:** The earlier versions of the library would return **-errno** and is not set to **errno** for some cases related to **ENOMEM**, **ENODEV**, **ENODATA**, **EINVAL**, and **EADDRNOTAVAIL** codes. Applications that require to verify the earlier version of the codes and that are compatible must manually set **errno** to negative of the return code, if it is **< -1**.

**rdma\_create\_id**

Allocates a communication identifier.

**Syntax**

```
#include <rdma/rdma_cma.h>
int rdma_create_id(struct rdma_event_channel *channel, struct rdma_cm_id **id, void *context,
enum rdma_port_space ps);
```

**Description**

The **rdma\_create\_id** function creates an identifier that is used to track communication information. The communication channel that the events are associated with the allocated **rdma\_cm\_id** identifier is communicated. This may be NULL.

**Notes:**

- The **rdma\_cm\_id** identifiers are equivalent to that of a socket in RDMA communication. The difference is that the RDMA communication requires explicit binding to a specified Remote Direct Memory Access (RDMA) device before communicating, and most operations are asynchronous in nature. The asynchronous communication events on an **rdma\_cm\_id** identifier are reported through the associated

event channel. If the channel parameter is NULL, the **rdma\_cm\_id** is placed into synchronous operation. While operating synchronously, calls that result in an event cause a block until the operation completes. The event is returned to the user through the **rdma\_cm\_id** structure, and is available for access until the next **rdma\_cm** call is made.

- You must release the `rdma_cm_id` identifier by calling the **rdma\_destroy\_id** function.

**Port Spaces:** RDMA\_PS\_TCP provides reliable, connection-oriented queue pair (QP). Unlike TCP, the RDMA port space provides stream-based communication.

## Parameters

Item	Description
<i>channel</i>	Specifies the communication channel for the allocated <code>rdma_cm_id</code> identifier to report the associated events.
<i>context</i>	Indicates the user-specified context that is associated with the communication identifier.
<i>id</i>	Specifies a reference identifier to return the allocated communication identifier.
<i>ps</i>	Specifies the RDMA port space.

## Return Values

The **rdma\_destroy\_event\_channel** function returns 0 on success, or -1 on error. If an error occurs, the **errno** indicates the reason for failure.

### **rdma\_destroy\_id**

Releases a communication identifier.

## Syntax

```
#include <rdma/rdma_cma.h>
int rdma_destroy_id(struct rdma_cm_id *id);
```

## Description

The **rdma\_destroy\_id** function destroys the specified `rdma_cm_id` identifier and cancels any outstanding asynchronous operation.

**Note:** You must release any queue pair (QP) that is associated with the `rdma_cm_id` identifier before you call the **rdma\_destroy\_id** function and acknowledge all the related events.

## Parameters

Item	Description
<i>id</i>	Specifies the communication identifier to destroy.

## Return Values

The **rdma\_destroy\_event\_channel** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

## ***rdma\_migrate\_id***

Moves a communication identifier to another event channel.

### **Syntax**

```
#include <rdma/rdma_cma.h>
int rdma_migrate_id(struct rdma_cm_id *id, struct rdma_event_channel *channel);
```

### **Description**

The **rdma\_migrate\_id** function migrates a communication identifier to a different event channel and moves the pending events associated with the `rdma_cm_id` identifier to the new channel.

#### **Notes:**

- You must not poll for current event channel on the `rdma_cm_id` identifiers or run any other routines on the `rdma_cm_id` identifier when migrating between channels.
- The **rdma\_migrate\_id** operation stops if any unacknowledged events are on the current event channel.
- If the channel parameter is NULL, the specified `rdma_cm_id` identifier is placed into synchronous operation mode. All calls on the ID is blocked until the operation completes.

### **Parameters**

<b>Item</b>	<b>Description</b>
<i>id</i>	Specifies the existing communication identifier to migrate.
<i>channel</i>	Specifies the communication channel that the events associated with the allocated <code>rdma_cm_id</code> identifier reports. This parameter may be NULL.

### **Return Values**

The **rdma\_migrate\_id** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

## ***rdma\_bind\_addr***

Binds an remote direct memory access (RDMA) identifier to a source address.

### **Syntax**

```
#include <rdma/rdma_cma.h>
int rdma_bind_addr(struct rdma_cm_id *id, struct sockaddr *addr);
```

### **Description**

The **rdma\_bind\_addr** function associates a source address with an `rdma_cm_id` identifier. The address can be a wildcard value. If an `rdma_cm_id` identifier has a local address, the identifier also has a local RDMA device.

#### **Notes:**

- The **rdma\_bind\_addr** operation is run before the **rdma\_listen** operation to bind to a specific port number. The **rdma\_bind\_addr** operation is run on the active side of a connection before the **rdma\_resolve\_addr** routine runs to bind to a specific address.
- If the **rdma\_bind\_addr** operation is used to bind to port 0, the `rdma_cm` function selects an available port that can be retrieved with the **rdma\_get\_src\_port** operation.

## Parameters

Item	Description
<i>id</i>	Specifies the RDMA identifier.
<i>addr</i>	Specifies the local address information. Wildcard values are permitted.

## Return Values

The **rdma\_bind\_addr** function returns the following values:

Item	Description
0	On success.
-1	Error, see <b>errno</b> .

## **rdma\_resolve\_addr**

Resolves the destination and optional source addresses.

## Syntax

```
#include <rdma/rdma_cma.h>
int rdma_resolve_addr(struct rdma_cm_id *id, struct sockaddr *src_addr, struct sockaddr
*dst_addr,
int timeout_ms);
```

## Description

The **rdma\_resolve\_addr** function resolves the destination and optional source addresses from an IP address to an Remote Direct Memory Access (RDMA) address. If successful, the specified `rdma_cm_id` identifier is associated with a local device.

### Notes:

- The **rdma\_resolve\_addr** operation is used to map a specified destination IP address to a usable RDMA address. The IP- RDMA address mapping is done by using the local routing table, or by using ARP.
- If the source address is specified, the `rdma_cm_id` identifier is associated with the source address, and the situation is similar to running the **rdma\_bind\_addr** operation. If no source address is specified, the `rdma_cm_id` identifier is not associated with a device, and the identifier gets associated with a source address based on the local routing tables.
- The **rdma\_resolve\_addr** operation is run from the active side of a connection, before running the **rdma\_resolve\_route** and **rdma\_connect** operations.

## Parameters

Item	Description
<i>id</i>	Specifies the RDMA identifier.
<i>src_addr</i>	Specifies the source address information, and this parameter can be NULL.
<i>dst_addr</i>	Specifies the destination address information.
<i>timeout_ms</i>	Specifies the time of resolution.

## Return Values

The **rdma\_resolve\_addr** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

## ***rdma\_resolve\_route***

Resolves the route information that is required to establish a connection.

### **Syntax**

```
#include <rdma/rdma_cma.h>
int rdma_resolve_route(struct rdma_cm_id *id, int timeout_ms);
```

### **Description**

The **rdma\_resolve\_route** function resolves an RDMA route to the destination address to establish a connection. The destination address must be resolved by running the **rdma\_resolve\_addr** subroutine.

**Note:** The **rdma\_resolve\_route** operation is called on the client side of a connection after running the **rdma\_resolve\_addr** operation, but before the **rdma\_connect** operation.

### **Parameters**

<b>Item</b>	<b>Description</b>
<i>id</i>	Specifies the RDMA identifier.
<i>timeout_ms</i>	Specifies the time of resolution.

### **Return Values**

The **rdma\_resolve\_route** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

## ***rdma\_connect***

Initiates an active connection request.

### **Syntax**

```
#include <rdma/rdma_cma.h>
int rdma_connect(struct rdma_cm_id *id, struct rdma_conn_param *conn_param);
```

### **Description**

The **rdma\_connect** function initiates a connection request to a remote destination.

**Note:** The route to the destination address must be resolved by running the **rdma\_resolve\_route** call or by running the **rdma\_create\_ep** call before the **rdma\_connect** operation.

### **Connection Properties**

The following properties are used to configure the communication that is specified by the *conn\_param* parameter when connecting or establishing a datagram communication.

#### **private\_data**

References a user-controlled data buffer. The contents of the buffer are copied and transparently passed to the remote side as part of the communication request. This property can be NULL if it is not required.

#### **private\_data\_len:**

Specifies the size of the user-controlled data buffer.

#### **responder\_resources:**

Specifies the maximum number of outstanding Remote Direct Memory Access (RDMA) read operations that the local side accepts from the remote side. This property applies only to the

RDMA\_PS\_TCP event. The **responder\_resources** value must be less than or equal to the local RDMA device attribute **max\_qp\_rd\_atom** and to the remote RDMA device attribute **max\_qp\_init\_rd\_atom**. The remote endpoint can adjust this value when accepting the connection.

**initiator\_depth:**

Specifies the maximum number of outstanding RDMA read operations that the local side must read to the remote side. This property applies only to the RDMA\_PS\_TCP event. The **initiator\_depth** value must be less than or equal to the local RDMA device attribute **max\_qp\_init\_rd\_atom** and to the remote RDMA device attribute **max\_qp\_rd\_atom**. The remote endpoint can adjust to this value when accepting the connection.

**flow\_control:**

Specifies if the hardware flow control is available. The **flow\_control** value is exchanged with the remote peer and is not used to configure the queue pair (QP). This property applies only to the RDMA\_PS\_TCP event, and is specific to the InfiniBand architecture.

**retry\_count:**

Specifies the maximum number of times the data transfer operation must be tried on the connection when an error occurs. The **retry\_count** setting controls the number of times to retry sending the data transmission to RDMA, and atomic operations when time outs occur. This property applies only to the RDMA\_PS\_TCP event, and is specific to the InfiniBand architecture.

**nr\_retry\_count:**

Specifies the maximum number of times that a send operation from the remote peer is tried on a connection after receiving a **receiver not ready** (RNR) error. RNR errors are generated when a send request arrives before a buffer is posted to receive the incoming data. This property applies only to the RDMA\_PS\_TCP event., and is specific to the InfiniBand architecture.

**srq:**

Specifies whether the QP that is associated with the connection is using a shared receive queue. The **srq** field is ignored by the library if a QP is created on the `rdma_cm_id` identifier. This property applies only to the RDMA\_PS\_TCP event, and is currently not supported.

**qp\_num:**

Specifies whether the QP number is associated with the connection. The **qp\_num** field is ignored by the library if a QP is created on the `rdma_cm_id` identifier. This property applies only to the RDMA\_PS\_TCP event.

**iWARP specific:**

Specifies the connections established over Internet Wide Area RDMA Protocol ( iWARP RDMA) devices that currently require the active side of the connection to send the first message.

## Parameters

Item	Description
<i>id</i>	Specifies the RDMA identifier.
<i>conn_param</i>	Specifies the connection parameters.

## Return Values

The **rdma\_connect** function returns the following values:

Item	Description
0	On success.
-1	Error, see <b>errno</b> .

## ***rdma\_listen***

Listens to the incoming connection requests.

### **Syntax**

```
#include <rdma/rdma_cma.h>
int rdma_listen(struct rdma_cm_id *id, int backlog);
```

### **Description**

The **rdma\_listen** function initiates a listen operation for the incoming connection requests. The listen operation is restricted to the locally bound source addresses.

#### **Notes:**

- You must have bound and associated the `rdma_cm_id` identifier with a local address by the **rdma\_bind\_addr** operation before the **rdma\_listen** operation.
- If the `rdma_cm_id` identifier is bound to a specific IP address, the listen operation is restricted to that address and the associated RDMA device.
- If the `rdma_cm_id` identifier is bound to an RDMA port number, the listen operation occurs across all RDMA devices.

### **Parameters**

<b>Item</b>	<b>Description</b>
<i>id</i>	Specifies the RDMA identifier.
<i>backlog</i>	Specifies the backlog of incoming connection requests.

### **Return Values**

The **rdma\_listen** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

## ***rdma\_accept***

Accepts a connection request.

### **Syntax**

```
#include <rdma/rdma_cma.h>
int rdma_accept(struct rdma_cm_id *id, struct rdma_conn_param *conn_param);
```

### **Description**

The **rdma\_accept** function is used to accept a connection lookup request.

#### **Notes:**

- The **rdma\_accept** operation is not called on a listening `rdma_cm_id` identifier. After the **rdma\_listen** operation is run, you must wait for a connection request event to occur.
- The `rdma_cm_id` identifier is created by the connection request events similar to a new socket, but the `rdma_cm_id` identifier is associated to a specific RDMA device. The **rdma\_accept** operation is called on the new `rdma_cm_id` identifier.

### **Connection Properties**

Refer to the **rdma\_connect** routine for details on establishing a connection with the identifier.

## Parameters

Item	Description
<i>id</i>	Specifies the connection identifier that is associated with the request.
<i>conn_param</i>	Specifies the information required to establish the connection.

## Return Values

The **rdma\_accept** function returns the following values:

Item	Description
0	On success.
-1	Error, see <b>errno</b> .

## InfiniBand specific

The InfiniBand QPs are configured with minimum RNR NAK timer and local ACK timeout values. The minimum RNR NAK timer value is set to 0, for a delay of 655 ms. The local ACK timeout is calculated based on the packet lifetime and local HCA ACK delay. The packet lifetime is determined by the InfiniBand Subnet Administrator and is part of the route (path record) information that is obtained from the active side of the connection. The HCA ACK delay is a property of the locally used HCA.

The RNR retry count is a 3-bit value.

### ***rdma\_reject***

Rejects a connection request.

## Syntax

```
#include <rdma/rdma_cma.h>
int rdma_reject(struct rdma_cm_id *id, const void *private_data, uint8_t private_data_len);
```

## Description

The **rdma\_reject** function is run from the listening side of the connection to reject a connection lookup request.

**Note:** You can run the **rdma\_reject** operation after receiving a connection request event. If the underlying RDMA transport function supports private data in the rejection message, the specified data is passed to the remote side.

## Parameters

Item	Description
<i>id</i>	Specifies the connection identifier that is associated with the request.
<i>private_data</i>	Specifies the optional private data to send with the rejection message.
<i>private_data_len</i>	Specifies the size of the <i>private_data</i> parameter that can be sent, in bytes.

## Return Values

The **rdma\_reject** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

## ***rdma\_disconnect***

Disconnects a connection.

### **Syntax**

```
#include <rdma/rdma_cma.h>
int rdma_disconnect(struct rdma_cm_id *id);
```

### **Description**

The **rdma\_disconnect** function disconnects a connection and transitions any associated queue pair (QP) with the error state. The error state moves the work requests that are posted to the completion queue. This routing can be run by the client and server side of a connection. An RDMA\_CM\_EVENT\_DISCONNECTED event is generated on both sides of the connection after successful disconnection.

### **Parameters**

<b>Item</b>	<b>Description</b>
<i>id</i>	Specifies the connection identifier that is associated with the request.

### **Return Values**

The **rdma\_destroy\_event\_channel** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

## ***rdma\_get\_src\_port***

Returns the local port number of the associated communication identifier.

### **Syntax**

```
#include <rdma/rdma_cma.h>
uint16_t rdma_get_src_port(struct rdma_cm_id *id)
```

### **Description**

The **rdma\_get\_src\_port** function returns the local port number for an `rdma_cm_id` identifier that is associated with a local address.

### **Parameters**

<b>Item</b>	<b>Description</b>
<i>id</i>	Specifies the connection identifier that is associated with the request.

### **Return Values**

The **rdma\_get\_src\_port** function returns the 16-bit port identifier associated with the local endpoint. If the **rdma\_cm\_id** is not bound to a port, the returned value is 0.

## ***rdma\_get\_dst\_port***

Returns the remote port number of the associated identifier.

### **Syntax**

```
#include <rdma/rdma_cma.h>
uint16_t rdma_get_dst_port(struct rdma_cm_id *id)
```

### **Description**

The **rdma\_get\_dst\_port** function returns the remote port number for an `rdma_cm_id` identifier that is associated with a remote address.

### **Parameters**

<b>Item</b>	<b>Description</b>
<i>id</i>	Specifies the connection identifier that is associated with the request.

### **Return Values**

The **rdma\_get\_dst\_port** function returns the 16-bit port identifier associated with the peer endpoint. If the `rdma_cm_id` is not connected, the returned value is 0.

## ***rdma\_get\_local\_addr***

Returns the local IP address of the associated identifier.

### **Syntax**

```
#include <rdma/rdma_cma.h>
struct sockaddr *rdma_get_local_addr(struct rdma_cm_id *id)
```

### **Description**

The **rdma\_get\_local\_addr** function returns the local IP address for an `rdma_cm_id` identifier that is associated with a local device.

### **Parameters**

<b>Item</b>	<b>Description</b>
<i>id</i>	Specifies the RDMA identifier.

### **Return Values**

The **rdma\_get\_local\_addr** function returns the local IP address for an `rdma_cm_id` identifier that has been bound with a local device.

## ***rdma\_get\_peer\_addr***

Returns the remote IP address of the associated communication identifier.

### **Syntax**

```
#include <rdma/rdma_cma.h>
struct sockaddr *rdma_get_peer_addr(struct rdma_cm_id *id)
```

## Description

The **rdma\_get\_peer\_addr** function returns the remote IP address that is associated with an `rdma_cm_id` identifier.

## Parameters

Item	Description
<i>id</i>	Specifies the RDMA identifier.

## Return Values

The **rdma\_get\_peer\_addr** function returns a pointer to the `sockaddr` address of the connected peer. If the `rdma_cm_id` identifier is not connected, the contents of the `sockaddr` structure is set to zero.

## **rdma\_create\_ep**

Creates an identifier (**rdma\_cm\_id**) to track information about communication.

## Syntax

```
#include <rdma/rdma_cma.h>
int rdma_create_ep [struct rdma_cm_id **id, struct rdma_addrinfo *res,
struct ibv_pd *pd, struct ibv_qp_init_attr *qp_init_attr,];
```

## Description

The **rdma\_cm\_id** identifier allocates a communication identifier and an optional queue pair (QP). The **rdma\_cm\_id** identifier is used in one of the following methods:

- If the **rdma\_cm\_id** identifier is used on the active side of a connection, the `RAI_PASSIVE` option is not set on the `res->ai_flag` flag. The **rdma\_create\_ep** function automatically creates a QP on the **rdma\_cm\_id** identifier if the `qp_init_attr` value is not NULL. If the domain is provided, the QP is associated with the specified protection domain; otherwise, a default protection domain is used. After calling the **rdma\_create\_ep** function, the **rdma\_cm\_id** identifier that is returned can be connected by calling the **rdma\_connect** function. The active side calls the **rdma\_resolve\_addr** function, and the **rdma\_resolve\_route** function is not necessary.
- If the **rdma\_cm\_id** identifier is used on the passive side of a connection, the `RAI_PASSIVE` option is set on the `res->ai_flag` flag. This function call saves the value of the `pd` and `qp_init_attr` parameters. A new connection request is retrieved by calling the **rdma\_get\_request** function. The **rdma\_cm\_id** identifier associated with the new connection is automatically associated with a QP by using the `pd` and `qp_init_attr` parameters. After calling the **rdma\_create\_ep** function, the **rdma\_cm\_id** identifier can be placed into a listening state by calling the **rdma\_listen** function. The passive side call the **rdma\_bind\_addr** is not necessary. The **rdma\_get\_request** function can be used to retrieve the connection. The **rdma\_cm\_id** identifier that is created is used for synchronous operation. To choose the asynchronous operation you must move the **rdma\_cm\_id** identifier to a user-created event channel by using the **rdma\_migrate\_id** function.

## Parameters

Item	Description
<i>id</i>	Specifies a reference by which the allocated communication identifier must be returned .
<i>res</i>	Specifies the address information that is associated with the <b>rdma_cm_id</b> identifier that is returned from the <b>rdma_getaddrinfo</b> function.

Item	Description
<i>pd</i>	Specifies the optional protection domain if a QP is associated with the <b>rdma_cm_id</b> identifier.
<i>qp_init_attr</i>	Specifies the optional initial, QP attributes.

## Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

### ***rdma\_destroy\_ep***

Destroys the specified communication identifier and its associated resources.

## Syntax

```
#include <rdma/rdma_cma.h>
int rdma_destroy_ep (struct rdma_cm_id *id)
```

## Description

The **rdma\_destroy\_ep** function destroys the specified **rdma\_cm\_id** identifier and all its associated resources. The **rdma\_destroy\_ep** function automatically destroys any queue pair (QP) that is associated with the **rdma\_cm\_id** identifier.

## Parameters

Item	Description
<i>id</i>	Specifies the communication identifier to destroy.

## Return Values

Returns 0 on success, or -1 on error. If an error occurs, then **errno** is set to indicate the reason for failure.

### ***rdma\_getaddrinfo***

Translates the transport independent address to establish communication.

## Syntax

```
#include <rdma/rdma_cma.h>
int rdma_getaddrinfo (char *node, char *service, struct rdma_addrinfo *hints, struct
rdma_addrinfo **res);
```

## Description

The **rdma\_getaddrinfo** function resolves the destination node and service address and returns information that is required to establish communication. The function provides the RDMA functional equivalent to **getaddrinfo**.

### Notes:

You must specify either node or service parameters for the translation. If hints are provided, the operation is controlled by the **hints.ai\_flags** flag. If the **RAI\_PASSIVE** flag is specified, the call resolves the address information that is used on the passive side of a connection.

Item	Description
<b>ai_flags</b>	<p data-bbox="732 184 1370 243">Specifies the hint flags that control the operation. The following flags are supported:</p> <ul data-bbox="732 264 1468 533" style="list-style-type: none"> <li data-bbox="732 264 1435 323">• <b>RAI_PASSIVE</b>: Indicates that the results are used on the passive or listening side of a connection.</li> <li data-bbox="732 344 1468 457">• <b>RAI_NUMERICHOST</b>: Indicates that if the flag is specified and if the node parameter is provided, the network address must be a numerical value. This flag suppresses any lengthy address resolution.</li> <li data-bbox="732 478 1403 533">• <b>RAI_NOROUTE</b>: Indicates that if the flag is set, the flag suppresses any lengthy route resolution.</li> </ul>
<b>ai_family</b>	<p data-bbox="732 562 1435 655">Specifies the address family for the source and destination address. The supported families are <b>AF_IB</b>, <b>AF_INET</b>, and <b>AF_INET6</b>.</p>
<b>ai_qp_type</b>	<p data-bbox="732 676 1451 768">Indicates the type of RDMA QP used for communication. The types that are supported are <b>IBV_UD</b> (unreliable datagram) and <b>IBV_RC</b> (reliable connected).</p>
<b>ai_port_space</b>	<p data-bbox="732 789 1451 848">Indicates the RDMA port space that is in use. The supported values are <b>RDMA_PS_UDP</b> and <b>RDMA_PS_TCP</b>.</p>
<b>ai_src_len</b>	<p data-bbox="732 869 1451 987">Indicates the length of the source address that is referenced by the <b>ai_src_addr</b> flag. If an appropriate source address for a given destination is not discovered the value of the <b>ai_src_len</b> flag is 0.</p>
<b>ai_dst_len</b>	<p data-bbox="732 1008 1435 1100">Indicates the length of the destination address that is referenced by <b>ai_src_addr</b> flag. This flag is set to 0, if the <b>RAI_PASSIVE</b> flag was specified as part of the hints.</p>
<b>ai_src_addr</b>	<p data-bbox="732 1121 1451 1180">Specifies the address for the local RDMA device, if the RDMA device is provided.</p>
<b>ai_dst_addr</b>	<p data-bbox="732 1201 1451 1260">Specifies the destination address for the RDMA device, if the RDMA device is provided.</p>
<b>ai_src_canonname</b>	<p data-bbox="732 1281 1182 1306">Specifies the canonical for the source.</p>
<b>ai_dst_canonname</b>	<p data-bbox="732 1327 1237 1352">Specifies the canonical for the destination.</p>
<b>ai_route_len</b>	<p data-bbox="732 1373 1468 1491">Specifies the size of the routing information buffer that is referenced by the <b>ai_route</b> flag. If the transport does not require routing data or none of the address could be resolved, the <b>ai_route</b> flag is 0.</p>
<b>ai_connect_len</b>	<p data-bbox="732 1512 1435 1671">Specifies the routing information for RDMA transports that require routing data for establishing the connection. The format of the routing data depends on the underlying transport. If InfiniBand transports are used, the <b>ai_route</b> flag references an array of <b>ibv_path_data</b> structures.</p>
<b>ai_connect</b>	<p data-bbox="732 1692 1468 1810">Specifies the size of connection information referenced by <b>ai_route</b> flag. If the underlying transport does not require any additional information to establish connection, the <b>ai_connect</b> flag is set to 0.</p>
<b>ai_next</b>	<p data-bbox="732 1831 1468 1890">Specifies the pointer to the next <b>rdma_addrinfo</b> structure in the list. The <b>ai_next</b> flag is NULL if no structures exist.</p>

## Parameters

Item	Description
<i>hints</i>	Specifies a reference to a <code>rdma_addrinfo</code> structure containing hints about the type of service the caller supports.
<i>node</i>	Specifies the optional name, dotted-decimal IPv4 or IPv6 hexadecimal address that must be resolved.
<i>res</i>	Specifies a pointer to a linked list of <code>rdma_addrinfo</code> structures that contains the response information.
<i>service</i>	Specifies the service name or port number of the address.

## Return Values

Returns 0 on success, or -1 on error. If an error occurs, then **errno** is set to indicate the reason for failure.

### ***rdma\_notify***

Notifies the asynchronous events that occurred on a queue pair (QP).

## Syntax

```
#include <rdma/rdma_cma.h>
int rdma_notify (struct rdma_cm_id *id, enum ibv_event_type event);
```

## Description

Notifies the `librdmacm` of asynchronous events that occurred on a queue pair (QP) associated with the `rdma_cm_id` identifier.

**Note:** Asynchronous events that occur on a QP are reported through the device of the user event handler. This routine is used to notify the `librdmacm` of communication events. In most cases, use of this routine is not necessary. However if connection establishment is done out of band (such as InfiniBand), it is possible to receive data on a QP that is not yet considered connected. This routine force the connection into an established state in order to handle situations where the connection never forms on its own. Calling this routine ensures the delivery of the **RDMA\_CM\_EVENT\_ESTABLISHED** event to the application. Events to be reported to the communication manager (CM) are **IB\_EVENT\_COMM\_EST**.

## Parameters

### **id**

RDMA identifier.

### **event**

Asynchronous event.

## Exit Status

### **= 0**

Success.

### **= -1**

Error. See **errno** for more details on the error.

## Event Handling Operations

Lists the event handling operations for the library verbs such as to get an event channel, acknowledge an event channel, and providing a string representation of the event channel.

### *rdma\_get\_cm\_event*

Retrieves the next pending communication event.

### Syntax

```
#include <rdma/rdma_cma.h>
int rdma_get_cm_event(struct rdma_event_channel *channel, struct rdma_cm_event **event);
```

### Description

The **rdma\_get\_cm\_event** function retrieves a communication event. If no events are pending, the call is blocked until an event is received.

#### Notes:

- You can change the file descriptor that is associated with the channel and change the default synchronous behavior of the **rdma\_get\_cm\_event** operation.
- All events that are reported must be acknowledged by running the **rdma\_ack\_cm\_event** operation.
- The `rdma_cm_id` identifier is not destroyed until the related events are acknowledged.

### Parameters

Item	Description
<i>channel</i>	Specifies the event channel to check for events.
<i>event</i>	Specifies the allocated information about the next communication event.

### Return Values

Item	Description
0	On success.
-1	Error, see <b>errno</b> . If an error occurs, the <b>errno</b> indicates the reason for failure.

### Event Data

The details of the communication event are returned to the **rdma\_cm\_event** function. This structure is allocated by the `rdma_cm` identifier and released by the **rdma\_ack\_cm\_event** operation. The **rdma\_cm\_event** function has the following parameters:

Item	Description
<i>id</i>	Specifies the <code>rdma_cm</code> identifier that is associated with the event. If <code>RDMA_CM_EVENT_CONNECT_REQUEST</code> is the event type, then for communication a new ID is referenced by the function.
<i>listen_id</i>	Specifies the corresponding listening request identifier for the <code>RDMA_CM_EVENT_CONNECT_REQUEST</code> event type.
<i>event</i>	Specifies the type of communication event that occurred.

Item	Description
<i>status</i>	Returns asynchronous error information associated with an event. The status is zero if the operation was successful, otherwise the status value is non-zero and is either set to an <b>errno</b> or a transport specific value. For details on transport specific status values, see the event type information below.
<i>param</i>	Provides additional details based on the type of event. You must select the <i>conn</i> subfield based on the <b>rdma_port_space</b> function of the <i>rdma_cm_id</i> identifier that is associated with the event.

## Connection Event Data

The event parameters are related to the connected queue pair (QP) services and the RDMA\_PS\_TCP event type. The connection related event data is valid for RDMA\_CM\_EVENT\_CONNECT\_REQUEST and RDMA\_CM\_EVENT\_ESTABLISHED event types.

Item	Description
<i>private_data</i>	References any user-specified data that is associated with the event. The data referenced by this field matches the value specified by the remote side when running the <b>rdma_connect</b> or <b>rdma_accept</b> functions. If the event does not include any private data, the <i>private_data</i> field is NULL. The buffer referenced by this pointer is deallocated when running the <b>rdma_ack_cm_event</b> function.
<i>private_data_len</i>	Specifies the size of the private data buffer.  <b>Note:</b> The size of the private data buffer might be larger than the amount of private data sent by the remote side. Any additional space in the buffer is zeroed out.
<i>responder_resources</i>	Specifies the number of responder resources that is requested by the recipient. The <i>responder_resources</i> field must match the initiator depth specified by the remote node when running the <b>rdma_connect</b> and <b>rdma_accept</b> functions.
<i>initiator_dept</i>	Specifies the maximum number of outstanding RDMA read operations that the recipient holds. The <i>initiator_dept</i> field must match the responder resources specified by the remote node when running the <b>rdma_connect</b> and <b>rdma_accept</b> functions.
<i>flow_control</i>	Indicates whether the hardware level flow control is provided by the sender. This value is specific to the InfiniBand architecture.
<i>retry_count</i>	Indicates the number of times that the recipient must retry a send operation specific to the RDMA_CM_EVENT_CONNECT_REQUEST events. This value is specific to the InfiniBand architecture.
<i>nr_retry_count</i>	Indicates the number of times that the recipient must retry receiver not ready (RNR) NACK errors. This value is specific to the InfiniBand architecture.
<i>srq</i>	Specifies whether the sender is using a shared-receive queue. Currently, the field is not supported.
<i>qp_num</i>	Indicates the remote QP number for the connection.

## Event Types

The following types of communication events are reported.

Item	Description
RDMA_CM_EVENT_ADDR_RESOLVED	Indicates that the address resolution ( <i>rdma_resolve_addr</i> ) completed successfully.
RDMA_CM_EVENT_ADDR_ERROR	Indicates that the address resolution ( <i>rdma_resolve_addr</i> ) failed.

Item	Description
RDMA_CM_EVENT_ROUTE_RESOLVED	Indicates that the route resolution ( <code>rdma_resolve_route</code> ) completed successfully.
RDMA_CM_EVENT_ROUTE_ERROR	Indicates that the route resolution ( <code>rdma_resolve_route</code> ) failed.
RDMA_CM_EVENT_CONNECT_REQUEST	Indicates that there is a new connection request on the passive side.
RDMA_CM_EVENT_CONNECT_RESPONSE	Indicates that there is a successful response to a connection request on the active side. It is generated on <code>rdma_cm_id</code> identifiers without a QP associated with them.
RDMA_CM_EVENT_CONNECT_ERROR	Indicates that an error has occurred trying to establish a connection. This event type can be generated on the active or passive side of a connection.
RDMA_CM_EVENT_UNREACHABLE	Indicates that the remote server is not reachable or unable to respond to a connection request on the active side. This option is generated on the active side to notify the user that the remote server is not reachable or unable to respond to a connection request. If this event is generated in response to a UD QP resolution request over InfiniBand, the event status field contains an <b>errno</b> , if negative, or the status result carried in the IB CM SIDR REP message.
RDMA_CM_EVENT_REJECTED	Indicates that a connection request or response was rejected by the remote end point. The event status field contains the transport specific reject reason if available. For InfiniBand, this event carries the reason for rejection in the IB CM REJ message.
RDMA_CM_EVENT_ESTABLISHED	Indicates that a connection is established with the remote end point.
RDMA_CM_EVENT_DISCONNECTED	Indicates that the connection is disconnected.
RDMA_CM_EVENT_DEVICE_REMOVED	Indicates that the local RDMA device associated with the <code>rdma_cm_id</code> identifier is removed. When you receive this event, you must destroy the associated <code>rdma_cm_id</code> identifier.
RDMA_CM_EVENT_TIMEWAIT_EXIT	Indicates that the QP associated with a connection has exited its timewait state and is ready to be reused. After a QP is disconnected, it is maintained in a timewait state to allow any in flight packets to exit the network. After the timewait state is complete, the <code>rdma_cm</code> identifier reports this event.

### ***rdma\_ack\_cm\_event***

Frees a communication event.

### **Syntax**

```
#include <rdma/rdma_cma.h>
int rdma_ack_cm_event(struct rdma_cm_event *event);
```

### **Description**

All events that are allocated by the **`rdma_get_cm_event`** function must be released. There must be a one-to-one correspondence between the events that are retrieved from a queue and events being released. The **`rdma_ack_cm_event`** function releases the event structure and any memory that it references.

## Parameters

Item	Description
<i>event</i>	Specifies the event to be released.

## Return Values

The `rdma_ack_cm_event` function returns the following values:

Item	Description
0	On success.
-1	If an error occurs, <b>errno</b> specifies the reason for failure.

## *rdma\_event\_str*

Returns a string representation of an RDMA CM event.

## Syntax

```
#include <rdma/rdma_cma.h>
const char *rdma_event_str(enum rdma_cm_event_type event);
```

## Description

The `rdma_event_str` function returns a string representation of an asynchronous event.

## Parameters

Item	Description
<i>event</i>	Specifies an asynchronous event.

## Return Values

The `rdma_event_str` function returns a pointer to a static character string that corresponds to the event.

## Data transfer operations

Lists the verbs that are used in data transfer operations such to retrieve the work request, send and receive work request, post the status of the request.

## *rdma\_get\_recv\_comp*

Retrieves a completed work request for the receive operation.

## Syntax

```
#include <rdma/rdma_cma.h>
int rdma_get_recv_comp (struct rdma_cm_id *id, struct ibv_wc *wc);
```

## Description

The `rdma_get_recv_comp` operation specifies the information about the completed request. The operation returns the information by using the `wc` parameter, and uses the `wr_id` identifier to set the context of the request.

**Notes:** The `rdma_get_recv_comp` operation polls the receive completion queue that is associated with an `rdma_cm_id` identifier. If the queue is not complete, the call is blocked until the request is completed.

This call must be used on the **rdma\_cm\_id** identifiers that do not share change queues (CQs) with other **rdma\_cm\_id** identifiers, and must maintain separate CQs to send and receive completed work request.

## Parameters

Item	Description
<i>id</i>	Specifies a reference to a communication identifier to check the completion of the request.
<i>wc</i>	Specifies a reference to a work completion structure that must be filled.

## Return Values

Returns 0 on success, or -1 on error. If an error occurs, then **errno** is set to indicate the reason for failure.

### ***rdma\_get\_request***

Retrieves the connection request event that is pending.

## Syntax

```
#include <rdma/rdma_cma.h>
int rdma_get_request (struct rdma_cm_id *listen, struct rdma_cm_id **id);
```

## Description

Retrieves a connection request event that is in pending state. If no requests are in a pending state, the call is blocked until an event is received.

**Notes:** The `rdma_get_request` call must be used on the listening `rdma_cm_id` communication identifiers that are operating synchronously. You receive a new `rdma_cm_id` identifier that represents the connection request on successful completion of the call. The new `rdma_cm_id` identifier is related to event information that is associated with the request until one of the following requisites is satisfied:

- The `rdma_reject` and `rdma_accept` operations are called.
- The `rdma_destroy_id` identifier is called on the newly created identifier.

If queue pair (QP) attributes are associated with the listening endpoint, the `rdma_cm_id` identifier that is returned is related an allocated to queue pair (QP).

## Parameters

Item	Description
<i>id</i>	Specifies the communication identifier that is associated to the new connection.
<i>listen</i>	Specifies the communication identifier that is listening.

## Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

## ***rdma\_get\_send\_comp***

Retrieves a completed request for send, read, or write operation.

### **Syntax**

```
#include <rdma/rdma_cma.h>
int rdma_get_send_comp (struct rdma_cm_id *id, struct ibv_wc *wc);
```

### **Description**

Retrieves a completed work request for a send, RDMA read, or RDMA write operation. Information about the completed request is returned by using the *wc* parameter, which has the *wr\_id* identifier set to the context of the request.

**Notes:** The *rdma\_get\_send\_comp* operation polls the send completion queue that is associated with an *rdma\_cm\_id* identifier. If a completion request is not found, the *rdma\_get\_send\_comp* call blocks the queue until a request is completed. The *rdma\_get\_send\_comp* call must be used on *rdma\_cm\_id* identifiers that do not share change queues (CQs) with other *rdma\_cm\_id* identifiers, and the function maintains separate CQs for send and receive completion requests.

### **Parameters**

<b>Item</b>	<b>Description</b>
<i>id</i>	Specifies a reference to a communication identifier to check for completions.
<i>wc</i>	Specifies a reference to a work completion structure that must be filled.

### **Return Values**

Returns 0 on success, or -1 on error. If an error occurs, then **errno** is set to indicate the reason for failure.

## ***rdma\_post\_read***

Posts a work request for RDMA read operation.

### **Syntax**

```
#include <rdma/rdma_cma.h>
int rdma_post_read (struct rdma_cm_id *id, void *context, void *addr, size_t length, struct
ibv_mr *mr,
int flags, uint64_t remote_addr, uint32_t rkey);
```

### **Description**

Posts a work request to the send queue of the queue pair that is associated with the *rdma\_cm\_id* identifier. The contents of the remote memory region are read into the local data buffer.

**Notes:** The remote and local data buffers must be registered before running the read operation, and the buffers must be registered until the read operation is completed. The read operation does not post the work request to an *rdma\_cm\_id* identifier or to the corresponding queue pair until it is connected. The user-defined context that is associated with the read request is returned by using the work completion *wr\_id* identifier and the work request identifier field.

## Parameters

Item	Description
<i>addr</i>	Specifies the address of the local destination of the read request.
<i>context</i>	Specifies the user-defined context that is associated with the request.
<i>flags</i>	Specifies the optional flags that are used to control the read operation.
<i>id</i>	Specifies a reference to a communication identifier where the request is posted.
<i>rkey</i>	Specifies the registered memory key that is associated with the remote address.
<i>length</i>	Specifies the length of the read operation.
<i>mr</i>	Specifies the registered memory region that is associated with the local buffer.
<i>remote_addr</i>	Specifies the address of the remote registered memory to read the address.

## Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

### ***rdma\_post\_readv***

Posts a work request to the send queue for RDMA read operation.

## Syntax

```
#include <rdma/rdma_cma.h>
int rdma_post_readv (struct rdma_cm_id *id, void *context, struct ibv_sge *sgl, int nsge, int
flags,
uint64_t remote_addr, uint32_t rkey);
```

## Description

Posts a work request to the send queue of the queue pair that is associated with the `rdma_cm_id` identifier. The contents of the remote memory region are read into the local data buffer.

**Notes:** The remote and local data buffers must be registered before running the read operation and the buffers must be registered until the read operation is completed. The read operation does not post the work request to an `rdma_cm_id` identifier or the corresponding queue pair until the connection is established. The user-defined context that is associated with the read request is returned by using the work completion `wr_id` identifier, the work request identifier, and the work request identifier field.

## Parameters

Item	Description
<i>context</i>	Specifies the user-defined context that is associated with the request.
<i>flags</i>	Specifies the optional flags that is used to control the read operation.

Item	Description
<i>id</i>	Specifies a reference to a communication identifier where the request is posted.
<i>nsge</i>	Specifies the number of scatter-gather array entries that are present.
<i>rkey</i>	Specifies the registered memory key that is associated with the remote address.
<i>remote_addr</i>	Specifies the address of the remote registered memory to read the address.
<i>sgl</i>	Specifies a scatter-gather list of the destination buffers that is associated with the read operation.

## Return Values

Returns 0 on success, or -1 on error. If an error occurs, then **errno** is set to indicate the reason for failure.

## ***rdma\_post\_recv***

Posts a work request to receive an incoming message.

## Syntax

```
#include <rdma/rdma_cma.h>
int rdma_post_recv (struct rdma_cm_id *id, void *context, void *addr, size_t length,
struct ibv_mr *mr);
```

## Description

Posts a work request to the receive queue of the queue pair that is associated with the `rdma_cm_id` identifier. The posted buffer is queued to receive an incoming message that is sent by the remote peer.

**Notes:** You must make sure that a receive buffer is posted. The receive buffer must be large enough to contain all the sent data before the peer posts the corresponding send message. You must register the message buffer before it is posted by using the `mr` parameter specifying the registration. The buffer must be registered until the receive operation is completed. The messages can be posted to an `rdma_cm_id` identifier after a queue pair is associated with the message. If the `rdma_cm_id` identifier is allocated by using the `rdma_create_id` identifier, a queue pair is bound to an `rdma_cm_id` identifier after calling the `rdma_create_ep` operation or `rdma_create_qp` operation. The user-defined context that is associated with the receive request is returned by using the work completion `wr_id` identifier, the work request identifier, and the work request identifier field.

## Parameters

Item	Description
<i>addr</i>	Specifies the address of the memory buffer to post the work request.
<i>context</i>	Specifies the user-defined context that is associated with the request.
<i>id</i>	Specifies a reference to a communication identifier where the message buffer is posted.
<i>length</i>	Specifies the length of the memory buffer.
<i>mr</i>	Specifies the registered memory region that is associated with the posted buffer.

## Return Values

Returns 0 on success, or -1 on error. If an error occurs, then **errno** is set to indicate the reason for failure.

### ***rdma\_post\_recv***

Posts a work request to the send queue for RDMA read operation.

## Syntax

```
#include <rdma/rdma_cma.h>
int rdma_post_recv (struct rdma_cm_id *id, void *context, struct ibv_sge *sgl, int nsg);
```

## Description

Posts a work request to the send queue of the queue pair that is associated with the `rdma_cm_id` identifier. The contents of the remote memory region is read into the local data buffer.

**Notes:** You must make sure that a receive buffer is posted. The receive buffer must be large enough to contain all the sent data before the peer posts the corresponding send message. You must register the message buffer before it is posted by using the `mr` parameter by specifying the registration. The buffer must be registered until the receive operation is completed. The messages can be posted to an `rdma_cm_id` identifier after a queue pair is associated with the message. A queue pair is bound to an `rdma_cm_id` identifier after calling the `rdma_create_ep` operation or `rdma_create_qp` operation, if the `rdma_cm_id` identifier is allocated by using the `rdma_create_id` identifier. The user-defined context that is associated with the receive request is returned by using the work completion `wr_id` identifier, the work request identifier, and the work request identifier field.

## Parameters

Item	Description
<i>context</i>	Specifies the user-defined context that is associated with the request.
<i>id</i>	Specifies a reference to a communication identifier where the request is posted.
<i>nsg</i>	Specifies the number of scatter-gather array entries that are present.
<i>sgl</i>	Specifies a scatter-gather list of the destination buffers that is associated with the read operation.

## Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

### ***rdma\_post\_send***

Posts a work request to send a message.

## Syntax

```
#include <rdma/rdma_cma.h>
int rdma_post_send (struct rdma_cm_id *id, void *context, void *addr, size_t length, struct
ibv_mr *mr,
int flags);
```

## Description

Posts a work request to the send queue of the queue pair that is associated with the `rdma_cm_id` identifier. The contents of the posted buffer are sent to the remote peer of a connection.

**Notes:** You must make sure that the remote peer posts a receive request before processing the send operations. If the send request is using inline data, the message buffer must be registered before being posted with the `mr` parameter by specifying the registration. The buffer must remain registered until the send operation is completed. The send operation cannot be posted to an `rdma_cm_id` identifier or the corresponding queue pair until the send operation is connected. The user-defined context that is associated with the send request is returned to the user by using the work completion `wr_id` identifier, the work request identifier, and the work request identifier field.

## Parameters

Item	Description
<i>addr</i>	Specifies the address of the memory buffer to post the work request.
<i>context</i>	Specifies the user-defined context that is associated with the request.
<i>flags</i>	Specifies the optional flags that are used to control the send operation.
<i>id</i>	Specifies a reference to a communication identifier where the message buffer is posted.
<i>length</i>	Specifies the length of the memory buffer.
<i>mr</i>	Specifies the optional registered memory region that is associated with the posted buffer.

## Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

### *rdma\_post\_sendv*

Posts a work request to send a message.

## Syntax

```
#include <rdma/rdma_cma.h>
int rdma_post_sendv (struct rdma_cm_id *id, void *context, struct ibv_sge *slg, int nsgs,
int flags);
```

## Description

Posts a work request to the send queue of the queue pair that is associated with the `rdma_cm_id` identifier. The contents of the posted buffer are sent to the remote peer of a connection.

**Notes:** You must make sure that the remote peer posts a receive request before processing the send operations. If the send request is using inline data, the message buffer must be registered before being posted with the `mr` parameter by specifying the registration. The buffer must remain registered until the send operation is completed. The send operation cannot be posted to an `rdma_cm_id` identifier or the corresponding queue pair until the send operation is connected. The user-defined context that is associated with the send request is returned to the user by using the work completion `wr_id` identifier, the work request identifier, and the work request identifier field.

## Parameters

Item	Description
<i>context</i>	Specifies the user-defined context that is associated with the request.
<i>flags</i>	Specifies the optional flags that are used to control the send operation.
<i>id</i>	Specifies a reference to a communication identifier where the message buffer is posted.
<i>nsge</i>	Specifies the number of scatter-gather entries in the <i>slg</i> array.
<i>slg</i>	Specifies a scatter-gather list of memory buffers that is posted as a single request.

## Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

### ***rdma\_post\_ud\_send***

Posts a work request to send a datagram.

## Syntax

```
#include <rdma/rdma_cma.h>
int rdma_post_ud_send (struct rdma_cm_id *id, void *context, void *addr, size_t length,
struct ibv_mr *mr, int flags, struct ibv_ah *ah, uint32_t remote_qpn);
```

## Description

Posts a work request to the send queue of the queue pair that is associated with the `rdma_cm_id` identified. The contents of the posted buffer is sent to the specified destination of the queue pair.

**Notes:** You must make sure that the remote peer posts a receive request before processing the send operations. If the send request is using inline data, the message buffer must be registered before being posted with the `mr` parameter by specifying the registration. The buffer must remain registered until the send operation is completed. The send operation cannot be posted to an `rdma_cm_id` identifier or the corresponding queue pair until the send operation is connected. The user-defined context that is associated with the send request is returned to the user by using the work completion `wr_id` identifier, work request identifier, and field.

## Parameters

Item	Description
<i>addr</i>	Specifies the address of the memory buffer to post the work request.
<i>ah</i>	Specifies an address handle that describes the address of the remote node.
<i>context</i>	Specifies the user-defined context that is associated with the request.
<i>flags</i>	Specifies the optional flags that are used to control the send operation.
<i>id</i>	Specifies a reference to a communication identifier where the message buffer is posted.

Item	Description
<i>length</i>	Specifies the length of the memory buffer.
<i>mr</i>	Specifies the optional registered memory region that is associated with the posted buffer.
<i>remote_qpn</i>	Specifies the number of the destination queue pair.

## Return Values

Returns 0 on success, or -1 on error. If an error occurs, then **errno** is set to indicate the reason for failure.

## ***rdma\_post\_write***

Posts a work request for RDMA write operation.

## Syntax

```
#include <rdma/rdma_cma.h>
int rdma_post_write (struct rdma_cm_id *id, void *context, void *addr, size_t length, struct
ibv_mr *mr,
int flags, uint64_t remote_addr, uint32_t rkey);
```

## Description

Posts a work request to the send queue of the queue pair that is associated with the `rdma_cm_id` identifier. The contents of the local data buffer are written into the remote memory region.

**Notes:** The remote and local data buffers must be registered before you run the write operation. The buffers must be registered until the write operation is complete. The write operation does not post the work request to an `rdma_cm_id` identifier or the corresponding queue pair until it is connected. The user-defined context that is associated with the write request is returned by using the work completion `wr_id` identifier, the work request identifier, and the work request identifier field.

## Parameters

Item	Description
<i>addr</i>	Specifies the local address of the source that is related to the write request.
<i>context</i>	Specifies the user-defined context that is associated with the request.
<i>flags</i>	Specifies the optional flags that are used to control the write operation.
<i>id</i>	Specifies a reference to a communication identifier where the request is posted.
<i>rkey</i>	Specifies the registered memory key that is associated with the remote address.
<i>length</i>	Specifies the length of the write operation.
<i>mr</i>	Specifies the optional memory region that is associated with the local buffer.
<i>remote_addr</i>	Specifies the address of the remote registered memory to write the data.

## Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

## *rdma\_post\_writev*

Posts a work request for RDMA write operation.

## Syntax

```
#include <rdma/rdma_cma.h>
int rdma_post_writev (struct rdma_cm_id *id, void *context, struct ibv_sge *sgl, int nsge,
int flags, uint64_t remote_addr, uint32_t rkey);
```

## Description

Posts a work request to the send queue of the queue pair that is associated with the `rdma_cm_id` identifier. The contents of the local data buffer are written into the remote memory region.

**Notes:** The remote and local data buffers must be registered before you run the write operation. The remote and local data buffers must be registered until the write operation is completed. The write operation does not post the work request to an `rdma_cm_id` identifier or the corresponding queue pair until it is connected. The user-defined context that is associated with the write request is returned by using the work completion `wr_id` identifier, the work request identifier, and the work request identifier field.

## Parameters

Item	Description
<i>context</i>	Specifies the user-defined context that is associated with the request.
<i>flags</i>	Specifies the optional flags that are used to control the write operation.
<i>id</i>	Specifies a reference to a communication identifier where the request is posted.
<i>nsge</i>	Specifies the number of scatter-gather array entries.
<i>remote_addr</i>	Specifies the address of the remote registered memory to write the data.
<i>rkey</i>	Specifies the registered memory key that is associated with the remote address.
<i>sgl</i>	Specifies a scatter-gather list of the source buffer that is related to the write operation.

## Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

## Queue Pair Management

Lists the functions that help to manage queue pair (QP) such as to create QP, and to destroy QP.

### rdma\_create\_qp

Allocates a queue pair (QP).

#### Syntax

```
#include <rdma/rdma_cma.h>
int rdma_create_qp(struct rdma_cm_id *id, struct ibv_pd *pd, struct ibv_qp_init_attr
*qp_init_attr);
```

#### Description

The **rdma\_create\_qp** function allocates a queue pair (QP) that is associated with a specified `rdma_cm_id` identifier, and transitions it for sending and receiving.

#### Notes:

- The `rdma_cm_id` identifier must be associated with a local RDMA device before running the **rdma\_create\_qp** function, and the protection domain must be for the same device.
- QPs that are allocated to an `rdma_cm_id` identifier are automatically transitioned by the `librdmacm` library through their states. The QP is ready to handle posting of received data after the QP is allocated. If the QP is not connected, it is ready to post send data.
- If a protection domain is not specified then the `pd` parameter is NULL, then the `rdma_cm_id` identifier is created by using a default protection domain. One default protection domain is allocated per RDMA device. The initial QP attributes are specified by using the `qp_init_attr` parameter. The `send_cq` and `recv_cq` fields in the `ibv_qp_init_attr` are optional. If a send or receive completion queue (CQ) is not specified, then a CQ is allocated by the `rdma_cm` for the QP, along with corresponding completion channels. Completion channels and CQ data created by the `rdma_cm` can be accessed by user by using the `rdma_cm_id` structure. The actual capabilities and properties of the QP that is created is returned to the user through the `qp_init_attr` parameter.

#### Parameters

Item	Description
<i>id</i>	Specifies the communication identifier to create.
<i>pd</i>	Specifies the optional protection domain for the QP.
<i>qp_init_attr</i>	Specifies the initial QP attributes.

#### Return Values

The **rdma\_destroy\_event\_channel** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

### rdma\_destroy\_qp

Releases a queue pair (QP).

#### Syntax

```
#include <rdma/rdma_cma.h>
void rdma_destroy_qp(struct rdma_cm_id *id);
```

## Description

The **rdma\_destroy\_qp** function destroys a QP that is allocated to an `rdma_cm_id` identifier.

**Note:** You must destroy any QP that is associated with an `rdma_cm_id` identifier before deleting the ID.

## Parameters

Item	Description
<i>id</i>	Specifies the RDMA identifier.

## Return Value

The **rdma\_destroy\_event\_channel** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

## Device Management

Lists the functions that is used to manage devices, which includes to get devices, and free devices.

### rdma\_get\_devices

Gets a list of RDMA devices that are available.

## Syntax

```
#include <rdma/rdma_cma.h>
struct ibv_context **rdma_get_devices(int *num_devices);
```

## Description

The **rdma\_get\_devices** function returns a NULL-terminated array of open RDMA devices. You can use this routine to allocate resources on specific RDMA devices that is shared with multiple `rdma_cm_id` identifiers.

**Note:** The returned array must be released by running the **rdma\_free\_devices** function. Devices remain opened when the **librdmacm** library is loaded.

## Parameters

Item	Description
<i>num_devices</i>	Specifies the number of devices that are returned if the value is not NULL.

## Return Values

The **rdma\_get\_devices** function returns an array of available RDMA devices, or NULL if the request fails. If an error occurs, **errno** indicates the reason for failure.

### rdma\_free\_devices

Frees the list of devices that are returned by the **rdma\_get\_devices** function.

## Syntax

```
#include <rdma/rdma_cma.h>
void rdma_free_devices(struct ibv_context **list);
```

## Description

The **rdma\_free\_devices** function frees the device array that is returned by the **rdma\_get\_devices** function.

## Parameters

Item	Description
<i>list</i>	Specifies the list of devices that are returned from the <b>rdma_get_devices</b> function.

## Return Value

There is no return value.

## Memory region management

Lists the functions that is used to manage memory, which includes to register memory, to provide read and write access to memory, and to register the buffer for sending and receiving messages.

### rdma\_dereg\_mr

Deregisters a memory region that is registered.

## Syntax

```
#include <rdma/rdma_cma.h>
struct ibv_mr * rdma_dereg_mr (struct ibv_mr *mr);
```

## Description

Deregisters a memory buffer that is registered for RDMA or message operations. You must run the **rdma\_dereg\_mr** call for all registered memory that is associated with an **rdma\_cm\_id** identifier before you destroy the **rdma\_cm\_id** identifier.

**Note:** All memory buffers that is registered with an **rdma\_cm\_id** identifier are associated with the protection domain that is associated with the ID. You must deregister all registered memory before the protection domain can be destroyed.

## Parameters

Item	Description
<i>mr</i>	Specifies a reference to a registered memory buffer.

## Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

### rdma\_reg\_msgs

Registers the data buffer for sending or receiving messages.

## Syntax

```
#include <rdma/rdma_cma.h>
struct ibv_mr * rdma_reg_msgs (struct rdma_cm_id *id, void *addr, size_t length);
```

## Description

Registers an array of memory buffers that are used for sending and receiving messages or for RDMA operations. Memory buffers that are registered by using the `rdma_reg_msgs` function can be posted to an `rdma_cm_id` identifier by using one of the following operations:

- Run the **`rdma_post_send`** operation
- Run the **`rdma_post_recv`** operation
- Specify the buffer as the target of an RDMA read operation
- Specify the buffer as the source of an RDMA write request

**Note:** The `rdma_reg_msgs` operation registers an array of data buffers that are used to send and receive messages on a queue pair that is associated with an `rdma_cm_id` identifier. The memory buffer is registered with the protection domain that is associated with the identifier. The start of the data buffer array is specified by using the `addr` parameter, and the total size of the array is specified by the `length` parameter. All data buffers must be registered before being posted as a work request. You must unregister all the registered memory by using the `rdma_dereg_mr` operation.

## Parameters

Item	Description
<i>addr</i>	Specifies the address of the memory buffer to register.
<i>id</i>	Specifies a reference to a communication identifier where the message buffer must be used.
<i>length</i>	Specifies the total length of the memory to register.

## Return Values

Returns a reference to the registered memory region on success, or NULL on error. If an error occurs, **`errno`** is set to indicate the reason for failure.

## `rdma_reg_read`

Registers the data buffer for remote direct memory access (RDMA) read access.

## Syntax

```
#include <rdma/rdma_cma.h>
struct ibv_mr * rdma_reg_read (struct rdma_cm_id *id, void *addr, size_t length);
```

## Description

Registers a memory buffer that is accessed by a remote direct memory access (RDMA) read operation. Memory buffers that are registered by using the **`rdma_reg_read`** operation can be targeted in an RDMA read request. The memory buffer is specified on the remote side of an RDMA connection as the `remote_addr` parameter of **`rdma_post_read`** operation, or a similar operation.

**Notes:** The **`rdma_reg_read`** operation registers a data buffer that is the target of an RDMA read operation on a queue pair that is associated with an **`rdma_cm_id`** identifier. The memory buffer is registered with the protection domain that is associated with the identifier. The start of the data buffer array is specified by using the `addr` parameter, and the total size of the array is specified by the `length` parameter. All data buffers must be registered before being posted as a work request. You must unregister all the registered memory by using the **`rdma_dereg_mr`** operation.

## Parameters

Item	Description
<i>addr</i>	Specifies the address of the memory buffer to register.
<i>id</i>	Specifies a reference to a communication identifier where the message buffer must be used.
<i>length</i>	Specifies the total length of the memory to register.

## Return Values

Returns a reference to the registered memory region on success, or NULL on error. If an error occurs, the **errno** is set to indicate the reason for failure.

## rdma\_reg\_write

Registers the data buffer for remote direct memory access (RDMA) write access.

## Purpose

## Syntax

```
#include <rdma/rdma_cma.h>
struct ibv_mr * rdma_reg_write (struct rdma_cm_id *id, void *addr, size_t length);
```

## Description

Registers a memory buffer that is accessed by a remote RDMA write operation. Memory buffers that are registered by using the **rdma\_reg\_write** operation can be targeted in an RDMA write request. The memory buffer is specified on the remote side of an RDMA connection as the `remote_addr` parameter of **rdma\_post\_write** operation, or a similar operation.

**Notes:** The **rdma\_reg\_write** operation registers a data buffer that is the target of an RDMA write operation on a queue pair that is associated with an **rdma\_cm\_id** identifier. The memory buffer is registered with the protection domain that is associated with the identifier. The start of the data buffer array is specified by using the `addr` parameter, and the total size of the array is specified by the `length` parameter. All data buffers must be registered before being posted as a work request. You must unregister all the registered memory by using the **rdma\_dereg\_mr** operation.

## Parameters

Item	Description
<i>addr</i>	Specifies the address of the memory buffer to register.
<i>id</i>	Specifies a reference to a communication identifier where the message buffer must be used.
<i>length</i>	Specifies the total length of the memory to register.

## Return Values

Returns a reference to the registered memory region on success, or NULL on error. If an error occurs, the **errno** is set to indicate the reason for failure.

# Libibverbs Library

---

The libibverbs library enables user-space processes to use remote direct memory access (RDMA) verbs as described in the InfiniBand Architecture Specification.

You can find information about the libibverbs library in the `/usr/include/rdma/verbs.h` file that is delivered with the libibverbs library sources.

Man pages are created to describe the various interfaces and test programs. For a full list, you can refer to the **verbs** man page.

## Returned error rules

Lists the errors returned by the Libibverbs library.

The values returned by the commands and their interpretation are as follows:

- The commands return 0 on success.
- The commands return NULL, -1, or the value **errno** that indicates the reason of failure.
- The commands return **ENOSYS** when the verb is not supported.

## Supported Verbs

Lists the supported verbs and their functions for the Libibverbs library.

### Device management

Lists the functions that manage devices for the libibverbs library.

#### ***ibv\_get\_device\_list***, ***ibv\_free\_device\_list***

Obtains and releases the list of available RDMA devices.

### Syntax

```
#include <rdma/verbs.h>
struct ibv_device **ibv_get_device_list(int *num_devices);
void ibv_free_device_list(struct ibv_device **list);
```

### Description

The **ibv\_get\_device\_list()** function returns a NULL-terminated array of RDMA devices that are available. The argument *num\_devices* is optional and if it is NULL, it is set to the number of devices returned in the array.

The **ibv\_free\_device\_list()** function frees the array of devices list that is returned by the **ibv\_get\_device\_list()** function.

**Note:** The client code must open all the devices it intends to use with the **ibv\_open\_device()** operation before running the **ibv\_free\_device\_list()** function. When the **ibv\_free\_device\_list()** function frees the array, the system can use the open devices, and the pointers to unopened devices is no longer valid.

### Errors

Error	Descriptor
EPERM	Permission denied.
ENOSYS	No kernel support for RDMA.
ENOMEM	Insufficient memory to complete the operation.

## Output Parameters

Item	Description
<i>num_devices</i>	(Optional) If not null, the number of devices returned in the array is stored in this parameter.

## Return Value

The **ibv\_get\_device\_list()** function returns the array of available RDMA devices, or NULL if the request fails. If no devices are found then **num\_devices** is set to 0, and a non-NULL is returned.

The **ibv\_free\_device\_list()** function returns no value.

### ***ibv\_get\_device\_name***

Obtains the name of the RDMA device.

## Syntax

```
#include <rdma/verbs.h>
const char *ibv_get_device_name(struct ibv_device *device);
```

## Description

The **ibv\_get\_device\_name** function returns a pointer to the device name that is contained within the struct **ibv\_device**.

## Parameters

Item	Description
<i>device</i>	Specifies the struct <b>ibv_device</b> for the required device.

## Return Value

The **ibv\_get\_device\_list()** function returns a pointer to the device name on success, and NULL if the request fails.

### ***ibv\_get\_device\_guid***

Returns the string that describes the **event\_type**, **node\_type**, and **port\_state** for the **enum** values.

## Syntax

```
#include <rdma/verbs.h>
uint64_t ibv_get_device_guid(struct ibv_device *device);
```

## Description

The **ibv\_get\_device\_guid** function returns a 64-bit global unique identifier (GUID) for the devices in the network byte order.

## Parameters

Item	Description
<i>device</i>	Specifies the struct <b>ibv_device</b> for the device.

## Return Value

The **ibv\_get\_device\_guid** function returns **uint64\_t** on success, and **0** on failure.

If the device is NULL, the open or write operation failed on the `/dev/rdma/ofed_admin` administrator device.

## **ibv\_open\_device, ibv\_close\_device**

Opens and closes an remote device memory access (RDMA) device context.

## Syntax

```
#include <rdma/verbs.h>
struct ibv_context *ibv_open_device(struct ibv_device *device);
int ibv_close_device(struct ibv_context *context);
```

## Description

The **ibv\_open\_device()** function opens the device *device*, and creates a context for further use.

The **ibv\_close\_device()** function closes the device context *context*.

**Note:** The **ibv\_close\_device()** function does not release all the resources that are allocated by using the parameter *context*. To avoid resource leaks, you must release all the associated resources before closing a context.

## Parameter

Item	Description
<i>devices</i>	Specifies the struct <b>ibv_device</b> for the required device.

## Return Value

The **ibv\_open\_device** and **ibv\_close\_device** functions return a verb context that can be used for future operations on the device on successful completion. The function returns NULL if the device is NULL, or if the open operation fails.

## **ibv\_query\_device**

Queries the attributes of an RDMA device.

## Syntax

```
#include <rdma/verbs.h>
int ibv_query_device(struct ibv_context *context, struct ibv_device_attr *device_attr)
```

## Description

The **ibv\_query\_device()** function returns the attributes of the device with context *context*. The parameter *device\_attr* is a pointer to an **ibv\_device\_attr** struct as defined in the `<rdma/verbs.h>` file.

**Note:** The maximum values that are returned by the **ibv\_query\_device()** function are the upper limits of the supported resources by the device. It is not possible to use these maximum values because the actual number of any resource that can be created is limited by the system configuration, the amount of host memory, user permissions, and the amount of resources in use.

## Input Parameter

Item	Description
<i>context</i>	Specifies the struct <code>ibv_context</code> from the <code>ibv_open_device</code> function.

## Output Parameter

Item	Description
<i>device_attr</i>	Specifies the struct <code>ibv_device_attr</code> that contains the device attributes.

## Return Values

Item	Description
0	On success.
<code>errno</code>	On failure.

## *ibv\_query\_port*

Queries the attributes of an RDMA port.

## Syntax

```
#include <rdma/verbs.h>
int ibv_query_port(struct ibv_context *context, uint8_t port_num, struct ibv_port_attr
*port_attr)
```

## Description

The `ibv_query_port()` function returns the attributes of port *port\_num* for device context *context* through the pointer *port\_attr*. The parameter *port\_attr* is an `ibv_port_attr` struct, as defined in the `<rdma/verbs.h>` file.

The struct `ibv_port_attr` is as follows:

```
struct ibv_port_attr {
enum ibv_port_state state; /* Logical port state */
enum ibv_mtu max_mtu; /* Max MTU supported by port */
enum ibv_mtu active_mtu; /* Actual MTU */
enum ibv_mtu gid_tbl_len; /* Length of source GID table */
int port_cap_flags; /* Port capabilities */
uint32_t max_msg_sz; /* Maximum message size */
uint32_t bad_pkey_cntr; /* Bad P_Key counter */
uint32_t qkey_viol_cntr; /* Q_Key violation counter */
uint16_t pkey_tbl_len; /* Length of partition table */
uint16_t lid; /* Base port LID */
uint16_t sm_lid; /* SM LID */
uint8_t lmc; /* LMC of LID */
uint8_t max_vl_num; /* Maximum number of VLS */
uint8_t sm_sl; /* SM service level */
uint8_t subnet_timeout; /* Subnet propagation delay */
uint8_t init_type_reply; /* Type of initialization performed by SM */
uint8_t active_width; /* Currently active link width */
uint8_t active_speed; /* Currently active link speed */
uint8_t phys_state; /* Physical port state */
uint8_t link_layer; /* link layer protocol of the port */
};
```

## Input Parameters

Item	Description
<i>context</i>	Specifies the struct <code>ibv_context</code> from the <code>ibv_open_device</code> function.

Item	Description
<i>port_num</i>	Specifies the physical port number (1 is the first port).

## Output Parameter

Item	Description
<i>port_attr</i>	Specifies the struct <b>ibv_port_attr</b> that contains the port attributes.

## Return Values

Item	Description
0	On success.
errno	On failure.

## **ibv\_query\_pkey**

Queries the P\_Key table of an remote direct memory access (RDMA) port.

## Syntax

```
#include <rdma/verbs.h>
int ibv_query_pkey(struct ibv_context *context, uint8_t port_num, int index, uint16_t *pkey)
```

## Description

The **ibv\_query\_pkey()** function returns the P\_Key value in the entry *index* of port *port\_num* for device context *context* through the pointer *pkey*.

## Input Parameters

Item	Description
<i>context</i>	Specifies the valid context pointer that is returned by the <b>ibv_open_device()</b> function.
<i>port_num</i>	Specifies the valid port number for the device that is returned by the <b>ibv_query_device()</b> function.
<i>index</i>	Specifies the valid index for the <i>port_num</i> parameter from attributes that are returned by the <b>ibv_query_port()</b> function.

## Output Parameter

Item	Description
<i>pkey</i>	Specifies the valid pointer to store protection key.

## Return Values

Item	Description
0	On success.
-1	The request fails because of the following reasons: <ul style="list-style-type: none"> <li>• The <i>context</i> or <i>pkey</i> parameter is NULL</li> <li>• The open or write operation failed on the <code>/dev/rdma/ofed_admin</code> administrator device.</li> </ul>

## ***ibv\_query\_gid***

Gets the group ID (GID) that is the network interface controller (NIC)'s Media Control Access (MAC) address.

### **Syntax**

```
#include <rdma/verbs.h>
int ibv_query_gid(struct ibv_context *context, uint8_t port_num, int index, union ibv_gid *gid)
```

### **Description**

The **ibv\_query\_gid()** function returns the MAC address of the NIC in the `subnet_prefix` parameter and 0 in the `interface_id` identifier.

### **Input Parameters**

<b>Item</b>	<b>Description</b>
<i>context</i>	Specifies the context pointer that is returned by the <b>ibv_open_device()</b> function.
<i>port_num</i>	Specifies the port number for the device that is returned by the <b>ibv_query_device()</b> function.
<i>index</i>	Specifies the index for the <i>port_num</i> parameter that is derived from the attributes that are returned by the <b>ibv_query_port()</b> function.

### **Output Parameter**

<b>Item</b>	<b>Description</b>
<i>gid</i>	Specifies the pointer where the group ID (GID) can be stored.

### **Return Values**

<b>Item</b>	<b>Description</b>
0	On success.
-1	The request fails because of one of the following reasons: <ul style="list-style-type: none"><li>• The <i>context</i> or <i>gid</i> parameter is NULL.</li><li>• The open or write operation failed on the <code>/dev/rdma/ofed_admin</code> administrator device.</li></ul>

```
ibv_gid
union ibv_gid
{
    uint8_t    raw[16];
    struct
    {
        uint64_t subnet_prefix;
        uint64_t interface_id;
    } global;
};
```

## Queue pair management

Lists the functions that are used to manage the queue.

### ***ibv\_create\_qp, ibv\_destroy\_qp***

Creates or destroys a queue pair (QP).

### Syntax

```
#include <rdma/verbs.h>
struct ibv_qp *ibv_create_qp(struct ibv_pd *pd,
struct ibv_qp_init_attr *qp_init_attr);int ibv_destroy_qp(struct ibv_qp *
qp)
```

### Description

The **ibv\_create\_qp()** function creates a queue pair (QP) that is associated with the *pd* protection domain. The *qp\_init\_attr* argument is an *ibv\_qp\_init\_attr* struct that is defined in the `<rdma/verbs.h>` file.

Name of the Struct	Item	File name	Description
struct ibv_qp_init_attr {	void	*qp_context;	/*Associated context of the QP*/
	struct ibv_cq	*send_cq;	/*CQ to be associated with the Send Queue (SQ)*/
	struct ibv_cq	*recv_cq;	/*CQ to be associated with the Receive Queue (RQ)*/
	struct ibv_srq	*srq;	/*Not Supported*/
	struct ibv_qp_cap	cap;	/*QP capabilities*/
	enum ibv_qp_type	qp_type;	/* QP Transport Service Type: IBV_QPT_RC, IBV_QPT_UC, IBV_QPT_UD, IBV_QPT_XRC or IBV_QPT_RAW_PACKET */
	int	sq_sig_all;	/*If set, each Work Request (WR) submitted to the SQ generates a completion entry*/
struct ibv_qp_cap {	struct ibv_xrc_domain	xrc_domain;	/*Not supported*/
	uint32_t	max_send_wr;	/*Requested maximum number of outstanding WRs in the SQ*/
	uint32_t	max_recv_wr;	/*Requested maximum number of outstanding WRs in the RQ*/
	uint32_t	max_send_sge;	/*Requested maximum number of Scatter-gather elements in a WR in the SQ*/
	uint32_t	max_recv_sge;	/*Requested maximum number of Scatter-gather elements in a WR in the RQ*/
uint32_t	max_inline_data;	/*Requested max number of data (bytes)that can be posted inline to the SQ, otherwise 0*/	

### Input Parameters

#### Item

*pd*

*qp\_init\_attr*

#### Descriptor

struct **ibv\_pd** from **ibv\_alloc\_pd**.

Initial attributes of queue pair.

## Output Parameters

Item	Description
<i>qp_init_attr</i>	Actual values that are entered.

## Return Value

The **ibv\_create\_qp()** function returns a pointer to the created QP on success, or NULL if the request fails.

The **ibv\_destroy\_qp()** function returns 0 on success, or **errno** on failure that indicates the reason for failure.

## *ibv\_modify\_qp*

Modifies the attributes of a queue pair (QP).

## Syntax

```
#include <rdma/verbs.h>

int ibv_modify_qp(struct ibv_qp *qp, struct ibv_qp_attr *attr, int ibv_qp_attr_mask attr_mask)
```

Queue pairs (QP) must be taken through an incremental sequence of states before using QP for communication.

The following table indicates the QP states:

Item	Descriptor
RESET	Newly created queues that are empty.
INIT	The basic information required for the queue is set and the queue is ready for posting to receive queue.
RTR	The queue is ready to receive. The remote address information is set for the connected QPs, and the QP can receive packets.
RTS	The queue is ready to send. The timeout and retry parameters are set. The QP can send packets.

The state transitions are used with **ibv\_modify\_qp** function.

## Description

The **ibv\_modify\_qp()** function modifies the attributes of a QP *qp* with the attributes in *attr* parameter according to the *attr\_mask* mask. The *attr* parameter is an **ibv\_qp\_attr** struct, as defined in the `<rdma/verbs.h>` file.

```
struct ibv_qp_attr {
enum ibv_qp_state qp_state; /* Move the QP to this state */
enum ibv_qp_state cur_qp_state; /* Assume this is the current QP state */
enum ibv_mtu path_mtu; /* Path MTU (valid only for RC/UC QPs) */
enum ibv_mig_state path_mig_state; /* Path migration state (valid if HCA supports APM) */
uint32_t qkey; /* Q_Key for the QP (valid only for UD QPs) */
uint32_t rq_psn; /* PSN for receive queue (valid only for RC/UC QPs) */
uint32_t sq_psn; /* PSN for send queue (valid only for RC/UC QPs) */
uint32_t dest_qp_num; /* Destination QP number (valid only for RC/UC QPs) */
int qp_access_flags; /* Mask of enabled remote access operations (valid only
for RC/UC QPs) */
struct ibv_qp_cap cap; /* QP capabilities (valid if HCA supports QP resizing) */
struct ibv_ah_attr ah_attr; /* Primary path address vector (valid only
for RC/UC QPs) */
struct ibv_ah_attr alt_ah_attr; /* Alternate path address vector (valid only
for RC/UC QPs) */
uint16_t pkey_index; /* Primary P_Key index */
uint16_t alt_pkey_index; /* Alternate P_Key index */
uint8_t en_sqd_async_notify; /* Enable SQD.drained async notification (Valid only
if qp_state is SQD) */
uint8_t sq_draining; /* Is the QP draining? Irrelevant for ibv_modify_qp() */
uint8_t max_rd_atomic; /* Number of outstanding RDMA reads & atomic operations
on the destination QP (valid only for RC QPs) */
uint8_t max_dest_rd_atomic; /* Number of responder resources for handling incoming
RDMA reads & atomic operations (valid only
for RC QPs) */
uint8_t min_rnr_timer; /* Minimum RNR NAK timer (valid only for RC QPs) */
};
```

```

uint8_t      port_num;          /* Primary port number */
uint8_t      timeout;         /* Local ack timeout for primary path (valid only
                               for RC QPs) */
uint8_t      retry_cnt;       /* Retry count (valid only for RC QPs) */
uint8_t      rnr_retry;       /* RNR retry (valid only for RC QPs) */
uint8_t      alt_port_num;    /* Alternate port number */
uint8_t      alt_timeout;     /* Local ack timeout for alternate path (valid only
                               for RC QPs) */
};

```

The *attr\_mask* parameter specifies the QP attributes to be modified. The argument is either 0 or bitwise OR of one or more of the following flags:

**IBV\_QP\_STATE**

Modify qp\_state

**IBV\_QP\_CUR\_STATE**

Set cur\_qp\_state

**IBV\_QP\_EN\_SQD\_ASYNC\_NOTIFY**

Set en\_sqd\_async\_notify

**IBV\_QP\_ACCESS\_FLAGS**

Set qp\_access\_flags

**IBV\_QP\_PKEY\_INDEX**

Set pkey\_index

**IBV\_QP\_PORT**

Set port\_num

**IBV\_QP\_QKEY**

Set qkey

**IBV\_QP\_AV**

Set ah\_attr

**IBV\_QP\_PATH\_MTU**

Set path\_mtu

**IBV\_QP\_TIMEOUT**

Set timeout

**IBV\_QP\_RETRY\_CNT**

Set retry\_cnt

**IBV\_QP\_RNR\_RETRY**

Set rnr\_retry

**IBV\_QP\_RQ\_PSN**

Set rq\_psn

**IBV\_QP\_MAX\_QP\_RD\_ATOMIC**

Set max\_rd\_atomic

**IBV\_QP\_ALT\_PATH**

Set the alternative path through alt\_ah\_attr, alt\_pkey\_index, alt\_port\_num, alt\_timeout

**IBV\_QP\_MIN\_RNR\_TIMER**

Set min\_rnr\_timer

**IBV\_QP\_SQ\_PSN**

Set sq\_psn

**IBV\_QP\_MAX\_DEST\_RD\_ATOMIC**

Set max\_dest\_rd\_atomic

**IBV\_QP\_PATH\_MIG\_STATE**

Set path\_mig\_state

**IBV\_QP\_CAP**

Set cap

**IBV\_QP\_DEST\_QPN**

Set dest\_qp\_num

## Notes:

- If any of the modify attributes or the modify mask is invalid, none of the attributes are modified (including the QP state).
- Not all devices support resizing QPs. To determine whether a device supports resizing of the QP, check whether the `IBV_DEVICE_RESIZE_MAX_WR` bit is set in the device capabilities flags.
- Not all devices support alternate paths. To determine whether a device supports alternate paths, check whether the `IBV_DEVICE_AUTO_PATH_MIG` bit is set in the device capabilities flags.
- The following tables indicate the type of the QP transport service, the minimum list of attributes that must be changed after changing the QP state from Reset to Init to RTR to RTS state.

The types of QP transport service for the `IBV_QPT_UD` type, follow:

Next state	Required attributes
Init	<code>IBV_QP_STATE</code> , <code>IBV_QP_PKEY_INDEX</code> , <code>IBV_QP_PORT</code> , <code>IBV_QP_QKEY</code>
RTR	<code>IBV_QP_STATE</code>
RTS	<code>IBV_QP_STATE</code> , <code>IBV_QP_SQ_PSN</code>

The types of QP transport service for the `IBV_QPT_UC` type, follow:

Next state	Required attributes
Init	<code>IBV_QP_STATE</code> , <code>IBV_QP_PKEY_INDEX</code> , <code>IBV_QP_PORT</code> , <code>IBV_QP_ACCESS_FLAGS</code>
RTR	<code>IBV_QP_STATE</code> , <code>IBV_QP_AV</code> , <code>IBV_QP_PATH_MTU</code> , <code>IBV_QP_DEST_QPN</code> , <code>IBV_QP_RQ_PSN</code>
RTS	<code>IBV_QP_STATE</code> , <code>IBV_QP_SQ_PSN</code>

The types of QP transport service for the `IBV_QPT_RC` type, follow:

Next state	Required attributes
Init	<code>IBV_QP_STATE</code> , <code>IBV_QP_PKEY_INDEX</code> , <code>IBV_QP_PORT</code> , <code>IBV_QP_ACCESS_FLAGS</code>
RTR	<code>IBV_QP_STATE</code> , <code>IBV_QP_AV</code> , <code>IBV_QP_PATH_MTU</code> , <code>IBV_QP_DEST_QPN</code> , <code>IBV_QP_RQ_PSN</code> , <code>IBV_QP_MAX_DEST_RD_ATOMIC</code> , <code>IBV_QP_MIN_RNR_TIMER</code>
RTS	<code>IBV_QP_STATE</code> , <code>IBV_QP_SQ_PSN</code> , <code>IBV_QP_MAX_QP_RD_ATOMIC</code> , <code>IBV_QP_RETRY_CNT</code> , <code>IBV_QP_RNR_RETRY</code> , <code>IBV_QP_TIMEOUT</code>

The types of QP transport service for the `IBV_QPT_RAW_PACKET` type, follow:

Next state	Required attributes
Init	<code>IBV_QP_STATE</code> , <code>IBV_QP_PORT</code>
RTR	<code>IBV_QP_STATE</code>
RTS	<code>IBV_QP_STATE</code>

Indicates the QP transport service types:

---

### Next state Required attributes

Init	<code>IBV_QP_STATE</code> , <code>IBV_QP_PKEY_INDEX</code> , <code>IBV_QP_PORT</code> , <code>IBV_QP_ACCESS_FLAGS</code>
RTR	<code>IBV_QP_STATE</code> , <code>IBV_QP_AV</code> , <code>IBV_QP_PATH_MTU</code> , <code>IBV_QP_DEST_QPN</code> , <code>IBV_QP_RQ_PSN</code> , <code>IBV_QP_MAX_DEST_RD_ATOMIC</code> , <code>IBV_QP_MIN_RNR_TIMER</code>
RTS	<code>IBV_QP_STATE</code> , <code>IBV_QP_SQ_PSN</code> , <code>IBV_QP_MAX_QP_RD_ATOMIC</code> , <code>IBV_QP_RETRY_CNT</code> , <code>IBV_QP_RNR_RETRY</code> , <code>IBV_QP_TIMEOUT</code>

## Input Parameters

### Item Descriptor

<i>qp</i>	Specifies the <code>ibv_qp</code> struct for the <code>ibv_create_qp</code> function.
<i>attr</i>	Specifies the QP attributes.

Item	Descriptor
------	------------

<i>attr_mask</i>	Specifies the bit mask. The bit mask defines the attributes within the <i>attr</i> parameter that is set for a call.
------------------	--

## Return Values

Item	Descriptor
------	------------

<b>0</b>	On success.
----------	-------------

<b>errno</b>	On failure.
--------------	-------------

## *ibv\_post\_recv*

Posts a list of work requests (WRs) to a receive queue.

## Syntax

```
#include <rdma/verbs.h>
int ibv_post_recv(struct ibv_qp *qp, struct ibv_recv_wr *wr, struct ibv_recv_wr **bad_wr)
```

## Description

The **ibv\_post\_recv()** function posts the linked list of work requests (WRs) starting with the *wr* parameter to the receive queue of the queue pair. The function stops processing WRs from the list at the first failure that can be detected immediately while requests are being posted, and returns the failing WR through the *bad\_wr* parameter.

The *wr* argument is an *ibv\_recv\_wr* struct, as defined in the `<rdma/verbs.h>` file.

```
struct ibv_recv_wr {
    uint64_t                wr_id;           /* User defined WR ID */
    struct ibv_recv_wr     *next;          /* Pointer to next WR in list, NULL if last
WR */
    struct ibv_sge         *sg_list;      /* Pointer to the scatter-gather array */
    int                    num_sge;      /* Size of the scatter-gather array
*/
};

struct ibv_sge {
    uint64_t                addr;          /* Start address of the local memory
buffer */
    uint32_t                length;       /* Length of the buffer */
    uint32_t                lkey;        /* Key of the local memory region */
};
```

**Note:** The buffers that is used by a WR can be safely reused after the request is completed, and a work completion is retrieved from the corresponding completion queue (CQ).

## Input Parameters

Item	Descriptor
------	------------

<i>qp</i>	Specifies the <i>ibv_qp</i> struct for the <b>ibv_create_qp</b> function.
-----------	---

<i>wr</i>	Specifies the first work request (WR) that contains the receive buffers.
-----------	--

## Output Parameter

Item	Descriptor
------	------------

<i>bad_wr</i>	Specifies the pointer to the first rejected WR.
---------------	---

## Return Values

Item	Descriptor
0	On success.
errno	On failure.

## ***ibv\_post\_send***

Posts a list of work requests (WR) to a send queue.

## Syntax

```
#include <rdma/verbs.h>
int ibv_post_send(struct ibv_qp *qp, struct ibv_send_wr *wr, struct ibv_send_wr **bad_wr)
```

## Description

The **ibv\_post\_send()** function posts the linked list of work requests (WR) starting with the *wr* parameter to the send queue of the queue pair *qp*. The function stops processing the WRs from the list after detecting the first failure while requests are being posted, and returns the failing WR by using the *bad\_wr* parameter.

The *wr* argument is an `ibv_send_wr` struct that is defined in the `<rdma/verbs.h>` file.

The transport service types for the operation codes that RC supports, follow:

OPCODE	IBV_QPT_RC
IBV_WR_SEND	Supported
IBV_WR_SEND_WITH_IMM	Supported
IBV_WR_RDMA_WRITE	Supported
IBV_WR_RDMA_WRITE_WITH_I MM	Supported
IBV_WR_RDMA_READ	Supported
IBV_WR_ATOMIC_CMP_AND_S WP	Not supported
IBV_WR_ATOMIC_FETCH_AND_ ADD	Not supported

The attribute `send_flags` describes the properties of the WR. It is either 0 or the bitwise OR of one or more of the following flags:

### **IBV\_SEND\_FENCE**

Sets the fence indicator. The `IBV_SEND_FENCE` flag is valid only for QPs with the transport service type `IBV_QPT_RC`.

### **IBV\_SEND\_SIGNALED**

Sets the completion notification indicator. The `IBV_SEND_SIGNALED` flag is relevant only if QP is created with the `sq_sig_all` parameter equal to 0.

### **IBV\_SEND\_SOLICITED**

Sets the solicited event indicator. The `IBV_SEND_SOLICITED` flag is valid only for send and remote device memory access (RDMA) write functions with immediate effect.

### **IBV\_SEND\_INLINE**

Sends data in given gather list as inline data in a send WQE. The `IBV_SEND_INLINE` flag is valid only for send and RDMA write functions. The `L_Key` parameter is not verified.

**Note:** The buffers used by a WR can be safely reused after the request is complete. A work completion is retrieved from the corresponding completion queue (CQ).

## Input Parameters

Item	Descriptor
<i>qp</i>	Specifies the <code>ibv_qp</code> struct for the <code>ibv_create_qp</code> function.
<i>wr</i>	Specifies the first work request (WR).

## Output Parameter

Item	Descriptor
<i>bad_wr</i>	Specifies the pointer to the first rejected WR.

## Return Values

Item	Descriptor
0	On success.
<code>errno</code>	On failure.

## *ibv\_query\_qp*

Gets the attributes of a queue pair (QP).

## Syntax

```
#include <rdma/verbs.h>
int ibv_query_qp(struct ibv_qp *qp, struct ibv_qp_attr *attr,
                int attr_mask,
                struct ibv_qp_init_attr *init_attr);
```

## Description

The `ibv_query_qp()` gets the attributes specified in `attr_mask` for the QP and returns them through the pointers `attr` and `init_attr`. The argument `attr` is an `ibv_qp_attr` struct, as defined in `<rdma/verbs.h>`.

```
struct ibv_qp_attr {
enum ibv_qp_state qp_state; /* Current QP state */
enum ibv_qp_state cur_qp_state; /* Current QP state - irrelevant for ibv_query_qp */
enum ibv_mtu path_mtu; /* Path MTU (valid only for RC/UC QPs) */
enum ibv_mig_state path_mig_state; /* Path migration state (valid if HCA supports APM) */
uint32_t qkey; /* Q_Key of the QP (valid only for UD QPs) */
uint32_t rq_psn; /* PSN for receive queue (valid only for RC/UC QPs) */
uint32_t sq_psn; /* PSN for send queue (valid only for RC/UC QPs) */
uint32_t dest_qp_num; /* Destination QP number (valid only for RC/UC QPs) */
int qp_access_flags; /* Mask of enabled remote access operations (valid only for RC/UC QPs) */

struct ibv_qp_cap cap; /* QP capabilities */
struct ibv_ah_attr ah_attr; /* Primary path address vector (valid only for RC/UC QPs) */
struct ibv_ah_attr alt_ah_attr; /* Alternate path address vector (valid only for RC/UC QPs) */
uint16_t pkey_index; /* Primary P_Key index */
uint16_t alt_pkey_index; /* Alternate P_Key index */
uint8_t en_sqd_async_notify; /* Enable SQD.drained async notification - irrelevant for ibv_query_qp */

uint8_t sq_draining; /* Is the QP draining? (Valid only if qp_state is SQD) */
uint8_t max_rd_atomic; /* Number of outstanding RDMA reads & atomic operations on the destination QP (valid only for RC QPs) */
uint8_t max_dest_rd_atomic; /* Number of responder resources for handling incoming RDMA reads & atomic operations (valid only for RC QPs) */

uint8_t min_rnr_timer; /* Minimum RNR NAK timer (valid only for RC QPs) */
uint8_t port_num; /* Primary port number */
uint8_t timeout; /* Local ack timeout for primary path (valid only for RC QPs) */

uint8_t retry_cnt; /* Retry count (valid only for RC QPs) */
uint8_t rnr_retry; /* RNR retry (valid only for RC QPs) */
uint8_t alt_port_num; /* Alternate port number */
uint8_t alt_timeout; /* Local ack timeout for alternate path (valid only for RC QPs) */
};
```

For details on struct **ibv\_qp\_cap()**, see the description of **ibv\_create\_qp** function. For details on struct **ibv\_ah\_attr**, see the description of **ibv\_create\_ah()** function.

## Return Values

On success, the **ibv\_query\_qp()** function returns 0, or the **errno** on failure that indicates the reason for failure.

## **ibv\_attach\_mcast**

Attaches and detaches a queue pair (QPs) to or from a multicast group

## Syntax

```
#include <rdma/verbs.h>
int ibv_attach_mcast(struct ibv_qp *qp, const union ibv_gid *gid,
                    uint16_t lid);
int ibv_detach_mcast(struct ibv_qp *qp, const union ibv_gid *gid,
                    uint16_t lid);
```

## Description

The **ibv\_attach\_mcast** function attaches the queue pair (QP) to the multicast group that has the MGID gid and MLID lid. The **ibv\_detach\_mcast** function detaches the QP to the multicast group that has the MGID gid and MLID lid.

### Note:

- QPs of Transport Service Type IBV\_QPT\_UD or IBV\_QPT\_RAW\_PACKET can be attached to multicast groups.
- If a QP is attached to the same multicast group multiple times, the QP receives a single copy of a multicast message.
- To receive multicast messages, a join request for the multicast group must be sent to the subnet administrator (SA). The fabric's multicast routing is configured on receiving the join request to deliver messages to the local port.

## Return Values

### 0

The **ibv\_attach\_mcast** and **ibv\_detach\_mcast** functions returns 0 on success.

### errno

The **ibv\_attach\_mcast** and **ibv\_detach\_mcast** functions returns 0 on failure. **errno** also specifies the reason for failure.

## Examples

To use **ibv\_attach\_mcast** function with RAW ETH QP, use the following program:

```
union ibv_gid mgid;

memset(&mgid, 0, sizeof(union ibv_gid));
memcpy(&mgid.raw[10], mmac, 6);
if (ibv_attach_mcast(qp, &mgid, 0)) {
    printf ("Failed to attach qp to mcast. Errno: %d\n",errno);
    return 1;
}
```

## Completion queue management

Lists the functions that is used to manage the completion queue for the `libibverbs` library.

### ***ibv\_create\_cq***, ***ibv\_destroy\_cq***

Creates or destroys a completion queue (CQ).

### Syntax

```
#include <rdma/verbs.h>
struct ibv_cq *ibv_create_cq(struct ibv_context *context, int cqe, void *cq_context,
struct ibv_comp_channel *channel, int comp_vector)
int ibv_destroy_cq(struct ibv_cq *cq)
```

### Description

The **`ibv_create_cq()`** function creates a completion queue (CQ). A completion queue holds completion queue events (CQE). Each queue pair (QP) has an associated send and receive CQ. A single CQ can be shared for sending, receiving, and sharing across multiple QPs.

The *cqe* parameter defines the minimum size of the queue. The actual size of the queue might be larger than the specified value.

The *cq\_context* parameter is a user-defined value. If the value is specified during CQ creation, this value is returned as a parameter in the **`ibv_get_cq_event()`** function when using a completion channel (CC).

The *channel* parameter is used to specify a CC. A CQ is merely a queue that does not have a built-in notification mechanism. When using a polling paradigm for CQ processing, a CC is not required. Poll the CQ at regular intervals. However, if you want to use a pend paradigm, a CC is required. The CC is a mechanism that allows the user to be notified that a new CQE is on the CQ.

The CQ uses the **`comp_vector`** parameter for signaling completion events. It must be at least zero and less than the `context->num_comp_vectors` parameter.

The **`ibv_destroy_cq()`** function destroys the CQ *cq*.

#### Notes:

- The **`ibv_create_cq()`** function can create a CQ with a size greater than or equal to the requested size. You can determine the actual size of the function from the *cqe* attribute in the returned CQ.
- The **`ibv_destroy_cq()`** function fails if any queue pair is still associated with the CQ.

### Parameters

Item	Descriptor
<i>context</i>	The <b><code>ibv_context</code></b> struct for the <b><code>ibv_open_device()</code></b> function.
<i>cqe</i>	Minimum number of entries that CQ supports.
<i>cq_context</i>	(Optional) Specifies a user-defined value that is returned with completion events.
<i>channel</i>	(Optional) Specifies the completion channel.
<i>comp_vector</i>	(Optional) Specifies the completion vector.

### Return Value

The **`ibv_create_cq()`** function returns a pointer to the CQ, or NULL if the request fails.

The **`ibv_destroy_cq()`** function returns 0 on success, or the value **`errno`** on failure, which indicates the reason for failure.

## ***ibv\_req\_notify\_cq***

Requests the completion notification on a completion queue (CQ).

### **Syntax**

```
#include <rdma/verbs.h>
int ibv_req_notify_cq(struct ibv_cq *cq, int solicited_only);
```

### **Description**

The ***ibv\_req\_notify\_cq()*** function requests a completion notification on the completion queue (CQ) *cq* parameter.

When a new CQ entry (CQE) is added to a *cq* parameter, a completion event is added to the completion channel that is associated with the CQ. If the *solicited\_only* argument is zero, a completion event is generated for any new CQE. If *solicited\_only* parameter is nonzero, an event is generated for a new CQE that is considered solicited. A CQE is solicited if it receives completion for a message that has the solicited event header bit set, or if the status is not successful.

All other successful receive completions or any successful send completion is unsolicited.

**Note:** The request for a notification is sent once. One completion event is generated for each call that is made to the ***ibv\_req\_notify\_cq()*** function.

### **Parameters**

<b>Item</b>	<b>Descriptor</b>
<i>cq</i>	Specifies the <i>ibv_cq</i> struct for the <b><i>ibv_create_cq</i></b> function.
<i>solicited_only</i>	Notifies only if the WR is flagged as solicited.

### **Return Values**

<b>Item</b>	<b>Descriptor</b>
0	On success.
errno	On failure.

## ***ibv\_poll\_cq***

Polls a completion queue (CQ).

### **Syntax**

```
#include <rdma/verbs.h>
int ibv_poll_cq(struct ibv_cq *cq, int num_entries, struct ibv_wc *wc)
```

### **Description**

The ***ibv\_poll\_cq()*** function polls the change queue (CQ) for work completions and returns the first *num\_entries* parameter with completions (or all available completions if the CQ contains less than this number) in the *wc* array. The *wc* argument is a pointer to an array of ***ibv\_wc*** struct that is defined in the `<rdma/verbs.h>` file.

```
struct ibv_wc {
uint64_t          wr_id;          /* ID of the completed Work Request (WR) */
enum ibv_wc_status status;      /* Status of the operation */
enum ibv_wc_opcode opcode;      /* Operation type specified in the completed WR */
uint32_t          vendor_err;   /* Vendor error syndrome */
uint32_t          byte_len;     /* Number of bytes transferred */
uint32_t          imm_data;     /* Immediate data (in network byte order) */
uint32_t          qp_num;       /* Local QP number of completed WR */
uint32_t          src_qp;       /* Source QP number (remote QP number) of completed
```

```

int          wc_flags;      /* WR (valid only for UD QPs) */
uint16_t    pkey_index;    /* Flags of the completed WR */
uint16_t    slid;         /* P_Key index (valid only for GSI QPs) */
uint8_t     sl;          /* Source LID */
uint8_t     dlid_path_bits; /* Service Level */
uint8_t     dlid_path_bits; /* DLID path bits (not applicable for multicast
                             messages) */
};

```

The **wc\_flags** attribute describes the properties of the work completion. The flag is either 0 or the bitwise OR of one or more of the following flags:

#### **IBV\_WC\_GRH**

GRH is present.

#### **IBV\_WC\_WITH\_IMM**

Immediate data value is valid.

Not all **wc** attributes are always valid. If the completion status is other than **IBV\_WC\_SUCCESS**, only the **wr\_id**, **status**, **qp\_num**, and **vendor\_err** attributes are valid.

**Note:** Each polled completion is removed from the CQ and cannot be returned to it. You must consume work completions at a rate that prevents a CQ overrun from occurrence. In a CQ overrun, the asynchronous **IBV\_EVENT\_CQ\_ERR** event is triggered, and the CQ cannot be used.

## Input Parameters

Item	Descriptor
<i>cq</i>	Specifies the <i>ibv_cq</i> struct from the <b>ibv_create_cq</b> function.
<i>num_entries</i>	Specifies the maximum number of completion queue entries (CQE) to return.

## Output Parameters

Item	Descriptor
<i>wc</i>	Specifies the CQE array.

## Return Values

On success, the **ibv\_poll\_cq()** function returns a non-negative value equal to the number of completions found. On failure, a negative value is returned.

### **ibv\_get\_cq\_event, ibv\_ack\_cq\_events**

Gets and acknowledges the completion queue (CQ) events.

## Syntax

```

#include <rdma/verbs.h>
int ibv_get_cq_event(struct ibv_comp_channel *channel, struct ibv_cq **cq, void **cq_context);
void ibv_ack_cq_events(struct ibv_cq *cq, unsigned int nevents);

```

## Description

The **ibv\_get\_cq\_event()** function waits for the next completion event in the completion event channel. The *cq* argument is used to return the CQ that caused the event and the *cq\_context* parameter is used to return the context of the CQ.

The **ibv\_ack\_cq\_events()** function acknowledges the *nevents* events on the CQ *cq* parameter.

### Notes:

- All completion events that the **ibv\_get\_cq\_event()** function returns must be acknowledged by using the **ibv\_ack\_cq\_events()** function.
- To avoid competition, when you destroy a CQ, the CQ waits for the completion of the events. This action guarantees a one-to-one correspondence between acknowledgements and successful get operation.
- When you call the **ibv\_ack\_cq\_events()** function, it is expensive in the datapath because it must take a mutex. To reduce the cost, a count of the number of events requesting acknowledgement and acknowledging several completion events in one call to the **ibv\_ack\_cq\_events()** function are performed.

## Input Parameters

Item	Descriptor
<i>channel</i>	The <b>ibv_comp_channel</b> struct for the <b>ibv_create_comp_channel()</b> function.

## Output Parameters

Item	Descriptor
<i>cq</i>	A pointer to the completion queue (CQ) that is associated with the event.
<i>cq_context</i>	The user-supplied context that is set in the <b>ibv_create_cq()</b> function.

## Return Value

The **ibv\_get\_cq\_event()** and **ibv\_ack\_cq\_events()** functions return 0 on success, and -1 if the request fails.

## Examples

1. The following code example demonstrates one possible way to work with completion events. It performs the following steps:
  - a. Preparation:
    - i) Creates a CQ.
    - ii) Requests notification after creation of a new (first) completion event.
  - b. Completion handling routine:
    - i) Waits for the completion event and acknowledges it.
    - ii) Requests notification for the next completion event.
    - iii) Empties the CQ.

**Note:** An extra event can be triggered without having a corresponding completion entry in the CQ. This occurs if a completion entry is added to the CQ between requesting for notification and emptying the CQ. Then, the CQ is emptied.

```

cq = ibv_create_cq(ctx, 1, ev_ctx, channel, 0);
if (!cq) {
    fprintf(stderr, "Failed to create CQ\n");
    return 1;
}

/* Request notification before any completion can be created */
if (ibv_req_notify_cq(cq, 0)) {
    fprintf(stderr, "Could not request CQ notification\n");
    return 1;
}

```

```

.
.
.
/* Wait for the completion event */
if (ibv_get_cq_event(channel, &ev_cq, &ev_ctx)) {
    fprintf(stderr, "Failed to get cq_event\n");
    return 1;
}

/* Ack the event */
ibv_ack_cq_events(ev_cq, 1);

/* Request notification upon the next completion event */
if (ibv_req_notify_cq(cq, 0)) {
    fprintf(stderr, "Could not request CQ notification\n");
    return 1;
}

/* Empty the CQ: poll all of the completions from the CQ (if any exist) */
do {
    ne = ibv_poll_cq(cq, 1, &wc);
    if (ne < 0) {
        fprintf(stderr, "Failed to poll completions from the CQ\n");
        return 1;
    }
    if (wc.status != IBV_WC_SUCCESS) {
        fprintf(stderr, "Completion with status 0x%x was found\n",
wc.status);
        return 1;
    }
} while (ne);

```

2. The following code example demonstrates a possible way to work with completion events in nonblocking mode. The code performs the following steps:
  - a. Sets the completion event channel in nonblocked mode.
  - b. Polls the channel until it has a completion event.
  - c. Gets the completion event and acknowledges it.

```

/* change the blocking mode of the completion channel */
flags = fcntl(channel->fd, F_GETFL);
rc = fcntl(channel->fd, F_SETFL, flags | O_NONBLOCK);
if (rc < 0) {
    fprintf(stderr, "Failed to change file descriptor of completion event channel\n");
    return 1;
}
/*
* poll the channel until it has an event and sleep ms_timeout
* milliseconds between any iteration
*/
my_pollfd.fd = channel->fd;
my_pollfd.events = POLLIN;
my_pollfd.revents = 0;

do {
    rc = poll(&my_pollfd, 1, ms_timeout);
    } while (rc == 0);
    if (rc &lt; 0){ fprintf(stderr, "poll failed\n");
    return 1;
    }
    ev_cq = cq;
    /* Wait for the completion event */
    if (ibv_get_cq_event(channel, &ev_cq, &ev_ctx)) {
        fprintf(stderr, "Failed to get cq_event\n");
        return 1;
    }
    /* Ack the event */
    ibv_ack_cq_events(ev_cq, 1);

```

## Protection domain management

Lists the functions to be used for managing a protection domain for the `libibverbs` library.

### ***ibv\_alloc\_pd, ibv\_dealloc\_pd***

Allocates or deallocates a protection domain (PD).

### Syntax

```
#include <rdma/verbs.h>
struct ibv_pd *ibv_alloc_pd(struct ibv_context *context)
int ibv_dealloc_pd(struct ibv_pd *pd)
```

### Description

The **`ibv_alloc_pd()`** function allocates a PD for the remote device memory access (RDMA) device context, the *context* parameter. The **`ibv_dealloc_pd()`** function deallocates PD, the *pd* parameter.

**Note:** The **`ibv_dealloc_pd()`** function fails if any other RDMA resource is still associated with the PD that must be freed.

### Parameters

Item	Descriptor
<i>context</i>	The <b><code>ibv_context</code></b> struct for the <b><code>ibv_open_device()</code></b> function.

### Return Value

The **`ibv_alloc_pd()`** function returns a pointer to the allocated PD, or NULL if the request fails. The **`ibv_dealloc_pd()`** function returns 0 on success, or the value of **`errno`** on failure (which indicates the reason for failure).

## Memory region management

Lists the functions to be used for memory region management for the `libibverbs` library.

### ***ibv\_reg\_mr***

Registers or releases a memory region (MR).

### Syntax

```
#include <rdma/verbs.h>
struct ibv_mr *ibv_reg_mr(struct ibv_pd *pd, void *addr, size_t length, int ibv_access_flags
access);
int ibv_dereg_mr(struct ibv_mr *mr);
```

### Description

The **`ibv_reg_mr()`** function registers a memory region (MR) that is associated with the protection domain, the *pd* parameter. The starting address of the MR is specified by using the *addr* parameter and its size is specified by using the *length* parameter. The *access* parameter describes the required memory protection attributes that are either 0 or the bitwise OR of one or more of the following flags:

#### **IBV\_ACCESS\_LOCAL\_WRITE**

Enables local write access

#### **IBV\_ACCESS\_REMOTE\_WRITE**

Enable remote write access

### **IBV\_ACCESS\_REMOTE\_READ**

Enable remote read access

### **IBV\_ACCESS\_REMOTE\_ATOMIC**

Enable remote atomic operation access (not supported)

### **IBV\_ACCESS\_MW\_BIND**

Enable memory window binding (not supported)

If the `IBV_ACCESS_REMOTE_WRITE` or `IBV_ACCESS_REMOTE_ATOMIC` flag is set, the `IBV_ACCESS_LOCAL_WRITE` flag must also be set.

**Note:** Local read access is always enabled for the MR.

The `ibv_dereg_mr()` function release the MR.

## **Parameters**

<b>Item</b>	<b>Descriptor</b>
<i>pd</i>	Specifies the <code>ibv_pd</code> struct for the <code>ibv_alloc_pd()</code> function.
<i>addr</i>	Specifies the memory base address.
<i>length</i>	Specifies the length of memory region in bytes.
<i>access</i>	Specifies the access flags.

## **Return Values**

The `ibv_reg_mr()` function returns a pointer to the registered MR on success, and NULL if the request fails. The local key (L\_Key) *lkey* field is used by the `ibv_sge` struct when posting buffers with `ibv_post_*` verbs, and the remote key (R\_Key) *rkey* field is used by remote processes to run the remote device memory access (RDMA) operations. The remote process places the *rkey* field in the `ibv_send_wr` struct that is sent to the `ibv_post_send()` function.

The `ibv_dereg_mr()` function returns 0 on success, and the value of `errno` on failure, which indicates the reason for failure.

## **Event Management**

Lists the functions that is used to manage an event for the `libibverbs` library.

### ***ibv\_create\_comp\_channel, ibv\_destroy\_comp\_channel***

Creates or destroys a completion event channel.

## **Syntax**

```
#include <rdma/verbs.h>
struct ibv_comp_channel *ibv_create_comp_channel(struct ibv_context *context)
int ibv_destroy_comp_channel(struct ibv_comp_channel *channel)
```

## **Description**

The `ibv_create_comp_channel()` function creates a completion event channel for the remote direct memory access (RDMA) device context, the *context* parameter. A completion channel is a mechanism to receive notifications when a new completion queue event (CQE) is placed on a completion queue (CQ).

The `ibv_destroy_comp_channel()` function destroys the completion event channel.

### **Notes:**

- A **completion channel** is an abstraction introduced by the `libibverbs` library that does not exist in the InfiniBand architecture verbs specification. A completion channel is essentially a file descriptor that is

used to deliver completion notifications to a userspace process. When a completion event is generated for a completion queue (CQ), the event is delivered through the completion channel attached to that CQ. This process might be useful to send completion events to different threads by using multiple completion channels.

- The **ibv\_destroy\_comp\_channel()** function fails if any CQs are still associated with the completion event channel that is being destroyed.

## Parameters

Item	Descriptor
<i>context</i>	The <b>ibv_context</b> struct for the <b>ibv_open_device()</b> function.

## Return Value

The **ibv\_create\_comp\_channel()** function returns a pointer to the created completion event channel, or NULL if the request fails.

The **ibv\_destroy\_comp\_channel()** function returns 0 on success, or the value of **errno** on failure (which indicates the reason for failure).

## **ibv\_get\_async\_event, ibv\_ack\_async\_event**

Gets or acknowledges the asynchronous events.

## Syntax

```
#include <rdma/verbs.h>
int ibv_get_async_event(struct ibv_context *context,
struct ibv_async_event *event); void ibv_ack_async_event
(struct ibv_async_event *event);
```

## Description

The **ibv\_get\_async\_event()** function waits for the next async event of the remote direct memory access (RDMA) device context and returns it through the *event* pointer, which is an **ibv\_async\_event** struct, as defined in the `<rdma/verbs.h>` file.

```
struct ibv_async_event {
    union {
        struct ibv_cq      *cq;           /* CQ that got the event */
        struct ibv_qp      *qp;           /* QP that got the event */
        struct ibv_srq      *srq;        /* SRQ that got the event(Not Supported)*/
        int                 port_num;    /* port number that got the event */
    } element;
    enum ibv_event_type     event_type;  /* type of the event */
};
```

The **ibv\_create\_qp()** function updates the *qp\_init\_attr* parameter in the **cap** struct with the actual QP values of the QP that was created. The values are greater than or equal to the values requested. The **ibv\_destroy\_qp()** function destroys the QP by using the *qp* parameter.

One member of the element union is valid, depending on the **event\_type** member of the structure. The **event\_type** member can be one of the following events:

Item	Descriptor
<i>QP events</i>	
<b>IBV_EVENT_QP_FATAL</b>	Error occurred on a QP and it transitions to error state.
<b>IBV_EVENT_QP_REQ_ERR</b>	Invalid request that causes a local work queue error.
<b>IBV_EVENT_QP_ACCESS_ERR</b>	Local access violation error.
<b>IBV_EVENT_COMM_EST</b>	Communication is established on a QP.

Item	Descriptor
<b>IBV_EVENT_SQ_DRAINED</b>	Send queue is drained of outstanding messages in progress.
<b>IBV_EVENT_PATH_MIG</b>	A connection is moved to an alternative path.
<b>IBV_EVENT_PATH_MIG_ERR</b>	A connection failed to moved to the alternative path.
<i>CQ events</i>	
<b>IBV_EVENT_CQ_ERR</b>	CQ is in error (CQ overrun).
<i>Port events</i>	
<b>IBV_EVENT_PORT_ACTIVE</b>	Link became active on a port.
<b>IBV_EVENT_PORT_ERR</b>	Link became unavailable on a port.
<b>IBV_EVENT_LID_CHANGE</b>	Link ID (LID) is changed on a port.
<b>IBV_EVENT_PKEY_CHANGE</b>	The <b>P_Key</b> table is changed on a port.
<i>CA events</i>	
<b>IBV_EVENT_DEVICE_FATAL</b>	CA is in <b>FATAL</b> state.

The **ibv\_ack\_async\_event()** function acknowledges the asynchronous event.

#### Notes:

- All asynchronous events that the **ibv\_get\_async\_event()** function returns must be acknowledged by using the **ibv\_ack\_async\_event()** event. To avoid competition, destroying an object (CQ or QP) waits for all affiliated events for the object to be acknowledged. This process avoids an application from retrieving an affiliated event after the corresponding object is destroyed.
- The **ibv\_get\_async\_event()** function is a blocking function. If multiple threads call this function simultaneously, then when an async event occurs, only one thread will receive this function. It is not possible to predict the thread that receives the function.

## Input Data

Item	Descriptor
<code>struct ibv_context *context</code>	The <b>ibv_context</b> struct for the <b>ibv_open_device</b> function.
<code>struct ibv_async_event *event</code>	The event pointer.

## Return Value

The **ibv\_get\_async\_event()** function returns 0 on success, and -1 if the request fails.

The **ibv\_ack\_async\_event()** function returns no value.

## Example

The following code example demonstrates one possible way to work with async events in nonblocking mode. The event executes the following steps:

1. Sets the async events queue in the nonblocked work mode.
2. Polls the queue until it has an asynchronous event.
3. Gets the asynchronous event and acknowledges it.

```

/* change the blocking mode of the async event queue */
flags = fcntl(ctx->async_fd, F_GETFL);
rc = fcntl(ctx->async_fd, F_SETFL, flags | O_NONBLOCK);
if (rc < 0) {
    fprintf(stderr, "Failed to change file descriptor of async event queue\n");
    return 1;
}
/*
 * poll the queue until it has an event and sleep ms_timeout
 * milliseconds between any iteration

```

```

    */
    my_pollfd.fd      = ctx->async_fd;
    my_pollfd.events  = POLLIN;
    my_pollfd.revents = 0;

    do {
        rc = poll(&my_pollfd;,1, ms_timeout);
    } while (rc == 0);
    if (rc < 0) {
        fprintf(stderr, "poll failed\n");
        return 1;
    }

    /* Get the async event */
    if (ibv_get_async_event(ctx, &async_event)) {
        fprintf(stderr, "Failed to get async_event\n");
        return 1;
    }
    /* Ack the event */
    ibv_ack_async_event(&async_event);

```

## ***ibv\_event\_type\_str()***

Returns the string that describes the **event\_type**, **node\_type**, and **port\_state** enum values.

## **Syntax**

```

const char *ibv_event_type_str(enum ibv_event_type event_type);
const char *ibv_node_type_str(enum ibv_node_type node_type);
const char *ibv_port_state_str(enum ibv_port_state port_state);

```

## **Description**

The **ibv\_node\_type\_str()** function returns a string that describes the *node\_type* enum value.

The **ibv\_port\_state\_str()** function returns a string that describes the *port\_state* enum value.

The **ibv\_event\_type\_str()** function returns a string that describes the *event\_type* enum value.

## **Return Value**

The **ibv\_node\_type\_str()**, **ibv\_port\_state\_str()**, and **ibv\_event\_type\_str()** functions return a constant string that describes the enum value passed as their argument.

The <<unknown>> string is passed if the enum value is not known.

## **Verbs not supported by the libibverbs library**

You can find the list of verbs that are not supported by the **libibverbs** library.

<b>Item</b>	<b>Descriptor</b>
<b>Shared Receive Queues (SRQ)</b>	
<b>ibv_create_srq</b>	Creates a shared receive queue.
<b>ibv_modify_srq</b>	Modifies attributes of a shared receive queue.
<b>ibv_query_srq</b>	Gets the attributes of a shared receive queue.
<b>ibv_destroy_srq</b>	Destroys a shared receive queue.
<b>ibv_post_srq_recv</b>	Posts a list of work requests to a shared receive queue.
<b>Extended Reliable Connection (XRC)</b>	
<b>ibv_create_xrc_srq</b>	Creates an XRC shared receive queue
<b>ibv_open_xrc_domain</b>	Opens an eXtended Reliable Connection domain.

<b>Item</b>	<b>Descriptor</b>
<b>ibv_close_xrc_domain</b>	Closes an eXtended Reliable Connection domain.
<b>ibv_create_xrc_rcv_qp</b>	Creates an XRC queue pair for serving as a receive-side only queue pair (QP).
<b>ibv_modify_xrc_rcv_qp</b>	Modifies the attributes of an XRC receive QP.
<b>ibv_query_xrc_rcv_qp</b>	Gets the attributes of an XRC receive QP.
<b>ibv_reg_xrc_rcv_qp</b>	Registers a user process with an XRC receive QP.
<b>ibv_unreg_xrc_rcv_qp</b>	Unregister a user process with an XRC receive QP.
<b>ibv_fork_init</b>	Initializes the libibverbs library to support the fork () function.

## Notices

---

This information was developed for products and services offered in the US.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing  
IBM Corporation  
North Castle Drive, MD-NC119  
Armonk, NY 10504-1785  
US*

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

*Intellectual Property Licensing  
Legal and Intellectual Property Law  
IBM Japan Ltd.  
19-21, Nihonbashi-Hakozakicho, Chuo-ku  
Tokyo 103-8510, Japan*

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

*IBM Director of Licensing  
IBM Corporation  
North Castle Drive, MD-NC119  
Armonk, NY 10504-1785  
US*

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

#### COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (your company name) (year).

Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. \_enter the year or years\_.

## Privacy policy considerations

---

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as the customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at <http://www.ibm.com/privacy> and IBM's Online Privacy Statement at <http://www.ibm.com/privacy/details> the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at <http://www.ibm.com/software/info/product-privacy>.

## Trademarks

---

IBM, the IBM logo, and [ibm.com](http://www.ibm.com) are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at [Copyright and trademark information at www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml).

UNIX is a registered trademark of The Open Group in the United States and other countries.



# Index

## Special Characters

- [\\_](#) [69](#)
- [\\_getlong](#) subroutine [69](#)
- [\\_getshort](#) subroutine [70](#)
- [\\_ll\\_log](#) subroutine [47](#)
- [\\_putlong](#) subroutine [71](#)
- [\\_putshort](#) [72](#)
- [\\_putshort](#) subroutine [72](#)
- [/etc/hosts](#) file
  - closing [88](#), [89](#)
  - opening [264](#), [265](#)
  - retrieving host entries [115](#), [116](#), [118](#), [120](#)
  - setting file markers [264](#), [265](#)
- [/etc/networks](#) file
  - closing [90–92](#)
  - opening [268](#), [269](#)
  - retrieving network entries [129](#), [132](#), [133](#)
  - setting file markers [268](#), [269](#)
- [/etc/protocols](#) file
  - closing [93](#)
  - opening [271](#), [272](#)
  - setting file markers [271](#), [272](#)
- [/etc/resolv.conf](#) file
  - retrieving host entries [115](#), [116](#), [118](#), [120](#)
  - searching for domain names [228](#)
  - searching for Internet addresses [228](#)
- [/etc/services](#) file
  - closing [94](#), [95](#)
  - opening [150](#), [272](#), [273](#)
  - reading [150](#)
  - retrieving service entries [146](#), [148](#)
  - setting file markers [272](#), [273](#)
- [/etc/socks5c.conf](#) File [304](#)

## A

- [a](#) [72](#)
- [accept](#) subroutine [72](#)
- [acknowledges](#) asynchronous events [379](#)
- [adjusting](#) the values of entries [54](#)
- [arp](#) subroutines
  - [arpresolve\\_common](#) [74](#)
  - [arpupdate](#) [76](#)
- [arpresolve\\_common](#) subroutine [74](#)
- [arpupdate](#) subroutine [76](#)
- [arrays](#)
  - translating into external representations [2](#), [3](#), [25](#)
- [ASCII](#) strings
  - converting to Internet addresses [183](#)
- [authentication](#) messages [13](#)
- [authentication](#) methods [262](#)

## B

- [b](#) [77](#)
- [bind](#) subroutine [77](#)

- [bind2addrsel](#) subroutine [79](#)
- [binds](#) RDMA identifier [328](#)
- [Booleans](#)
  - translating [2](#)
- [buffers](#)
  - checking for end of file [31](#)
- [byte](#) streams
  - placing long byte quantities [71](#)

## C

- [c](#) [80](#)
- [C](#) language, translating
  - [characters](#) [5](#)
  - [discriminated](#) unions [24](#)
  - [enumerations](#) [7](#)
  - [floats](#) [7](#)
  - [integers](#) [2](#), [11](#)
  - [long](#) integers [12](#)
  - [numbers](#) [27](#)
  - [short](#) integers [19](#)
  - [strings](#) [20](#), [26](#)
  - [unsigned](#) characters [21](#)
  - [unsigned](#) integers [21](#)
  - [unsigned](#) long integers [22](#), [23](#)
- [call](#) header messages [4](#)
- [call](#) messages [5](#)
- [clients](#)
  - [server](#) authentication [246](#)
- [closing](#) [92](#)
- [communicating](#) with the SNMP agent [62](#)
- [Communication](#) Manager (CM) ID operations
  - [rdma\\_bind\\_addr](#) [328](#)
  - [rdma\\_connect](#) [330](#)
  - [rdma\\_destroy\\_id](#) [327](#)
  - [rdma\\_resolve\\_addr](#) [329](#)
  - [rdma\\_resolve\\_route](#) [330](#)
- [communications](#) kernel service subroutines
  - [res\\_ninit](#) [231](#)
- [Completion](#) queue management [372](#)
- [Completion](#) Queue management
  - [completion](#) notification [373](#)
  - [completion](#) queue event [374](#)
  - [ibv\\_get\\_cq\\_event](#) [374](#)
  - [ibv\\_poll\\_cq](#) [373](#)
  - [ibv\\_req\\_notify\\_cq](#) [373](#)
  - [polls](#) a completion queue [373](#)
- [compressed](#) domain names
  - [expanding](#) [86](#)
- [connect](#) [80](#)
- [connect](#) subroutine [80](#)
- [connected](#) sockets
  - [creating](#) pairs [301](#)
  - [receiving](#) messages [221](#)
  - [sending](#) messages [250](#), [252](#)
- [connecting](#) [80](#)
- [Connection](#) Manager (CM)

- Connection Manager (CM) *(continued)*
  - ID operations [323](#)
- Connection Manager (CM) ID operations
  - [rdma\\_create\\_ep](#) 336
  - [rdma\\_destroy\\_ep](#) 337
  - [rdma\\_get\\_dst\\_port](#) 335
  - [rdma\\_get\\_local\\_addr](#) 335
  - [rdma\\_get\\_peer\\_addr](#) 335
  - [rdma\\_get\\_src\\_port](#) 334
  - [rdma\\_getaddrinfo](#) 337
  - [rdma\\_migrate\\_id](#) 328
  - [rdma\\_notify](#) 339
  - [rdma\\_reject](#) 333
- converter subroutines
  - [inet\\_net\\_ntop](#) 178
  - [inet\\_net\\_pton](#) 179
  - [inet\\_ntop](#) 184
  - [inet\\_pton](#) 185
- Create event channel
  - open channel [322](#)
- CreateIoCompletionPort Subroutine [83](#)
- creates a completion event channel [378](#)
- current domain names
  - returning [114](#)
  - setting [263](#)
- current host identifiers
  - retrieving [123](#)

## D

- d [84](#)
- data
  - marking outgoing as records [30](#)
- data streams
  - getting position of [9](#)
- Data transfer operations
  - [rdma\\_get\\_rcv\\_comp](#) 343
  - [rdma\\_get\\_request](#) 344
  - [rdma\\_get\\_send\\_comp](#) 345
  - [rdma\\_post\\_read](#) 345
  - [rdma\\_post\\_readv](#) 346
  - [rdma\\_post\\_rcv](#) 347
  - [rdma\\_post\\_rcvv](#) 348
  - [rdma\\_post\\_send](#) 348
  - [rdma\\_post\\_sendv](#) 349
  - [rdma\\_post\\_ud\\_send](#) 350
  - [rdma\\_post\\_write](#) 351
  - [rdma\\_post\\_writev](#) 352
- DCE principal mapping [200](#)
- default domains
  - getting [37](#)
  - searching names [228](#)
- Destroying event channel
  - closes event channel [322](#)
- destroys a completion event channel [378](#)
- Device management
  - attributes of an RDMA port [361](#)
  - [ibv\\_get\\_device\\_list](#) 358
  - [ibv\\_get\\_device\\_name](#) 359
  - [ibv\\_open\\_device](#) 360
  - [ibv\\_query\\_device](#) 360
  - [ibv\\_query\\_gid](#) 363
  - [ibv\\_query\\_pkey](#) 362
  - [ibv\\_query\\_port](#) 361

- Device management *(continued)*
  - Libibverbs library [358](#)
  - NIC MAC address [363](#)
  - P\_key table [362](#)
  - [rdma\\_free\\_devices](#) 354
  - [rdma\\_get\\_devices](#) 354
- Device Management [354](#)
- Device mangement
  - [ibv\\_get\\_device\\_guid](#) 359
- discriminated unions
  - translating [24](#)
- [dn\\_comp](#) subroutine [84](#)
- [dn\\_expand](#) subroutine [86](#)
- domain names
  - compressing [84](#)

## E

- e [87](#)
- [ebind](#) subroutine [87](#)
- encoding values from [50](#)
- [endhostent](#) subroutine [88](#)
- [endhostent\\_r](#) subroutine [89](#)
- ending SNMP communications [57](#)
- [endnetent](#) subroutine [90](#)
- [endnetent\\_r](#) subroutine [91](#)
- [endnetgrent](#) subroutine [186](#)
- [endnetgrent\\_r](#) subroutine [92](#)
- [endprotoent](#) [92](#)
- [endprotoent](#) subroutine [92](#)
- [endprotoent\\_r](#) subroutine [93](#)
- [endservent](#) subroutine [94](#)
- [endservent\\_r](#) subroutine [95](#)
- entries in the [134](#), [186](#)
- enum values [381](#)
- error codes
  - using as input to NIS subroutines [44](#)
- error strings
  - returning pointer [43](#)
- [ether\\_aton](#) subroutine [96](#)
- [ether\\_hostton](#) subroutine [96](#)
- [ether\\_line](#) subroutine [96](#)
- [ether\\_ntoa](#) subroutine [96](#)
- [ether\\_ntohost](#) subroutine [96](#)
- Event channel operations [322](#)
- Event handling operations [340](#)
- Event Handling Operations
  - RDMA CM event [343](#)
  - [rdma\\_ack\\_cm\\_event](#) 342
  - [rdma\\_event\\_str](#) 343
  - [rdma\\_get\\_cm\\_event](#) 340
- Event management
  - Libibverbs library [378](#)
- extending base subroutines [54](#)
- extending number of entries in [54](#)
- eXternal Data Representation [1](#)
- external representations, translating from
  - arrays [2](#), [3](#), [25](#)
  - Booleans [2](#)
  - C language characters [5](#), [21](#)
  - C language enumerations [7](#)
  - C language floats [7](#)
  - C language integers [11](#)
  - C language long integers [12](#)

external representations, translating from (*continued*)

- C language numbers [27](#)
- C language short integers [19](#)
- C language strings [20](#)
- C language unsigned integers [21](#)
- C language unsigned long integers [22](#)
- C language unsigned short integers [23](#)
- discriminated unions [24](#)
- opaque data [12](#)

## F

- f [97](#)
- FrcaCacheCreate subroutine [97](#)
- FrcaCacheDelete subroutine [98](#)
- FrcaCacheLoadFile Subroutine [99](#)
- FrcaCacheUnloadFile Subroutine [102](#)
- FrcaCtrlCreate Subroutine [103](#)
- FrcaCtrlDelete Subroutine [106](#)
- FrcaCtrlLog Subroutine [107](#)
- FrcaCtrlStart Subroutine [108](#)
- FrcaCtrlStop Subroutine [109](#)
- freeaddrinfo subroutine [110](#)
- freeing [59](#)
- from host byte order [157](#)

## G

- g [110](#)
- get\_auth\_method subroutine
  - authentication methods [113](#)
- getaddrinfo subroutine [110](#)
- getdomainname subroutine [114](#)
- gethostbyaddr subroutine [115](#)
- gethostbyaddr\_r subroutine [116](#)
- gethostbyname subroutine [118](#)
- gethostbyname\_r subroutine [120](#)
- gethostent [121](#)
- gethostent subroutine [121](#)
- gethostent\_r [122](#)
- gethostent\_r subroutine [122](#)
- gethostid subroutine [123](#)
- gethostname subroutine [123](#)
- getip4sourcefilter Subroutine [283](#)
- GetMultipleCompletionStatus Subroutine [124](#)
- getnameinfo subroutine [128](#)
- getnetbyaddr subroutine [129](#)
- getnetbyaddr\_r [130](#)
- getnetbyaddr\_r subroutine [130](#)
- getnetbyname [131](#)
- getnetbyname subroutine [131](#)
- getnetbyname\_r subroutine [132](#)
- getnetent subroutine [133](#)
- getnetent\_r [134](#)
- getnetent\_r subroutine [134](#)
- getnetgrent subroutine [186](#)
- getnetgrent\_r subroutine [134](#)
- getpeername subroutine [135](#)
- getprotobyname [137](#)
- getprotobyname subroutine [137](#)
- getprotobyname\_r subroutine [138](#)
- getprotobynumber [139](#)
- getprotobynumber subroutine [139](#)

- getprotobynumber\_r [140](#)
- getprotobynumber\_r subroutine [140](#)
- getprotoent subroutine [141](#)
- getprotoent\_r [142](#)
- getprotoent\_r subroutine [142](#)
- GetQueuedCompletionStatus Subroutine [143](#)
- gets asynchronous events [379](#)
- getservbyname [144](#)
- getservbyname subroutine [144](#)
- getservbyname\_r subroutine [146](#)
- getservbyport [147](#)
- getservbyport subroutine [147](#)
- getservbyport\_r [148](#)
- getservbyport\_r subroutine [148](#)
- getservent [149](#)
- getservent subroutine [149](#)
- getservent\_r subroutine [150](#)
- getsmuxEntrybyidentity subroutine [75](#)
- getsmuxEntrybyname subroutine [45](#)
- getsockname subroutine [151](#)
- getsockopt subroutine [152](#)
- getsourcefilter [283](#)
- group network
  - entries in the handling [270](#)

## H

- h [157](#)
- handling [134](#), [186](#)
- host machines
  - setting names [267](#)
  - setting unique identifiers [266](#)
- htonl [157](#)
- htonl subroutine [157](#)
- htonll [158](#)
- htonll subroutine [158](#)
- htons [159](#)
- htons subroutine [159](#)

## I

- i [160](#)
- I/O Completion Port (IOCP) Kernel Extension
  - CreateCompletionPort [83](#)
  - GetMultipleCompletionStatus [124](#)
  - GetQueuedCompletionStatus [143](#)
  - PostQueuedCompletionStatus [205](#)
  - ReadFile [220](#)
  - WriteFile [317](#)
- ibv\_ack\_async\_event [379](#)
- ibv\_attach\_mcast [371](#)
- ibv\_create\_comp\_channel [378](#)
- ibv\_create\_cq [372](#)
- ibv\_destroy\_comp\_channel [378](#)
- ibv\_destroy\_cq [372](#)
- ibv\_detach\_mcast [371](#)
- ibv\_event\_type\_str [381](#)
- ibv\_get\_async\_event [379](#)
- ibv\_reg\_mr [377](#)
- if\_freenameindex [160](#)
- if\_freenameindex subroutine [160](#)

- if\_indexname subroutine [160](#)
- if\_nameindex subroutine [161](#)
- if\_nameindex subroutine [162](#)
- incoming connections
  - limiting backlog [201](#)
- incoming messages alert [63](#)
- inet\_addr subroutine [174](#)
- inet\_anaof subroutine [176](#)
- inet\_makeaddr subroutine [177](#)
- inet\_net\_ntop subroutine [178](#)
- inet\_net\_pton subroutine [179](#)
- inet\_netof subroutine [180](#)
- inet\_network subroutine [181](#)
- inet\_ntoa subroutine [183](#)
- inet\_ntop subroutine [184](#)
- inet\_ntop6\_zone [162](#)
- inet\_ntop6\_zone subroutine [162](#)
- inet\_pton subroutine [185](#)
- inet\_pton6\_zone [163](#)
- inet\_pton6\_zone subroutine [163](#)
- inet6\_is\_srcaddr Subroutine [164](#)
- inet6\_opt\_append Subroutine [165](#)
- inet6\_opt\_find Subroutine [166](#)
- inet6\_opt\_finish Subroutine [167](#)
- inet6\_opt\_get\_val [167](#)
- inet6\_opt\_get\_val Subroutine [167](#)
- inet6\_opt\_init [168](#)
- inet6\_opt\_init Subroutine [168](#)
- inet6\_opt\_next Subroutine [169](#)
- inet6\_opt\_set\_val Subroutine [169](#)
- inet6\_rth\_add Subroutine [170](#)
- inet6\_rth\_getaddr Subroutine [170](#)
- inet6\_rth\_init Subroutine [171](#)
- inet6\_rth\_reverse Subroutine [172](#)
- inet6\_rth\_segments Subroutine [172](#)
- inet6\_rth\_space Subroutine [173](#)
- initializing logging facility variables [46](#)
- initiating SMUX peers [60](#)
- inetgr subroutine [186](#)
- Internet addresses
  - converting [174](#)
  - converting to ASCII strings [183](#)
  - returning network addresses [176](#)
  - searching [228](#)
- Internet numbers
  - converting Internet addresses [174](#)
  - converting network addresses [181](#)
- ioctl BPF Control Operations [319](#)
- ioctl commands [187](#)
- ioctl socket control operations [187](#)
- IP addresses
  - constructing [177](#)
- isinet\_addr Subroutine [198](#)
- ISODE library
  - logging subroutines [47](#)
- isodetailor subroutine [46](#)

## K

- key-value pairs
  - returning first [36](#)
- keys
  - searching for associated values [38](#)
- kvalid\_user subroutine [200](#)

## L

- Libibverb library [377](#)
- Libibverbs library [372](#)
- list of [262](#)
- listen subroutine [201](#)
- ll\_dbinit subroutine [47](#)
- ll\_hdinit subroutine [47](#)
- ll\_log subroutine [47](#)
- local host names
  - retrieving [123](#)
- long byte quantities
  - retrieving [69](#)
- long integers, converting
  - from host byte order [158](#)
  - from network byte order [202](#), [203](#)
  - to host byte order [202](#), [203](#)
  - to network byte order [158](#)

## M

- Management Information Base (MIB)
  - registering a section [61](#)
- manipulating entries [54](#)
- manipulating the [52](#)
- mapping
  - Ethernet number [96](#)
- master servers
  - returning machine names [37](#)
- memory
  - freeing [8](#)
- memory management subroutines
  - getaddrinfo [110](#)
  - getnameinfo [128](#)
- memory region [377](#)
- Memory region management
  - rdma\_dereg\_mr [355](#)
  - rdma\_reg\_msgs [355](#)
  - rdma\_reg\_read [356](#)
  - rdma\_reg\_write [357](#)
- message replies [1](#), [17](#), [18](#)
- MIB list [59](#)
- MIB variables [50](#), [56](#)

## N

- n [202](#)
- name servers
  - creating packets [229](#)
  - creating query messages [229](#)
  - retrieving responses [239](#)
  - sending queries [239](#)
- name2inst subroutine [67](#)
- names
  - binding to sockets [77](#)
- network addresses
  - converting [181](#)
  - returning [176](#)
  - returning network numbers [180](#)
- network entries
  - retrieving [133](#)
  - retrieving by address [129](#)
  - retrieving by name [132](#)

- network host entries
  - retrieving by address [115](#), [116](#)
  - retrieving by name [118](#), [120](#)
- network host files
  - opening [264](#), [265](#)
- Network Information Service [33](#)
- Network Information Services+ (NIS) [33](#)
- next2inst subroutine [67](#)
- nexttot2inst subroutine [67](#)
- NIS maps
  - changing [42](#)
  - returning order number [41](#)
- NIS master servers
  - returning machine names [37](#)
- NIS subroutines
  - yp\_all [33](#)
  - yp\_bind [35](#)
  - yp\_first [36](#)
  - yp\_get\_default\_domain [37](#)
  - yp\_master [37](#)
  - yp\_match [38](#)
  - yp\_next [39](#)
  - yp\_order [41](#)
  - yp\_unbind [41](#)
  - yp\_update [42](#)
  - yperr\_string [43](#)
  - ypprot\_err [44](#)
- ntohl subroutine [202](#)
- ntohl subroutine [203](#)
- ntohs subroutine [204](#)

## O

- o\_ subroutines [50](#)
- o\_generic subroutine [50](#)
- o\_igeneric subroutine [50](#)
- o\_integer subroutine [50](#)
- o\_ipaddr subroutine [50](#)
- o\_number subroutine [50](#)
- o\_specific subroutine [50](#)
- o\_string subroutine [50](#)
- object identifier data structure [52](#)
- object tree (OT) [59](#)
- ode2oid subroutine [52](#)
- OID
  - converting text strings to [68](#)
- OID (object identifier data structure) [52](#)
- oid\_cmp subroutine [52](#)
- oid\_cpy subroutine [52](#)
- oid\_extend subroutine [54](#)
- oid\_free subroutine [52](#)
- oid\_normalize subroutine [54](#)
- oid2ode subroutine [52](#)
- oid2ode\_aux subroutine [52](#)
- oid2prim subroutine [52](#)
- opaque data
  - translating [12](#)
- opening [149](#)

## P

- Packet Capture [319](#)
- peer entries [45](#)

- peer responsibility level [61](#)
- peer socket names
  - retrieving [135](#)
- placing short byte quantities [72](#)
- port mappings
  - describing [15](#)
- portmap procedures
  - describing parameters [14](#)
- PostQueuedCompletionStatus Subroutine [205](#)
- prim2oid [52](#)
- processes
  - managing socket descriptors [41](#)
- Protection domain management
  - ibv\_alloc\_pd [377](#)
  - ibv\_dealloc\_pd [377](#)
- protocol data unit (PDU) [57](#), [62](#), [63](#)
- protocol entries
  - retrieving [141](#)
  - retrieving by name [138](#)
- psap.h file [53](#)

## Q

- queries
  - awaiting response [236](#)
- querying [152](#)
- Queue pair (QP) management
  - rdma\_create\_qp [353](#)
  - rdma\_destroy\_qp [353](#)
  - releases QP [353](#)
- Queue pair management
  - ibv\_create\_qp [364](#)
  - ibv\_destroy\_qp [364](#)
  - ibv\_modify\_qp [365](#)
  - ibv\_post\_recv [368](#)
  - ibv\_post\_send [369](#)
  - Libibverbs library [364](#)
  - work requests [368](#)
- Queue Pair management
  - ibv\_query\_qp [370](#)

## R

- r [206](#)
- rcmd subroutine [206](#)
- rcmd\_af Subroutine [208](#)
- rdma\_accept [332](#)
- rdma\_cm [323](#)
- rdma\_create\_id [326](#)
- rdma\_disconnect [334](#)
- rdma\_listen [332](#)
- ReadFile Subroutine [220](#)
- reading [149](#)
- reading a MIB variable structure into [55](#)
- reading the smux\_errno variable [58](#)
- readobjects subroutine [55](#)
- records
  - input streams
    - moving position [31](#)
    - marking outgoing data as [30](#)
    - skipping [31](#)
- recv subroutine [221](#)
- recvfrom subroutine [223](#)

- recvmsg subroutine [225](#)
- registering an MIB tree for [61](#)
- remote hosts
  - executing commands [206](#)
  - starting command execution [240](#)
- reporting errors to log files [47](#)
- res\_init subroutine [228](#)
- res\_mkquery subroutine [229](#)
- res\_ninit subroutine [231](#)
- res\_query subroutine [233](#)
- res\_search subroutine [236](#)
- res\_send subroutine [239](#)
- retrieving [121](#), [122](#), [134](#), [142](#)
- retrieving by address [130](#)
- retrieving by name [131](#), [137](#), [144](#), [146](#)
- retrieving by number [139](#), [140](#)
- retrieving by port [148](#)
- retrieving host entries [121](#), [122](#)
- retrieving network entries [130](#), [131](#), [134](#)
- retrieving protocol entries [137–142](#)
- retrieving service entries [144](#), [147](#)
- retrieving variables [67](#)
- Returned error rules
  - Libibverbs library [358](#)
- rexec subroutine [240](#)
- rexec\_af Subroutine [242](#)
- RPC authentication messages [13](#)
- RPC authentication subroutines
  - xdr\_authunix\_parms [27](#)
- RPC call header messages [4](#)
- RPC call messages [5](#)
- RPC message replies [1](#), [17](#), [18](#)
- RPC reply messages
  - encoding [1](#)
- RPC subroutines
  - receiving XDR subroutines [25](#)
  - xdr\_accepted\_reply [1](#)
  - xdr\_callhdr [4](#)
  - xdr\_callmsg [5](#)
  - xdr\_opaque\_auth [13](#)
  - xdr\_pmap [14](#)
  - xdr\_pmaplist [15](#)
  - xdr\_rejected\_reply [17](#)
  - xdr\_replymsg [18](#)
- rresvport subroutine [243](#)
- rresvport\_af Subroutine [244](#)
- ruserok subroutine [246](#)

**S**

- s [247](#)
- s\_generic subroutine [56](#)
- SCTP subroutines
  - sctp\_opt\_info [247](#)
  - sctp\_peeloff [248](#)
- sctp\_opt\_info subroutine [247](#)
- sctp\_peeloff subroutine [248](#)
- send subroutine [250](#)
- send\_file
  - send the contents of file through a socket [258](#)
- send\_file subroutine
  - socket options [258](#)
- sending [62](#)
- sending an open [63](#)
- sending an open PDU [63](#)
- sending traps to SNMP [64](#)
- sendmsg subroutine [252](#)
- sendto subroutine [256](#)
- server query mechanisms
  - providing interfaces to [233](#)
- service entries
  - retrieving by port [147](#)
- service file entries
  - retrieving [149](#), [150](#)
- set\_auth\_method subroutine [262](#)
- setdomainname subroutine [263](#)
- sethostent subroutine [264](#)
- sethostent\_r subroutine [265](#)
- sethostid subroutine [266](#)
- sethostname subroutine [267](#)
- setip4sourcefilter [283](#)
- setnetent subroutine [268](#)
- setnetent\_r subroutine [269](#)
- setnetgrent subroutine [186](#)
- setnetgrent\_r subroutine [270](#)
- setprotoent subroutine [271](#)
- setprotoent\_r subroutine [272](#)
- setservent subroutine [272](#)
- setservent\_r subroutine [273](#)
- setsockopt subroutine [274](#)
- setsourcefilter [283](#)
- setting variable values [56](#)
- short byte quantities
  - retrieving [70](#)
- short integers, converting
  - from host byte order [159](#)
  - from network byte order [204](#)
  - to host byte order [204](#)
  - to network byte order [159](#)
- shutdown subroutine [284](#)
- Simple Network Management Protocol (SNMP) [45](#)
- SLP subroutines
  - SLPAttrCallback [286](#)
  - SLPClose [286](#)
  - SLPEscape [288](#)
  - SLPFindAttrs [288](#)
  - SLPFindScopes [289](#)
  - SLPFindSrvs [290](#)
  - SLPFindSrvTypes [291](#)
  - SLPFree [292](#)
  - SLPGetProperty [292](#)
  - SLPOpen [293](#)
  - SLPParseSrvURL [294](#)
  - SLPSrvTypeCallback [296](#)
  - SLPSrvURLCallback [297](#)
  - SLPUnescape [298](#)
- SLPAttrCallback subroutine [286](#)
- SLPClose subroutine [286](#)
- SLPDereg subroutine [287](#)
- SLPEscape subroutine [288](#)
- SLPFindAttrs subroutine [288](#)
- SLPFindScopes subroutine [289](#)
- SLPFindSrvs subroutine [290](#)
- SLPFindSrvTypes subroutine [291](#)
- SLPFree subroutine [292](#)
- SLPGetProperty subroutine [292](#)
- SLPOpen subroutine [293](#)
- SLPParseSrvURL subroutine [294](#)

- SLPReg subroutine [295](#)
- SLPRegReport callback subroutine [296](#)
- SLPSrvTypeCallback subroutine [296](#)
- SLPSrvURLCallback subroutine [297](#)
- SLPUnescape subroutine [298](#)
- SMUX
  - communicating with the snmpd daemon [60](#)
  - initiating transmission control protocol (TCP) [60](#)
  - retrieving peer entries [45](#)
  - setting debug level for subroutines [60](#)
- smux\_close subroutine [57](#)
- smux\_error subroutine [58](#)
- smux\_free\_tree subroutine [59](#)
- smux\_init subroutine [60](#)
- smux\_register subroutine [61](#)
- smux\_response subroutine [62](#)
- smux\_simple\_open subroutine [63](#)
- smux\_trap subroutine [64](#)
- smux\_wait subroutine [66](#)
- smux.h file [58](#)
- SNMP multiplexing peers [45](#)
- snmpd daemon [63](#)
- snmpd.peers file [45](#)
- socket connections
  - accepting [72](#)
  - listening [201](#)
- socket names
  - retrieving [151](#)
- socket options
  - setting [274](#)
- socket receive operations
  - disabling [284](#)
- socket send operations
  - disabling [284](#)
- socket subroutine [299](#)
- socket subroutines
  - freeaddrinfo subroutine [110](#)
  - if\_indextoname subroutine [160](#)
  - if\_nameindex subroutine [161](#)
  - if\_nametoindex subroutine [162](#)
  - inet6\_opt\_append [165](#)
  - inet6\_opt\_find [166](#)
  - inet6\_opt\_finish [167](#)
  - inet6\_opt\_next [169](#)
  - inet6\_opt\_set\_val [169](#)
  - inet6\_rth\_add [170](#)
  - inet6\_rth\_getaddr [170](#)
  - inet6\_rth\_init [171](#)
  - inet6\_rth\_reverse [172](#)
  - inet6\_rth\_segments [172](#)
  - inet6\_rth\_space [173](#)
  - rcmd\_af [208](#)
  - rexec\_af [242](#)
  - rresvport\_af [244](#)
- socketpair subroutine [301](#)
- sockets
  - creating [299](#)
  - initiating TCP for SMUX peers [60](#)
  - managing [316](#)
  - retrieving with privileged addresses [243](#)
- Sockets [69](#)
- sockets kernel service subroutines
  - accept [72](#)
  - bind [77](#)
- sockets kernel service subroutines (*continued*)
  - dn\_comp [84](#)
  - getdomainname [114](#)
  - gethostid [123](#)
  - gethostname [123](#)
  - getpeername [135](#)
  - getsockname [151](#)
  - getsockopt [152](#)
  - listen [201](#)
  - recv [221](#)
  - recvfrom [223](#)
  - recvmsg [225](#)
  - send [250](#)
  - sendmsg [252](#)
  - sendto [256](#)
  - setdomainname [263](#)
  - sethostid [266](#)
  - sethostname [267](#)
  - setsockopt [274](#)
  - shutdown [284](#)
  - socket [299](#)
  - socketpair [301](#)
- sockets messages
  - receiving from connected sockets [221](#)
  - receiving from sockets [223](#), [225](#)
  - sending through any socket [252](#)
- sockets network library subroutines
  - \_getlong [69](#)
  - \_getshort [70](#)
  - \_putlong [71](#)
  - dn\_expand [86](#)
  - endhostent [88](#)
  - endhostent\_r [89](#)
  - endnetent [90](#)
  - endnetent\_r [91](#)
  - endnetgrent\_r [92](#)
  - endprotoent\_r [93](#)
  - endservent [94](#)
  - endservent\_r [95](#)
  - gethostbyaddr [115](#)
  - gethostbyaddr\_r [116](#)
  - gethostbyname [118](#)
  - gethostbyname\_r [120](#)
  - getnetbyaddr [129](#)
  - getnetbyname\_r [132](#)
  - getnetent [133](#)
  - getprotobyname\_r [138](#)
  - getprotoent [141](#)
  - getservbyname\_r [146](#)
  - getservent\_r [150](#)
  - inet\_addr [174](#)
  - inet\_Inaof [176](#)
  - inet\_makeaddr [177](#)
  - inet\_netof [180](#)
  - inet\_network [181](#)
  - inet\_ntoa [183](#)
  - ntohl [202](#)
  - ntohll [203](#)
  - ntohs [204](#)
  - rcmd [206](#)
  - rds [209](#)
  - rds-info [213](#)
  - rds-ping [216](#)
  - rds-rdma [216](#)

sockets network library subroutines (*continued*)

- [res\\_init](#) [228](#)
- [res\\_mkquery](#) [229](#)
- [res\\_query](#) [233](#)
- [res\\_search](#) [236](#)
- [res\\_send](#) [239](#)
- [rexec](#) [240](#)
- [rresvport](#) [243](#)
- [ruserok](#) [246](#)
- [sethostent](#) [264](#)
- [sethostent\\_r](#) [265](#)
- [setnetent](#) [268](#)
- [setnetent\\_r](#) [269](#)
- [setprotoent](#) [271](#)
- [setprotoent\\_r](#) [272](#)
- [setservent](#) [272](#)
- [setservent\\_r](#) [273](#)

- [socks5\\_getserv](#) Subroutine [302](#)
- [socks5tcp\\_accept](#) Subroutine [305](#)
- [socks5tcp\\_bind](#) Subroutine [307](#)
- [socks5tcp\\_connect](#) Subroutine [310](#)
- [socks5udp\\_associate](#) Subroutine [312](#)
- [socks5udp\\_sendto](#) Subroutine [314](#)
- [splice](#) subroutine [316](#)
- [sprintoid](#) subroutine [52](#)
- [str2oid](#) subroutine [52](#)
- [string conversions](#) [68](#)
- structures
  - [providing pointer chasing](#) [15](#), [16](#)
  - [serializing null pointers](#) [15](#)
- [Supported verbs](#) [322](#)
- [Supported Verbs](#)
  - [Libibverbs library](#) [358](#)

## T

- [text2inst](#) subroutine [67](#)
- [text2obj](#) subroutine [68](#)
- [text2oid](#) subroutine [68](#)
- [to network byte order](#) [157](#)
- [traps](#) [64](#)

## U

- unconnected sockets
  - [receiving messages](#) [223](#)
  - [sending messages](#) [252](#), [256](#)
- unions
  - [translating](#) [24](#)
- unique identifiers
  - [retrieving](#) [123](#)
- UNIX credentials
  - [generating](#) [27](#)
- [unregistered trees](#) [59](#)

## V

- [variable bindings](#) [50](#)
- [variable initialization](#) [46](#)

## W

- [waiting for a message](#) [66](#)

[WriteFile Subroutine](#) [317](#)

## X

XDR library filter primitives

- [xdr\\_array](#) [2](#)
- [xdr\\_bool](#) [2](#)
- [xdr\\_bytes](#) [3](#)
- [xdr\\_char](#) [5](#)
- [xdr\\_double](#) [27](#)
- [xdr\\_enum](#) [7](#)
- [xdr\\_float](#) [7](#)
- [xdr\\_int](#) [11](#)
- [xdr\\_long](#) [12](#)
- [xdr\\_opaque](#) [12](#)
- [xdr\\_reference](#) [16](#)
- [xdr\\_short](#) [19](#)
- [xdr\\_string](#) [20](#)
- [xdr\\_u\\_char](#) [21](#)
- [xdr\\_u\\_int](#) [21](#)
- [xdr\\_u\\_long](#) [22](#)
- [xdr\\_u\\_short](#) [23](#)
- [xdr\\_union](#) [24](#)
- [xdr\\_vector](#) [25](#)
- [xdr\\_void](#) [25](#)
- [xdr\\_wrapstring](#) [26](#)

XDR library non-filter primitives

- [xdrrec\\_endofrecord](#) [30](#)
- [xdrrec\\_skiprecord](#) [31](#)
- [xdrstdio\\_create](#) [32](#)

XDR streams

- [changing current position](#) [18](#)
- [containing long sequences of records](#) [29](#)
- [destroying](#) [6](#)
- [initializing](#) [32](#)
- [initializing local memory](#) [28](#)
- [returning pointer to buffer](#) [10](#)

XDR subroutines

- [supplying to RPC system](#) [25](#)
- [xdr\\_accepted\\_reply](#) subroutine [1](#)
- [xdr\\_array](#) subroutine [2](#)
- [xdr\\_authunix\\_parms](#) subroutine [27](#)
- [xdr\\_bool](#) subroutine [2](#)
- [xdr\\_bytes](#) subroutine [3](#)
- [xdr\\_callhdr](#) subroutine [4](#)
- [xdr\\_callmsg](#) subroutine [5](#)
- [xdr\\_char](#) subroutine [5](#)
- [xdr\\_destroy](#) macro [6](#)
- [xdr\\_double](#) subroutine [27](#)
- [xdr\\_enum](#) subroutine [7](#)
- [xdr\\_float](#) subroutine [7](#)
- [xdr\\_free](#) subroutine [8](#)
- [xdr\\_getpos](#) macro [9](#)
- [xdr\\_hyper](#) subroutine [10](#)
- [xdr\\_inline](#) macro [10](#)
- [xdr\\_int](#) subroutine [11](#)
- [xdr\\_long](#) subroutine [12](#)
- [xdr\\_opaque](#) subroutine [12](#)
- [xdr\\_opaque\\_auth](#) subroutine [13](#)
- [xdr\\_pmap](#) subroutine [14](#)
- [xdr\\_pmaplist](#) subroutine [15](#)
- [xdr\\_pointer](#) subroutine [15](#)
- [xdr\\_reference](#) subroutine [16](#)
- [xdr\\_rejected\\_reply](#) subroutine [17](#)

xdr\_replymsg subroutine [18](#)  
xdr\_setpos macro [18](#)  
xdr\_short subroutine [19](#)  
xdr\_string subroutine [20, 26](#)  
xdr\_u\_char subroutine [21](#)  
xdr\_u\_int subroutine [21](#)  
xdr\_u\_long subroutine [22](#)  
xdr\_u\_short subroutine [23](#)  
xdr\_union subroutine [24](#)  
xdr\_vector subroutine [25](#)  
xdr\_void subroutine [25](#)  
xdr\_wrapstring subroutine [26](#)  
xdrmem\_create subroutine [28](#)  
xdrrec\_create subroutine [29](#)  
xdrrec\_endofrecord subroutine [30](#)  
xdrrec\_eof subroutine [31](#)  
xdrrec\_skiprecord subroutine [31](#)  
xdrstdio\_create subroutine [32](#)

## Y

yp\_all subroutine [33](#)  
yp\_bind subroutine [35](#)  
yp\_first subroutine [36](#)  
yp\_get\_default\_domain subroutine [37](#)  
yp\_master subroutine [37](#)  
yp\_match subroutine [38](#)  
yp\_next subroutine [39](#)  
yp\_order subroutine [41](#)  
yp\_unbind subroutine [41](#)  
yp\_update subroutine [42](#)  
ypbind daemon  
    calling [35](#)  
yperr\_string subroutine [43](#)  
ypprot\_err subroutine [44](#)





