VSI OpenVMS

VSI TCP/IP Programmer's Reference

Document Number: DO-DVTPRG-00A

Publication Date: August 2018

This manual documents the programmer's interface to VSI TCP/IP and is intended to guide the programmer in developing applications that use network services.

Revision Update Information: This is a new manual.

Operating System and Version: VSI OpenVMS Version 8.4-2L1 or higher

Software Version: VSI TCP/IP for OpenVMS Version 10.5

VMS Software, Inc., (VSI)
Bolton, Massachusetts, USA

VSI TCP/IP Programmer's Reference:

Legal Notice

The VSI OpenVMS documentation set is available on DVD.

Preface

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

VSI seeks to continue the legendary development prowess and customer-first priorities that are so closely associated with the OpenVMS operating system and its original author, Digital Equipment Corporation.

2. Intended Audience

This manual is intended for programmers who will develop applications that use VSI TCP/IP network services. It provides the description of programmer's interface to VSI TCP/IP and contains information about:

- Various aspects of application programming using VSI TCP/IP
- Purpose and format of each VSI TCP/IP socket library function
- $QIO Interface
- Application Programming Interface (API) routines required for an application program to export private Management Information Bases (MIBs) using the VSI TCP/IP SNMP agent
- VSI TCP/IP RPC Services and building distributed applications with RCP
- RPCGEN Compiler
- RPC Run-Time Library (RTL) conventions and management, client, port-mapper, server, XDR routines in the RPC RTL

3. Typographical Conventions

The following conventions are used in this manual:

Convention	Meaning
Ctrl/*X*	A sequence such as **Ctrl/*X*** indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *X*	A sequence such as **PF1 *X*** indicates that you must first press and release the key labeled PF1 and then press and release another key (***x***) or a pointing device button.
Enter	In examples, a key name in bold indicates that you press that key.
. . .	A horizontal ellipsis in examples indicates one of the following possibilities:- Additional optional arguments in a statement have been omitted.- The preceding item or items can be repeated one or more times.- Additional parameters, values, or other information can be entered.

Convention	Meaning
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose choices in parentheses if you specify more than one. In installation or upgrade examples, parentheses indicate the possible answers to a prompt, such as: `Is this correct? (Y/N) [Y]`
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for directory specifications and for a substring specification in an assignment statement. In installation or upgrade examples, brackets indicate the default answer to a prompt if you press **Enter** without entering a value, as in: `Is this correct? (Y/N) [Y]`
|	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are optional; within braces, at least one choice is required. Do not type the vertical bars on the command line.
{ }	In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line.
bold type	Bold type represents the name of an argument, an attribute, or a reason. In command and script examples, bold indicates user input. Bold type also represents the introduction of a new term.
italic type	Italic type indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER=*name*), and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE TYPE	Uppercase type indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
`Example`	This typeface indicates code examples, command examples, and interactive screen displays. In text, this type also identifies website addresses, UNIX command and pathnames, PC-based commands and folders, and certain elements of the C programming language.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes-binary, octal, or hexadecimal-are explicitly indicated.

4. VSI TCP/IP Support

VSI supports VSI TCP/IP running on VSI OpenVMS Integrity Version 8.4-2L1 (or higher) only. Please contact your support channel for help with this product. Users who have OpenVMS support

contracts through VSI can contact support@vmssoftware.com [mailto:support@vmssoftware.com] for help with this product. Users who have OpenVMS support contracts through HPE should contact their HPE Support channel for assistance.

5. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmssoftware.com>.

6. How to Order Additional Documentation

For information about how to order additional documentation, email the VSI OpenVMS information account: <info@vmssoftware.com>. We will be posting links to documentation on our corporate website soon.

Chapter 1. VSI TCP/IP Programming Tutorial

This chapter contains short tutorials on various aspects of application programming using VSI TCP/IP.

1.1. Sockets

A socket is an endpoint for communication. Two cooperating sockets, one on the local host and one on the remote host, form a connection. Each of the two sockets has a unique address that is described generically by the **sockaddr** C programming language structure. The **sockaddr** structure is defined as follows:

```
struct sockaddr {
 u_char sa_len;      /* length of data structure */
 u_char sa_family;   /* Address family */
 char sa_data[14];   /* up to 14 bytes of direct address*/
};
```

The **sa_family** field specifies the address family for the communications domain to which the socket belongs. For example, AF_INET for the Internet family. The **sa_data** field contains up to 14 bytes of data, the interpretation of which depends on the value of **sa_family**.

If the **sa_family** field is AF_INET, the same sockaddr structure can also be interpreted as a **sockaddr_in** structure that describes an Internet address. A **sockaddr_in** structure is defined as follows:

```
struct sockaddr_in {
 u_char sin_len;
 u_char sin_family;
 u_short sin_port;
 struct in_addr sin_addr;
 char sin_zero[8];
};
```

The **sin_family** field specifies the address family AF_INET. The **sin_port** field specifies the TCP (Transmission Control Protocol) or UDP (User Datagram Protocol) port number of the address. Whether the communication uses TCP or UDP is not determined here, but rather by the type of socket created with the **socket()** call: SOCK_STREAM for TCP or SOCK_DGRAM for UDP. The **sin_addr** field specifies the Internet address. The **sin_zero** field must be zero. Both the **sin_port** field and the **sin_addr** field are in network byte order. For more information about network byte ordering see Chapter 4, *$QIO Interface*.

The **sockaddr** and **sockaddr_in** structures serve as input and output to a number of library routines. For example, they may be used as input, specifying the address to which to make a connection or send a packet, or as output, reporting the address from which a connection was made or a packet transmitted.

Internet addresses are normally manipulated with the **gethostbyname()**, **gethostbyaddr()**, **inet_addr()**, and **inet_ntoa()** functions. **gethostbyname()** and **inet_addr()** convert a host name or ASCII representation of an address into the binary representation for the **sockaddr_in** structure. **gethostbyaddr()** and **inet_ntoa()** are used to convert the binary representation into the host name or ASCII representation for display.

Port numbers are normally manipulated with the **getservbyname()** and **getservbyport()** functions. **getservbyname()** converts the ASCII service name to the numeric value, and **getservbyport()** converts the numeric value to the ASCII name.

The following example shows a typical program that converts the Internet address and the port into binary representations.

```
#include "IP$root:[IP.include.sys]types.h"
#include "IP$root:[IP.include.sys]socket.h"
#include "IP$root:[IP.include]netdb.h"
#include "IP$root:[IP.include.netinet]in.h"

main(argc,argv)
int argc;
char *argv[];
{
 struct sockaddr_in sin;
 struct hostent *hp;
 struct servent *sp;

 /* Zero the sin structure to initialize it */

 bzero((char *) &sin, sizeof(sin));
 sin.sin_family = AF_INET;

 /* Lookup the host and initialize sin_addr */

 hp = gethostbyname(argv[1]);
 if (!hp) { /* Perhaps it is an ASCII string */
 sin.sin_addr.s_addr = inet_addr(argv[1]);
 if (sin.sin_addr.s_addr == -1) {
 printf("syntax error in IP address\n");
 exit(1);
 }
} else {  /* Extract the IP address */
 bcopy(hp->h_addr, (char *) &sin.sin_addr,
  hp->h_length);
}

/* Lookup up the name of the SMTP service */

sp = getservbyname("smtp","tcp");|
if (!sp) {
 printf("unable to find smtp service");
 exit(1);
}

sin.sin_port = sp->s_port;

/* Now we are ready to create a socket and
pass the address of this sockaddr_in
structure to the connect() call to
connect to the remote SMTP port */
}
```

1.2. TCP Client

A TCP client process establishes a connection to a server and uses the **socket_read()** and **socket_write()** functions to transfer data. Typically, you use the following sequence of functions to set up the connection:

1. Create a TCP socket:

   ```
   socket(AF_INET, SOCK_STREAM, 0);
   ```

2. Set up a **sockaddr_in** structure with the address you want to connect to by calling **gethostbyname()** and **getservbyname()**.

3. Make a connection to the server with the **connect()** function.

4. Once **connect()** completes, the TCP connection is established and you can use **socket_read()** and **socket_write()** to transfer data.

1.3. TCP Server

A TCP server process binds a socket to a well-known port and listens on that port for connection attempts. When a connection arrives, the server processes it by transferring data using **socket_read()** and **socket_write()**. Typically, you use the following sequence of functions to set up a server:

1. Create a TCP socket:

   ```
   socket(AF_INET, SOCK_STREAM, 0);
   ```

2. Use the **getservbyname()** function to get the port number of the service on which you want to listen for connections.

3. Set up a **sockaddr_in** structure with the port number and an Internet address of INADDR_ANY, and bind this address to the socket with the **bind()** function.

4. Use the **listen()** function to inform the VSI TCP/IP kernel that you are listening for connections on this socket. Then wait for a connection and accept it with **accept()**.

5. Once **accept()** completes, the TCP connection is established and you can use **socket_read()** and **socket_write()** to transfer data. When you are done with the connection, you can close the channel returned by **accept()** and start a new **accept()** call on the original channel to wait for another connection.

Note

When writing a TCP server that will run under the control of the `IP_SERVER` process, you must assign a channel to `SYS$INPUT` before calling any of the C I/O routines.

Another way to write a TCP server is to let the `IP$SERVER` process do the work for you. The `IP$SERVER` can perform all of the above steps, and when a connection request arrives, can use the OpenVMS system service $CREPRC to create a process running your program.

1.4. UDP

A UDP program sends and receives packets to and from a remote port using the **send()** or **sendto()** and **recv()** or **recvfrom()** functions. UDP is a connectionless transport protocol. It does not incur the

overhead of creating and maintaining a connection between two sockets, but rather merely sends and receives datagrams. It is not a reliable transport, and does not provide guaranteed data delivery, packet ordering, or flow control.

Typically, you use the following sequence of functions in a UDP program:

1. Create a UDP socket:

   ```
   socket(AF_INET, SOCK_DGRAM, 0);
   ```

2. Bind the socket to a local port number with the **bind()** function. Specify the **sin_port** field as 0 (zero) if you want VSI TCP/IP to choose an unused port number for you automatically (typical of a client), or specify the **sin_port** field as the UDP port number (typical of a server). The **sin_addr** field is usually specified as INADDR_ANY, which means that packets addressed to any of the host's Internet addresses are accepted.

3. Optionally, use **connect()** to specify the remote port and Internet address. If you do not use **connect()**, you must use **sendto()** to specify the remote address when you send packets, and **recvfrom()** to learn the address when you receive them.

4. Read and write packets to transfer data using the **send()** or **sendto()** and **recv()** or **recvfrom()** functions, respectively.

Note

When writing a UDP server that will run under the control of the `IP_SERVER` process, you must assign a channel to `SYS$INPUT` before calling any of the C I/O routines.

Another way to write a UDP server is to let the `IP$SERVER` process handle the work. The `IP $SERVER` can perform all the above steps, and when a packet arrives on a UDP port, can use the OpenVMS system service $CREPRC to create a process running your program.

1.5. BSD-Specific Tips

The following sections contain information specific to working with BSD code.

1.5.1. BSD Sockets Porting Note

When porting a program written for BSD sockets to VSI TCP/IP, observe the following guidelines:

- Change any *#include* statements to reference files with the same names in the `IP$ ROOT: [IP.INCLUDE...]` directory areas.
- Implement your change in the source code using *#ifdef* statements to enable the use of VSI TCP/IP include files; you can then compile your software in a UNIX environment by selecting the other side of the *#ifdef*.

1.5.2. BSD 4.4 TCP/IP Future Compatibility Considerations

VSI TCP/IP supports both BSD 4.3 and BSD 4.4 format sockaddrs.

The BSD 4.4 format is:

```
struct sockaddr_in {
  u_char   sin_len;
  u_char   sin_family;
  u_char   sin_port;
  struct   in_addr sin_addr;
  char     sin_zero[8];
};
```

The BSD 4.3 format of the sockaddr_in structure is:

```
struct sockaddr_in {
  short     sin_family;
  u_short   sin_port;
  struct    in_addr sin_addr;
  char      sin_zero[8];
};
```

VSI TCP/IP will accept either format from customer applications. This affects applications that explicitly check the **sin_family** field for the value AF_INET. Applications can avoid incompatibilities by avoiding explicit references or checks of the **sin_family** field, or by assuming that it can be in either format. The INET device uses the `IO$M_EXTEND` modifier to specify that a BSD 4.4 sockaddr (or current format) is used when `IO$M_EXTEND` is not used on the function code, the old (BSD 4.3) format is used.

Support for the BSD 4.4 style *`sockaddr`* data structure is included in the BGDRIVER (UCX interface). If the `IO$M_EXTEND` modifier is set on any one of the following QIO operations, the *`sockaddr`* parameter passed in these operations is assumed to be in BSD 4.4 format.

- `IO$_SETMODE/IO$_SETCHAR` (socket, bind)
- `IO$_ACCESS` (connect, listen)
- `IO$_SENSEMODE/IO$_SENSECHAR` (getsockname, getpeername)
- `IO$_READVBLK` (recv_from, when P3 is specified for a UDP or raw IP message)
- `IO$_WRITEVBLK` (send_to, when P3 is specified for a UDP or raw IP message)

When the `IO$M_EXTEND` modifier is used in the creation of a socket via `IO$_SETMODE/IO$_SETCHAR` (socket, bind), the setting is remembered for the lifetime of the socket and all **sockaddr** structures passed in are assumed to be in BSD 4.4 format. Refer to the *TCP/IP Services for OpenVMS System Services and C Socket Programming* manual for additional information.

Operations that return a **sockaddr** (READVBLK (recv_from) like accept, getsockname, and getpeername), return that **sockaddr** in BSD 4.4 format. Operations that accept a **sockaddr** (WRITEVBLK (send_to) like connect and bind) expect the address family value to be in the position it is in for the BSD 4.4 structure. When a **CONNECT /BIND /ACCEPT** operation is done for a TCP connection with the `IO$V_EXTEND` bit set, the setting is remembered for the duration of the connection and all specified **sockaddr** structures are expected to be in BSD 4.4 format, and operations returning a **sockaddr** will return it in BSD 4.4 format.

For `IO$_ACCESS` (connect) and `IO$_SETMODE` (bind), if the portion of the **sockaddr** structure that is used to specify the address family in BSD 4.4 format is non-zero, then the **sockaddr** structure is assumed to be in BSD 4.4 format.

1.5.3. TCP/IP Services (UCX) Compatibility

VSI TCP/IP supports programs written for TCP/IP Services. The C run time library will automatically use the compatible entry points in the UCX$IPC_SHR.EXE image included with VSI TCP/IP. VSI TCP/IP supports the following IPv6 compatible routines:

```
getaddrinfo
freeaddrinfo
getnameinfo
 gai_strerror
inet_pton
inet_ntop
```

Chapter 2. Socket Library Functions

This chapter describes the purpose and format of each VSI TCP/IP socket library function.

The socket functions described in this chapter are available in the shareable image IP$:IP $SOCKET_LIBRARY.EXE, included in the standard VSI TCP/IP distribution.

In addition to supporting the VSI TCP/IP socket library, applications developed for the VSI OpenVMS/ULTRIX Connection (UCX) software using the C socket library (`UCX$IPC.OLB`) will run over VSI TCP/IP, using an emulation of UCX$IPC_SHR.EXE.

Note

To avoid potential conflicts between VSI TCP/IP socket library definitions and C compiler definitions, include a reference to the file `IP$ROOT:[IP.INCLUDE.SYS]TYPES`.H before any other header file references.

2.1. Debugging and Tracing

VSI TCP/IP provides a call tracing facility that can be used to debug and trace the use of the sockets API for many applications. This facility works for both the VSI TCP/IP socket library and the API that the newer versions of the C compiler work with. This does NOT log QIO operations. To enable the tracing define the `IP$SOCKET_TRACE` logical name. The value of the logical name can be used in the following ways:

- As a bit mask for types of operations to trace. Bit 0 (zero) signifies control operations, bit 1signifies read operations and bit 2 signifies write operations. When these values are used the information is written to `SYS$OUTPUT:`.
- As a partial or full file name. When used as a partial file name the default name specified to open the file is: `SYS$SCRATCH:IP$SOCKET_<process_name>.LOG`. Control, read and write operations are logged when logging is done to a file.

2.2. AST Reentrancy

The VSI TCP/IP socket library is based on the equivalent UNIX programming library, and was therefore not designed with reentrancy in mind. If you call into the socket library with AST delivery disabled, some of the library routines will suspend execution and fail to return control to the caller.

This situation occurs most often when applications try to call those functions from within an AST routine where AST delivery is not possible.

Any routine that relies on the **select()** function is subject to this restriction (including the **select()** call itself, and most of the domain name resolution routines such as **gethostbyname()**, and so on).

Another reentrancy consideration is the socket library's use of static internal data structures, some of which are passed back to the application, as in the case of the **hostent** structure address returned by **gethostbyname()**. Other functions use these data structures internally to maintain context.

In either case, it is dangerous to call into these routines from an AST because it is possible to interrupt a similar call already in progress, using the same static buffer, thereby corrupting the contents of the buffer.

Another consideration is the use of routines that send and receive data. Every socket in the kernel contains two fixed-size buffers for sending and receiving data. If an application tries to transmit data when there is insufficient buffer space, that call will block (or suspend execution) until buffer space becomes available. This can become an issue if the application blocks while attempting to transmit a large data buffer, and an AST routine tries to transmit a small data buffer. The small data buffer is transmitted before the large one.

The same situation applies to the functions that read data from the network. This situation may also arise if multiple reads and writes are performed on sockets which have been set up to be non-blocking (NBIO).

These considerations might seem overly restrictive; however, the VSI TCP/IP socket library is a port of the BSD socket library, which is subject to all of the same restrictions. Applications which need to perform I/O from within AST routines should use the `SYS$QIO` system service to talk directly to the VSI TCP/IP device driver.

Therefore, *none* of the socket routines should be considered AST reentrant.

The following are the Socket Library functions:

accept()/accept_44()	**ntohl()**
bcmp()	**ntohs()**
bind()/bind_44()	**recv()/recv_44()**
Section 2.3, “Domain Name Resolver Routines”	**recvfrom()recvfrom_44()**
endhostent()	**recvmsg()/recvmsg_44()**
endnetent()	**select()**
endprotoent()	**select_wake()**
endservent()	**send()/send_44()**
getaddrinfo()	**getnameinfo()**
getdtablesize()	**sendmsg()/sendmsg_44()**
gethostbyaddr()/gethostbyaddr_44()	**sendto()/sendto_44**
gethostbysockaddr()/gethostbysockaddr_44()	**sethostent()**
gethostname()	**setnetent()**
getnetbyaddr()	**setprotoent()**
getnetbyname()	**setservent()**
getpeername()/getpeername_44()	**setsockopt()**
getprotobyname()	**shutdown()**
getprotobynumber()	**socket()**
getprotoent()	**socket_close()**
getservbyname()	**socket_ioctl()**
getservbyport()	**socket ioctl FIONBIO**
getservent()	**socket ioctl FIONREAD**
getsockname()/getsockname_44()	**socket ioctl SIOCADDRT**
getsockopt()	**socket ioctl SIOCDELRT**
gettimeofday()	**socket ioctl SIOCATMARK**

hostalias()	**socket ioctl SIOCDARP**
htonl()	**socket ioctl SIOCGARP**
htons()	**socket ioctl SIOCSARP**
inet_addr()	**socket ioctl SIOCGIFADDR**
inet_lnaof()	**socket ioctl SIOCSIFADDR**
inet_makeaddr()	**socket ioctl SIOCGIFBRDADDR**
inet_netof()	**socket ioctl SIOCSIFBRDADDR**
inet_network()	**socket ioctl SIOCGIFCONF**
inet_ntoa()	**socket ioctl SIOCGIFDSTADDR**
klread()	**socket ioctl SIOCSIFDSTADDR**
klseek()	**socket ioctl SIOCGIFFLAGS**
klwrite()	**socket ioctl SIOCSIFFLAGS**
listen()	**socket ioctl SIOCGIFMETRIC**
ip_kernel_nlist	**socket ioctl SIOCSIFMETRIC**
nlist()	**socket ioctl SIOCGIFNETMASK**
socket option SO_BROADCAST	**socket ioctl SIOCSIFNETMASK**
socket option SO_DEBUG	**socket option SO_REUSEADDR**
socket option SO_DONTROUTE	**socket option SO_SNDBUF**
socket option SO_ERROR	**socket option SO_RCVLOWAT**
socket option SO_KEEPALIVE	**socket option SO_SNDTIMEO**
socket option SO_LINGER	**socket option SO_TYPE**
socket option SO_OOBINLINE	**socket option TCP_KEEPALIVE**
socket option SO_RCVBUF	**socket option TCP_NODELAY**
socket option SO_RCVLOWAT	**socket_perror()**
socket option SO_RCVTIMEO	**socket_read()**
vms_errno_string()	**socket_write()**

accept()/accept_44()

accept()/accept_44() — Extracts the first connection from the queue of pending connections on a socket, creates a new socket with the same properties as the original socket, and assigns a new OpenVMS channel to the new socket. If no pending connections are present on the queue, **accept()** blocks the caller until a new connection is present. The original socket remains open and can be used to accept more connections, but the new socket cannot be used to accept additional connections.

Format

New_VMS_Channel = accept(VMS_Channel, Address, AddrLen);

short New_VMS_Channel, VMS_Channel;

struct sockaddr *Address;

unsigned int *AddrLen;

Description

The original socket is created with the **socket()** function, bound to an address with **bind()**, and is listening for connections after a **listen()**.

The **accept()** function is used with connection-based socket types. Currently the only connection-based socket is SOCK_STREAM, which, together with AF_INET, constitutes a TCP socket.

The **accept_44()** function is the BSD 4.4 sockaddr variant of this call. This call is used automatically when `IP$ROOT:[IP.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

Arguments

VMS_Channel	
VMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the original socket from which to accept the connection.

Address	
VMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**write only**
mechanism:	**by reference**

The optional Address argument is a result parameter. It is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the Address argument is determined by the domain in which the communication is occurring.

AddrLen	
VMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

On entry, the optional **AddrLen** argument contains the length of the space pointed to by **Address**, in bytes. On return, it contains the actual length, in bytes, of the address returned.

Returns

If the **accept()** is successful, an OpenVMS channel number is returned. If an error occurs, a value of -1 is returned, and a more specific message is returned in the global variables **socket_errno** and **vmserrno**.

An error code of *`ENETDOWN`* can indicate that the program has run out of VMS channels to use in creating new sockets. This can be due to either the *`SYSGEN`* parameter *`CHANNELCNT`* being too low

for the number of connections in use by the program, or to a socket leak in the code. Make sure the code closes the socket (using **close()**) when it is done with the socket.

bcmp()

bcmp() — Compares a range of memory. This function operates on variable-length strings of bytes and does not check for null bytes as **strcmp()** does. **bcmp()** is part of the 4.3BSD run-time library, but is not provided by VSI as part of the C run-time library. It is provided here for compatibility with the 4.3BSD library.

Format

Status = bcmp(String1, String2, Length);

char *String1, *String2;

unsigned int Length;

Arguments

String1, String2	
VMS Usage:	**arbitrary**
type:	**byte buffer**
access:	**read only**
mechanism:	**by reference**

Pointers to the two buffers to be compared.

Length	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The number of bytes to be compared.

Returns

The **bcmp()** function returns zero if the strings are identical. It returns a nonzero value if they are different.

bcopy()

bcopy() — Copies memory from one location to another. This function operates on variable-length strings of bytes and does not check for null bytes as **strcpy()** does. **bcopy()** is part of the 4.3BSD run-time library, but is not provided by VSI as part of the C run-time library. It is provided here for compatibility with the 4.3BSD library.

Format

(void) bcopy(String1, String2, Length);

char *String1, *String2;

unsigned int Length;

Arguments

String1	
VMS Usage	**arbitrary**
type:	**byte buffer**
access:	**read only**
mechanism:	**by reference**

The source buffer for the copy operation.

String2	
VMS Usage:	**arbitrary**
type:	**byte buffer**
access:	**write only**
mechanism:	**by reference**

The destination buffer for the copy operation.

Length	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The number of bytes to be copied.

bind()/bind_44()

bind()/bind_44() — Assigns an address to an unnamed socket. When a socket is created with **socket()**, it exists in a name space (address family) but has no assigned address. **bind()** requests that the address be assigned to the socket. If the port number specified in the **sin_port** field of the **sockaddr** structure is less than 1024, SYSPRV is required to use this function. The **bind_44()** function is the BSD 4.4 sockaddr variant of this call. This call is used automatically when `IP$ROOT:[IP.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

Format

Status = bind(VMS_Channel, Name, NameLen);

short VMS_Channel;

struct sockaddr *Name;

unsigned int NameLen;

Arguments

VMS_Channel	
VMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Name	
VMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**read only**
mechanism:	**by reference**

The address to which the socket should be bound. The exact format of the **Address** argument is determined by the domain in which the socket was created.

NameLen	
VMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The length of the Name argument, in bytes.

Returns

If the **bind()** is successful, a value of 0 is returned. If an error occurs, a value of -1 is returned, and a more specific message is returned in the global variables **socket_errno** and **vmserrno**.

bzero()

bzero() — Fills memory with zeros. **bzero()** is part of the 4.3BSD run-time library, but is not provided by VSI as part of the C run-time library. It is provided here for compatibility with the 4.3BSD library.

Format

(void) bzero(String, Length);

char *String;

unsigned int Length;

Arguments

String	
VMS Usage:	**arbitrary**
type:	**byte buffer**
access:	**write only**
mechanism:	**by reference**

The address of the buffer to receive the zeros.

Length	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The number of bytes to be zeroed.

connect()/connect_44()

connect()/connect_44() — When used on a SOCK_STREAM socket, **connect()** attempts to make a connection to another socket. This function, when used on a SOCK_DGRAM socket, permanently specifies the peer to which datagrams are sent to and received from. The peer socket is specified by name, which is an address in the communications domain of the socket. Each communications domain interprets the name parameter in its own way. If the address of the local socket has not yet been specified with **bind()**, the local address is also set to an unused port number when **connect()** is called. The **connect_44()** function is the BSD 4.4 sockaddr variant of this call. This call is used automatically when `IP$ROOT:[IP.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

Format

Status = connect(VMS_Channel, Name, NameLen);

short VMS_Channel;

struct sockaddr *Name;

unsigned int NameLen;

Arguments

VMS_Channel	
VMS Usage:	**channel**

type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Name	
VMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**read only**
mechanism:	**by reference**

The address of the peer to which the socket should be connected. The exact format of the **Address** argument is determined by the domain in which the socket was created.

NameLen	
VMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The length of the **Name** argument, in bytes.

Returns

If the **connect()** is successful, a value of 0 is returned. If an error occurs, a value of -1 is returned, and a more specific message is returned in the global variables **socket_errno** and **vmserrno**.

2.3. Domain Name Resolver Routines

The following functions exist for compatibility with UNIX 4.3BSD programs that call the DNS Name Resolver directly rather than through **gethostbyname()**. The arguments and calling conventions are compatible with BIND Version 4.8.3. They are subject to change and are not documented here.

The `h_errno` variable in the VSI TCP/IP socket library that contains the error status of the resolver routine is accessible to C programs.

dn_comp()	p_rr()
dn_expand()	p_type()
dn_skip()	putlong()
dn_skipname()	putshort()
fp_query()	_res_close()
_getlong()	res_init()
_getshort()	res_mkquery()
herror()	res_query()
p_cdname()	res_querydomain()

p_class()	res_search()
p_query()	res_send()

endhostent()

endhostent() — Tells the DNS Name Resolver to close the TCP connection to the DNS Name Server that may have been opened as the result of calling **sethostent()** with **StayOpen** set to 1.

Format

(void) endhostent();

endnetent()

endnetent() — Tells the DNS Name Resolver to close the TCP connection to the DNS Name Server that may have been opened as the result of using **setnetent()** with **StayOpen** set to 1.

Format

(void) endnetent();

endprotoent()

endprotoent() — Tells the host table routines that the scan started by **getprotoent()** is complete. **endprotoent()** is provided only for compatibility with UNIX 4.3BSD, and is ignored by the VSI TCP/IP software.

Format

(void) endprotoent();

endservent()

endservent() — Tells the host table routines that the scan started by **getservent()** is complete. **endservent()** is provided only for compatibility with UNIX 4.3BSD, and is ignored by the VSI TCP/IP software.

Format

(void) endservent();

getdtablesize()

getdtablesize() — Returns the maximum number of channels available to a process. This function is normally used to determine the **Width** argument to the **select()** function.

Format

Width = getdtablesize();

Returns

The size of the channel table.

gethostbyaddr()/gethostbyaddr_44()

gethostbyaddr()/gethostbyaddr_44() — Looks up a host by its address in the binary host table or the DNS Name Server and returns information about that host. An alternate entry point **_gethostbyaddr()**, that looks only in the binary host table, is also available. The VSI TCP/IP socket library is not reentrant. If you call into it from an AST (interrupt) routine, the results are unpredictable. The **gethostbyaddr_44()** function is the BSD 4.4 sockaddr variant of this call. This call is used automatically when `IP$ROOT:[IP.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

Format

(struct hostent *) gethostbyaddr(Addr, Length, Family);

(struct hostent *) _gethostbyaddr(Addr, Length, Family);

char *Addr;

unsigned int Length;

unsigned int Family;

Arguments

Addr	
VMS Usage:	**address**
type:	**dependent on Family**
access:	**read only**
mechanism:	**by reference**

A pointer to the address to look up. The type is dependent on the **Family** argument. For Internet (AF_INET family) addresses, **Addr** is a pointer to an **in_addr** structure.

Length	
VMS Usage:	**address_length**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The size, in bytes, of the buffer pointed to by Addr.

Family	
VMS Usage:	**address_family**
type:	**longword (unsigned)**

access:	**read only**
mechanism:	**by value**

The address family, and consequently the interpretation of the **Addr** argument. Normally, this is AF_INET, indicating the Internet family of addresses.

Returns

If **gethostbyaddr()** succeeds, it returns a pointer to a structure of type **hostent**. (See **int sctp_getaddrlen (int family)** for more information on the hostent structure.) If this function fails, a value of 0 is returned, and the global variable **h_errno** is set to one of the DNS Name Server error codes defined in the file `ip$root:[IP.include]netdb.h`.

getaddrinfo()

getaddrinfo() — Looks up hostname and/or service name and returns results. This call supports both IPv4 and IPv6 requests.

Format

int getaddrinfo(hostname, servname, hints, res)

char *hostname, *servname;

struct addrinfo *hints, **res;

Arguments

hostname	
VMS Usage:	**host_name**
type:	**ASCIZ string**
access:	**read only**
mechanism:	**by reference**

A C-language string containing the name of the host to look up.

servname	
VMS Usage:	**service_name**
type:	**ASCIZ string**
access:	**read only**
mechanism:	**by reference**

A C-language string containing the name of the service to look up.

hints	
VMS Usage:	**hints**
type:	**struct addrinfo**

access:	**read only**
mechanism:	**by reference**

An addrinfo structure that provides hints on the lookups to be performed.

res	
VMS Usage:	**results**
type:	**pointer**
access:	**write only**
mechanism:	**by reference**

A linked list of addrinfo structures that contain the results of the operation.

Returns

An integer value is returned. Zero is success, non-zero is failure. Failure values can be interpreted with gaistrerror().

```
struct addrinfo {
 int ai_flags;
 int ai_family;
 int ai_socktype;
 int ai_protocol;
 size_t ai_addrlen;
 char *ai_canonname;
 struct sockaddr *ai_addr;
 struct addrinfo *ai_next;
};
```

Use freeaddrinfo(res) to free the chain of data structures returned when the program is done using it.

getnameinfo()

getnameinfo() — Returns hostname and/or servicename information from a sockaddr structures. This call can handle both IPv6 and IPv4 requests.

Format

int getnameinfo(sa, salen, host, hostlen, serv, servlen, flags)

struct sockaddr *sa;

size_t salen, hostlen, servlen;

char *host, *serv;

int flags;

Arguments

sa	

VMS Usage:	**sockaddr**
type:	**sockaddr**
access:	**read only**
mechanism:	**by reference**

A pointer to a sockaddr to obtain information on.

salen	
VMS Usage:	**sockaddr length**
type:	**integer**
access:	**read only**
mechanism:	**by value**

The length of the sockaddr structure.

host	
VMS Usage:	**hostname**
type:	**ASCIZ string**
access:	**write only**
mechanism:	**by reference**

Storage area for a hostname to be returned.

hostlen	
VMS Usage:	**length of hostname string space**
type:	**integer**
access:	**read only**
mechanism:	**by value**

The amount of space available in the host string for storing the hostname.

serv	
VMS Usage:	**service_name**
type:	**ASCIZ string**
access:	**write only**
mechanism:	**by reference**

Storage area for a service name to be returned.

servlen	
VMS Usage:	**length of servicename string space**
type:	**integer**
access:	**read only**

mechanism:	**by value**

The amount of space available in the **serv** string for storing the service name

Returns

An integer value is returned. Zero is success, non-zero is failure. Failure values can be interpretted with gaistrerror().

gethostbyname()/gethostbyname_44()

gethostbyname()/gethostbyname_44() — Looks up a host by name in the binary host table or the DNS Name Server and returns information about that host. An alternate entry point **_gethostbyname()**, that looks only in the binary host table, is also available. The VSI TCP/IP socket library is not reentrant. If you call into it from an AST (interrupt) routine, the results are unpredictable. The **gethostbyname_44()** function is the BSD 4.4 sockaddr variant of this call. This call is used automatically when `IP$ROOT:[IP.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

Format

(struct hostent *) gethostbyname(Name);

(struct hostent *) _gethostbyname(Name);

char *Name;

Arguments

Name	
VMS Usage:	**host_name**
type:	**ASCIZ string**
access:	**read only**
mechanism:	**by reference**

A C-language string containing the name of the host to look up.

Returns

If **gethostbyname()** succeeds, it returns a pointer to a structure of type **hostent**. If this function fails, a value of 0 is returned, and the global variable **h_errno** is set to one of the DNS Name Server error codes defined in the file `IP$root:[IP.include]netdb.h`.

The hostent structure is defined as follows:

```
struct  hostent {
 char   *h_name;         /* official name */
 char   **h_aliases;     /* alias list */
 int    h_addrtype;      /* host address type */
 int    h_length;        /* length of address */
 char   **h_addr_list;   /* list of addresses */
#define h_addr  h_addr_list[0]  /* address, for compat */
```

```
 char   *h_cputype;      /* cpu type */
 char   *h_opsys;        /* operating system */
 char   **h_protos;      /* protocols */
 struct  sockaddr *h_addresses; /* sockaddr form */
};
```

gethostbysockaddr()/gethostbysockaddr_44()

gethostbysockaddr()/gethostbysockaddr_44() — Looks up a host by socket address in the binary host table or the DNS Name Server and returns information about that host. An alternate entry point **_gethostbysockaddr()**, that looks only in the binary host table, is also available. **gethostbysockaddr()** is identical in functionality to **gethostbyaddr()**, but takes its arguments in a different form. The VSI TCP/IP socket library is not reentrant. If you call into it from an AST (interrupt) routine, the results are unpredictable. The **gethostbysockaddr_44()** function is the BSD 4.4 sockaddr variant of this call. This call is used automatically when `IP$ROOT:[IP.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

Format

(struct hostent *) gethostbysockaddr(Addr, Length);

struct sockaddr *Addr;

unsigned int Length;

Arguments

Addr	
VMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**read only**
mechanism:	**by reference**

A pointer to a **sockaddr** structure describing the address to look up.

Length	
VMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The size, in bytes, of the **sockaddr** structure pointed to by **Addr**.

Returns

If **gethostbysockaddr()** succeeds, it returns a pointer to a structure of type **hostent**. (See **int sctp_getaddrlen (int family)** for more information on the **hostent** structure.) If this function fails, a value of 0 is returned, and the global variable **h_errno** is set to one of the DNS Name Server error codes defined in the file `IP$root:[IP.include]netdb.h`.

gethostname()

gethostname() — Returns the Internet name of the host it is executed on. This name comes from the logical name `IP$HOST_NAME` and can be set using the SET HOST-NAME command in the VSI TCP/IP Network Configuration utility (NET-CONFIG).

Format

Status = gethostname(String, Length);

char *String;

unsigned int Length;

Arguments

String	
VMS Usage:	**hostname**
type:	**ASCIZ string**
access:	**write only**
mechanism:	**by reference**

A pointer to a buffer to receive the host name.

Length	
VMS Usage:	**hostname_length**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The length of the buffer, in bytes. The buffer should be at least 33 bytes long to guarantee that the complete host name is returned.

Returns

If the **gethostname()** function is successful, it returns a 0. It returns a -1 if it is unable to translate the logical name.

getnetbyaddr()

getnetbyaddr() — Looks up a network by its network number in the binary host table or the DNS Name Server and returns information about that network. An alternate entry point **_getnetbyaddr()**, that looks only in the binary host table, is also available.

Format

(struct netent *) getnetbyaddr(Net, Protocol);

(struct netent *) _getnetbyaddr(Net, Protocol);

unsigned int Net, Protocol;

Arguments

Net	
VMS Usage:	**network_number**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The network number to look up.

Protocol	
VMS Usage:	**protocol_number**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The address family of the network to look up. For Internet networking, this should be specified as AF_INET.

Returns

If **getnetbyaddr()** succeeds, it returns a pointer to a structure of type **netent**. (See **int sctp_getaddrlen (int family)** for more information on the netent structure.) If this function fails, a value of 0 is returned, and the global variable **h_errno** is set to one of the DNS Name Server error codes defined in `IP$root:[IP.include]netdb.h`.

getnetbyname()

getnetbyname() — Looks up a network by name in the binary host table or the DNS Name Server and returns information about that network. An alternate entry point **_getnetbyname()**, that looks only in the binary host table, is also available.

Format

(struct netent *) getnetbyname(Name);

(struct netent *) _getnetbyname(Name);

char *Name;

Arguments

Name	
VMS Usage:	**network_name**
type:	**ASCIZ string**
access:	**read only**
mechanism:	**by reference**

A pointer to a C-language string containing the name of the network.

Returns

If **getnetbyname()** succeeds, it returns a pointer to a structure of type **netent**. If this function fails, a value of 0 is returned, and the global variable **h_errno** is set to one of the DNS Name Server error codes defined in `IP$root:[IP.include]netdb.h`.

The **netent** structure is defined as follows:

```
struct  netent {
    char             *n_name;       /* official name */
    char            **n_aliases;    /* alias list */
    int               n_addrtype;   /* address type */
    unsigned long     n_net;        /* network # */
    struct  sockaddr *n_addresses;  /* sockaddr form */
};
```

getpeername()/getpeername_44()

getpeername()/getpeername_44() — Returns the name of the peer connected to the specified socket. The **accept_44()** function is the BSD 4.4 sockaddr variant of this call. This call is used automatically when `IP$ROOT:[IP.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

Format

Status = getpeername(VMS_Channel, Address, AddrLen);

short VMS_Channel;

struct sockaddr *Address;

unsigned int *AddrLen;

Arguments

VMS_Channel	
VMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Address	
VMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**write only**
mechanism:	**by reference**

A result parameter. This argument is filled in with the address of the peer, as known to the communications layer. The exact format of the **Address** argument is determined by the domain in which the communication is occurring.

AddrLen	
VMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	by reference

On entry, contains the length of the space pointed to by **Address**, in bytes. On return, it contains the actual length, in bytes, of the address returned.

Returns

If the **getpeername()** is successful, a value of 0 is returned. If an error occurs, a value of -1 is returned, and a more specific message is returned in the global variables **socket_errno** and **vmserrno**.

getprotobyname()

getprotobyname() — Looks up a protocol by name in the binary host table and returns information about that protocol.

Format

(struct protoent *) getprotobyname(Name);

char *Name;

Arguments

Name	
VMS Usage:	**protocol_name**
type:	**ASCIZ string**
access:	**read only**
mechanism:	**by reference**

A pointer to a C-language string containing the name of the protocol.

Returns

If **getprotobyname()** succeeds, it returns a pointer to a structure of type **protoent**. If this function fails, a value of 0 is returned.

The **protoent** structure is defined as follows:

```
struct  protoent {
 char    *p_name;     /* official protocol name */
 char   **p_aliases;  /* alias list */
 int      p_proto;    /* protocol # */
};
```

getprotobynumber()

getprotobynumber() — Looks up a protocol by number in the binary host table and returns information about that protocol.

Format

(struct protoent *) getprotobynumber(Number);

unsigned int Number;

Arguments

Number	
VMS Usage:	**protocol_number**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The numeric value of the protocol.

Returns

If **getprotobynumber()** succeeds, it returns a pointer to a structure of type **protoent**. If this function fails, a value of 0 is returned.

The **protoent** structure is defined as follows:

```
struct  protoent {
 char    *p_name;     /* official protocol name */
 char   **p_aliases;  /* alias list */
 int      p_proto;    /* protocol # */
};
```

getprotoent()

getprotoent() — Returns the next protocol entry from the binary host table. It is used with **setprotoent()** and **endprotoent()** to scan through the protocol table. The scan is initialized with **setprotoent()**, run by calling **getprotoent()** until it returns a 0, and terminated by calling **endprotoent()**.

Format

(struct protoent *) getprotoent();

Returns

The **getprotoent()** function returns either a 0, indicating that there are no more entries, or a pointer to a structure of type protoent.

The **protoent** structure is defined as follows:

```
struct  protoent {
```

```
 char   *p_name;       /* official protocol name */
 char   **p_aliases;   /* alias list */
 int    p_proto;       /* protocol # */
};
```

getservbyname()

getservbyname() — Looks up a service by name in the binary host table and returns information about that service. The service must be present in the `HOSTS.SERVICES` or `HOSTS.LOCAL` file, and the host table must be compiled into binary form using the host table compiler. See the *VSI TCP/IP Administrator's Guide: Volume II* for more information about editing and compiling the host table files.

Format

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

char *Name, *Protocol;

Arguments

Name	
VMS Usage:	**service_name**
type:	**ASCIZ string**
access:	**read only**
mechanism:	**by reference**

A pointer to a C-language string containing the name of the service.

Protocol	
VMS Usage:	**protocol_name**
type:	**ASCIZ string**
access:	**read only**
mechanism:	**by reference**

A pointer to a C-language string containing the name of the protocol associated with the service, such as "TCP".

Returns

If **getservbyname()** succeeds, it returns a pointer to a structure of type **servent**. If this function fails, a value of 0 is returned.

The **servent** structure is defined as follows:

```
struct  servent {
 char    *s_name;        /* official service name */
 char   **s_aliases;     /* alias list */
 int      s_port;        /* port # */
 char    *s_proto;       /* protocol to use */
};
```

getservbyport()

getservbyport() — Looks up a service by protocol port in the binary host table and returns information about that service. The service must be present in the `HOSTS.SERVICES` or `HOSTS.LOCAL` file, and the host table must be compiled into binary form using the host table compiler. See the *VSI TCP/IP Administrator's Guide: Volume II* for more information about editing and compiling the host table files.

Format

(struct servent *) getservbyport(Number, Protocol);

unsigned int Number;

char *Protocol;

Arguments

Number	
VMS Usage:	**service_number**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The numeric value of the service port.

Protocol	
VMS Usage:	**protocol_name**
type:	**ASCIZ string**
access:	**read only**
mechanism:	**by reference**

A pointer to a C-language string containing the name of the protocol associated with the service, such as "TCP".

Returns

If **getservbyport()** succeeds, it returns a pointer to a structure of type **servent**. (See **int sctp_getaddrlen (int family)** for the format of the **servent** structure.) If this function fails, a value of 0 is returned.

getservent()

getservent() — Returns the next server entry from the binary host table. This function is used with **setservent()** and **endservent()** to scan through the service table. The scan is initialized with **setservent()**, run by calling **getservent()** until it returns a 0, and terminated by calling **endservent()**.

Format

(struct servent *) getservent();

Returns

If **getservent()** succeeds, it returns a pointer to a structure of type **servent**. (See **int sctp_getaddrlen (int family)** for the format of the **servent** structure.) If this function fails, a value of 0 is returned.

getsockname()/getsockname_44()

getsockname()/getsockname_44() — Returns the current name of the specified socket. The **getsockname_44()** function is the BSD 4.4 sockaddr variant of this call. This call is used automatically when IP$ROOT:[IP.INCLUDE.NETINET]IN.H is used and the program is compiled with USE_BSD44_ENTRIES defined.

Format

Status = getsockname(VMS_Channel, Address, AddrLen);

short VMS_Channel;

struct sockaddr *Address;

unsigned int *AddrLen;

Arguments

VMS_Channel	
VMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Address	
VMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**write only**
mechanism:	**by reference**

A result parameter. It is filled in with the address of the local socket, as known to the communications layer. The exact format of the **Address** argument is determined by the domain in which the communication is occurring.

AddrLen	
VMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

On entry, contains the length of the space pointed to by **Address**, in bytes. On return, it contains the actual length, in bytes, of the address returned.

Returns

If **getsockname()** is successful, a ,value of 0 is returned. If an error occurs, a value of -1 is returned and a more specific message is returned in the global variables **socket_errno** and **vmserrno**.

getsockopt()

getsockopt() — Retrieves the options associated with a socket. Options can exist at multiple protocol levels; however, they are always present at the uppermost socket level. When manipulating socket options, you must specify the level at which the option resides and the name of the option. To manipulate options at the socket level, specify *Level* as *SOL_SOCKET*. To manipulate options at any other level, specify the protocol number of the appropriate protocol controlling the option. For example, to indicate that an option will be interpreted by the TCP protocol, set *Level* to the protocol number of TCP, which can be determined by calling **getprotobyname()**. **OptName** and any specified options are passed without modification to the appropriate protocol module for interpretation. The include file `IP$root:[IP.include.sys]socket.h` contains definitions for socket-level options. Options at other protocol levels vary in format and name. For more information on what socket options may be retrieved with **getsockopt()**, see **setsockopt()**.

Format

Status = getsockopt(VMS_Channel, Level, OptName, OptVal, OptLen);

short VMS_Channel;

unsigned int Level, OptName, *OptLen;

char *OptVal;

Arguments

VMS_Channel	
VMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Level	
VMS Usage:	**option_level**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The protocol level at which the option will be manipulated. Specify *Level* as *SOL_SOCKET*, or as a protocol number as returned by **getprotobyname()**.

OptName	
VMS Usage:	**option_name**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The option to be manipulated.

OptVal	
VMS Usage:	**dependent on OptName**
type:	**byte buffer**
access:	**write only**
mechanism:	by reference

A pointer to a buffer that will receive the current value of the option. The format of this buffer is dependent on the option requested.

OptLen	
VMS Usage:	**option_length**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

On entry, contains the length of the space pointed to by **OptVal**, in bytes. On return, it contains the actual length, in bytes, of the option returned.

Returns

If the **getsockopt()** is successful, a value of 0 is returned. If an error occurs, a value of -1 is returned, and a more specific message is returned in the global variables **socket_errno** and **vmserrno**.

gettimeofday()

gettimeofday() — Returns the current time of day in UNIX format. This is the number of seconds and microseconds elapsed since January 1, 1970. **gettimeofday()** is part of the 4.3BSD run-time library, but is not provided by VSI as part of the C run-time library. It is provided here for compatibility with the 4.3BSD library.

Format

Status = gettimeofday(TimeVal);

struct timeval *TimeVal;

Arguments

TimeVal	
VMS Usage:	**UNIX_time**

type:	**struct timeval**
access:	**write only**
mechanism:	**by reference**

A pointer to a structure that receives the current time. The timeval structure is defined as follows:

```
struct timeval {
 long    tv_sec;     /* seconds */
 long    tv_usec;    /* and microseconds */
};
```

Returns

The **gettimeofday()** function always returns a value of 0, which indicates it was successful.

hostalias()

hostalias() — Examines the user-specific host alias table (if the user has set one by defining the `IP $HOSTALIASES` logical name) to see if the specified host name is a valid alias for another host name. This is normally called by **gethostbyname()** and **res_search()** automatically.

Format

(char *) hostalias(Name);

char *Name;

Arguments

Name	
VMS Usage:	**host_name**
type:	**ASCIZ string**
access:	**read only**
mechanism:	**by reference**

A C-language string containing the name of the host to look up in the host alias table.

Returns

If successful, the **hostalias()** function returns a pointer to the character string of the canonical name of the host. Otherwise, it returns a 0 to indicate that no alias exists.

htonl()

htonl() — Swaps the byte order of a four-byte integer from OpenVMS byte order to network byte order. This allows you to develop programs that are independent of the hardware architecture on which they are running.

Format

RetVal = htonl(Val);

unsigned long Val;

Arguments

Val	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The four-byte integer to convert to network byte order.

Returns

The **htonl()** function returns the byte-swapped integer that corresponds to *Val*. For example, if *Val* is 0xc029e401, the returned value is 0x01e429c0.

htons()

htons() — Swaps the byte order of a two-byte integer from OpenVMS byte order to network byte order. This allows you to develop programs that are independent of the hardware architecture on which they are running.

Format

RetVal = htons(Val);

unsigned short Val;

Arguments

Val	
VMS Usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**read only**
mechanism:	**by value**

The two-byte integer to convert to network byte order.

Returns

The **htons()** function returns the byte-swapped integer that corresponds to *Val*. For example, if *Val* is 0x0017, the returned value is 0x1700.

inet_addr()

inet_addr() — Converts Internet addresses represented in the ASCII form "xx.yy.zz.ww" to a binary representation in network byte order.

Format

RetVal = inet_addr(Address);

char *Address;

Arguments

Address	
VMS Usage:	**internet_address_string**
type:	**ASCIZ string**
access:	**read only**
mechanism:	**by reference**

A pointer to a C-language string containing an ASCII representation of the Internet address to convert.

Returns

If successful, the **inet_addr()** function returns an integer corresponding to the binary representation of the Internet address in network byte order. It returns a -1 to indicate that it could not parse the specified **Address** string.

inet_lnaof()

inet_lnaof() — Returns the local network address portion of the specified Internet address. For example, the class A address 0x0a050010 (10.5.0.16) is returned as 0x00050010 (5.0.16).

Format

RetVal = inet_lnaof(Address);

struct in_addr Address;

Arguments

Address	
VMS Usage:	**internet_address**
type:	**struct in_addr**
access:	**read only**
mechanism:	**by value**

The Internet address from which to extract the local network address portion. The Internet address is specified in network byte order.

Returns

The **inet_lnaof()** function returns the local network address portion of the Internet address in OpenVMS byte order.

inet_makeaddr()

inet_makeaddr() — Builds a complete Internet address from the separate host and network portions.

Format

RetVal = inet_makeaddr(Network, Host);

unsigned int Network, Host;

Arguments

Network	
VMS Usage:	**network_number**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The network portion of the Internet address to be constructed. The network portion is specified in OpenVMS byte order.

Host	
VMS Usage:	**host_number**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The host portion of the Internet address to be constructed. The host portion is specified in OpenVMS byte order.

Returns

The **inet_makeaddr()** function returns the complete Internet address in network byte order.

inet_netof()

inet_netof() — Returns the network number portion of the specified Internet address. For example, the class A address 0x0a050010 (10.5.0.16) is returned as 0x0a (10).

Format

RetVal = inet_netof(Address);

struct in_addr Address;

Arguments

Address	

VMS Usage:	**internet_address**
type:	**struct in_addr**
access:	**read only**
mechanism:	**by value**

The Internet address from which to extract the network number portion. The Internet address is specified in network byte order.

Returns

The **inet_netof()** routine returns the network portion of the Internet address in OpenVMS byte order.

inet_network()

inet_network() — Interprets Internet network numbers represented in the ASCII form "xx", "xx.yy", or "xx.yy.zz", and converts them into a binary representation in OpenVMS byte order.

Format

RetVal = inet_network(Address);

char *Address;

Arguments

Address	
VMS Usage:	**network_address_string**
type:	**ASCIZ string**
access:	**read only**
mechanism:	**by reference**

A pointer to a C-language string containing an ASCII representation of the Internet network number to convert.

Returns

If successful, the **inet_network()** function returns an integer corresponding to the binary representation of the Internet network in OpenVMS byte order. It returns a -1 to indicate that it could not parse the specified string.

inet_ntoa()

inet_ntoa() — Converts an Internet address represented in binary form into an ASCII string suitable for printing.

Format

(char *) inet_ntoa(Address);

struct in_addr Address;

Arguments

Address	
VMS Usage:	**internet_address**
type:	**struct in_addr**
access:	**read only**
mechanism:	**by value**

The Internet address in binary form. The Internet address is specified in network byte order.

Returns

The **inet_ntoa()** function returns a pointer to a C- language string corresponding to the Internet address.

klread()

klread() — Used with **klseek()** and **ip_kernel_nlist()** to emulate the UNIX 4.3BSD **nlist()** function and the reading of the **/dev/kmem** device. **klread()** and **klseek()** read OpenVMS kernel memory through an interface that is similar to using **read()** and **lseek()** on the **/dev/kmem** device. The OpenVMS CMKRNL privilege is required to use **klread()**. Before calling **klread()**, specify the address to read from using **klseek()**.

Format

Status = klread(Buffer, Size);

char *Buffer;

unsigned int Size;

Arguments

Buffer	
VMS Usage:	**arbitrary**
type:	**byte buffer**
access:	**write only**
mechanism:	**by reference**

The address to which to return the kernel memory.

Size	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The number of bytes to read.

Returns

If successful, the **klread()** function returns the number of bytes read. It returns a -1 to indicate that it failed because the kernel memory was not readable. This usually indicates that the current position, as set by **klseek()**, is invalid.

klseek()

klseek() — Used with **klread()** and **ip_kernel_nlist()** to emulate the UNIX 4.3BSD **nlist()** function and reading the **/dev/kmem** device. **klread()** and **klseek()** read OpenVMS kernel memory through an interface that is similar to using **read()** and **lseek()** on the **/dev/kmem** device. Use **klseek()** to set the current position in the network kernel. This position will be used by **klread()** and **klwrite()** in the next attempt to read or write data.

Format

Status = klseek(Position);

unsigned int Position;

Arguments

Position	
VMS Usage:	**kernel_address**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The address in the network kernel to make the current position for the next **klread()** or **klwrite()** call.

Returns

The **klseek()** routine returns the current position as a success status.

klwrite()

klwrite() — Used with **klseek()** and **ip_kernel_nlist()** to emulate the UNIX 4.3BSD **nlist()** and writing the **/dev/kmem** device. **klwrite()** and **klseek()** write OpenVMS kernel memory through an interface that is similiar to using **write()** and **lseek()** on the **/dev/kmem** device. The OpenVMS CMKRNL privilege is required to use **klwrite()**. Before calling **klwrite()**, specify the address to write using **klseek()**.

Format

Status = klwrite(Buffer, Size);

char *Buffer;

unsigned int Size;

Arguments

Buffer	
VMS Usage:	**arbitrary**
type:	**byte buffer**
access:	**read only**
mechanism:	**by reference**

The address of the data to write into kernel memory.

Size	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The number of bytes to write.

Returns

If successful, the **klwrite()** function returns the number of bytes written. It returns a -1 to indicate that it failed because the kernel memory was not writable. This usually indicates that the current position, as set by **klseek()**, is invalid.

listen()

listen() — Specifies the number of incoming connections that may be queued waiting to be accepted. This backlog must be specified before accepting a connection on a socket. The **listen()** function applies only to sockets of type **SOCK_STREAM**.

Format

Status = listen(VMS_Channel, Backlog);

short VMS_Channel;

unsigned int Backlog;

Arguments

VMS_Channel	
VMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Backlog	
VMS Usage:	**connection_backlog**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The maximum length of the queue of pending connections. If a connection request arrives when the queue is full, the request is ignored. The backlog queue length is limited to 5.

Returns

If **listen()** is successful, a value of 0 is returned. If an error occurs, a value of -1 is returned, and a more specific message is returned in the global variables **socket_errno** and **vmserrno**.

ip_kernel_nlist

ip_kernel_nlist — A special version of the UNIX 4.3BSD **nlist()** function that reads the symbol table to the VSI TCP/IP kernel. Unlike the UNIX 4.3BSD kernel, the VSI TCP/IP kernel's symbol table must be relocated before you can use **klseek()**, **klread()**, or **klwrite()** to examine the networking kernel. Many of the same kernel symbols available under 4.3BSD are also available under the VSI TCP/IP software. Use of this interface is unsupported, as the symbol names and data types may change in future releases of the Berkeley TCP/IP networking code and in future releases of the VSI TCP/IP software. To access the symbol table to the VSI TCP/IP image that is currently running, read from the file indicated by the logical name `IP$NETWORK_IMAGE:`. For more information about how to use **ip_kernel nlist()**, see **nlist()**.

Format

ip_kernel_nlist

nlist()

nlist() — Examines the symbol table in an executable image or symbol table file.

Format

Status = nlist(Filename, nl);

char *Filename;

struct nlist nl[];

Arguments

Filename	
VMS Usage:	**filename**
type:	**ASCIZ string**
access:	**read only**
mechanism:	**by reference**

The file name of the executable image or symbol table file to read.

nl	
VMS Usage:	**symbol_table_info**
type:	**array of struct nlist**
access:	**modify**
mechanism:	**by reference**

An array of **nlist** structures. The **n_name** field of each element specifies the name of the symbol to look up; the array is terminated by a null name. Each symbol is looked up in the file. If the symbol is found, the **n_type** and **n_value** fields are filled in with the type and value of the symbol. Otherwise, they are set to 0.

Returns

If successful, the **nlist()** function returns a 0. Otherwise, it returns a -1.

ntohl()

ntohl() — Swaps the byte order of a four-byte integer from network byte order to OpenVMS byte order. This allows you to develop programs that are independent of the hardware architecture on which they are running.

Format

RetVal = ntohl(Val);

unsigned long Val;

Arguments

Val	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The four-byte integer to convert to OpenVMS byte order.

Returns

The **ntohl()** function returns the byte-swapped integer that corresponds to **Val**. For example, if **Val** is 0x01e429c0, the returned value is 0xc029e401.

ntohs()

ntohs() — Swaps the byte order of a two-byte integer from network byte order to OpenVMS byte order. This allows you to develop programs that are independent of the hardware architecture on which they are running.

Format

RetVal = ntohs(Val);

unsigned short Val;

Arguments

Val	
VMS Usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**read only**
mechanism:	**by value**

The two-byte integer to convert to OpenVMS byte order.

Returns

The **ntohs()** function returns the byte-swapped integer that corresponds to **Val**. For example, if **Val** is 0x1700, the returned value is 0x0017.

recv()/recv_44()

recv()/recv_44() — Receives messages from a socket. This function is equivalent to a **recvfrom()** function called with the **From** and **FromLen** arguments specified as zero. The **socket_read()** function is equivalent to a **recv()** function called with the **Flags** argument specified as zero. The length of the message received is returned as the status. If a message is too long to fit in the supplied buffer and the socket is type SOCK_DGRAM, excess bytes are discarded. If no messages are at the socket, the receive function waits for a message to arrive, unless the socket is non-blocking (see **socket ioctl FIONBIO**). In this case, a status of -1 is returned and the global variable **socket_errno** is set to EWOULDBLOCK. The **recv_44()** function is the BSD 4.4 sockaddr variant of this call. This call is used automatically when `IP$ROOT:[IP.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

Format

Status = int recv (short VMS_Channel, char *Buffer, int Size, int Flags);

Arguments

VMS_Channel	
VMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Buffer	

VMS Usage:	arbitrary
type:	byte buffer
access:	write only
mechanism:	by reference

The address of a buffer in which to place the data read.

Size	
VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by value**

The length of the buffer specified by **Buffer**. The actual number of bytes read is returned in the **Status**.

Flags	
VMS Usage:	**mask_word**
type:	**word (unsigned)**
access:	**read only**
mechanism:	**by value**

Control information that affects the **recv()** function. The **Flags** argument is formed by ORing one or more of the following values:

```
#define   MSG_OOB    0x1    /* process out-of-band data */
#define   MSG_PEEK   0x2    /* peek at incoming message */
```

The MSG_OOB flag causes **recv()** to read any out-of-band data that has arrived on the socket.

The MSG_PEEK flag causes **recv()** to read the data present in the socket without removing the data. This allows the caller to view the data, but leaves it in the socket for future **recv()** calls.

Returns

If **recv()** is successful, a count of the number of characters received is returned. A return value of 0 indicates an end-of-file; that is, the connection has been closed. A return value of -1 indicates an error occurred. A more specific message is returned in the global variables **socket_errno** and **vmserrno**.

recvfrom()recvfrom_44()

recvfrom()recvfrom_44() — Receives messages from a socket. This function is equivalent to the **recv()** function, but takes two additional arguments that allow the caller to determine the remote address from which the message was received. The length of the message received is returned as the status. If a message is too long to fit in the supplied buffer and the socket is type SOCK_DGRAM, excess bytes are discarded. If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking (see **socket ioctl FIONBIO**). In this case, a status of -1 is returned and the global variable **socket_errno** is set to EWOULDBLOCK. The **recvfrom_44()** function is the BSD 4.4 sockaddr variant of this call. This call is used automatically

when `IP$ROOT:[IP.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

Format

Status = int recvfrom (short VMS_Channel, char *Buffer, int Size, int Flags, struct sockaddr *From, unsigned int *FromLen);

Arguments

VMS_Channel	
VMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Buffer	
VMS Usage:	**arbitrary**
type:	**byte buffer**
access:	**write only**
mechanism:	**by reference**

The address of a buffer in which to place the data read.

Size	
VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by value**

The length of the buffer specified by **Buffer**. The actual number of bytes read is returned in the **Status**.

Flags	
VMS Usage:	**mask_word**
type:	**word (unsigned)**
access:	**read only**
mechanism:	**by value**

Control information that affects the **recvfrom()** function. The **Flags** argument is formed by ORing one or more of the following values:

```
#define   MSG_OOB   0x1  /* process out-of-band data */
#define   MSG_PEEK  0x2  /* peek at incoming message */
```

The **MSG_OOB** flag causes **recvfrom()** to read any out-of-band data that has arrived on the socket.

The **MSG_PEEK** flag causes **recvfrom()** to read the data present in the socket without removing the data. This allows the caller to view the data, but leaves it in the socket for future **recvfrom()** calls.

From	
VMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**write only**
mechanism:	**by reference**

On return, this optional argument is filled in with the address of the host that transmitted the packet, as known to the communications layer. The exact format of the **Address** argument is determined by the domain in which the communication is occurring.

FromLen	
VMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

On entry, this optional argument contains the length of the space pointed to by **From**, in bytes. On return, it contains the actual length, in bytes, of the address returned.

Returns

If **recvfrom()** is successful, a count of the number of characters received is returned. A return value of 0 indicates an end-of-file condition; that is, the connection has been closed. If an error occurs, a value of -1 is returned, and a more specific message is returned in the global variables **socket_errno** and **vmserrno**.

recvmsg()/recvmsg_44()

recvmsg()/recvmsg_44() — Receives messages from a socket. This function is equivalent to the **recvfrom()** function, but takes its arguments in a different fashion and can receive into noncontiguous buffers. The length of the message received is returned as the status. If a message is too long to fit in the supplied buffer and the socket is type SOCK_DGRAM, excess bytes are discarded. If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking (see **socket ioctl FIONBIO**). In this case, a status of -1 is returned and the global variable **socket_errno** is set to EWOULDBLOCK. The **recvmsg_44()** function is the BSD 4.4 sockaddr variant of this call. This call is used automatically when `IP$ROOT:[IP.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

Format

Status = recvmsg(VMS_Channel, Message, Flags);

short VMS_Channel;

struct msghdr *Message;

unsigned int Flags;

Arguments

VMS_Channel	
VMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Message	
VMS Usage:	**message header**
type:	**struct msghdr**
access:	**read only**
mechanism:	**by reference**

A pointer to a "msghdr" structure that describes the buffer to be received into. The access rights portion of the structure is unused.

Flags	
VMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Control information that affects the **recvmsg()** function. The **Flags** argument is formed by ORing one or more of the following values:

```
#define   MSG_OOB   0x1  /* process out-of-band data */
#define   MSG_PEEK  0x2  /* peek at incoming message */
```

The **MSG_OOB** flag causes **recvmsg()** to read any out-of-band data that has arrived on the socket.

The **MSG_PEEK** flag causes **recvmsg()** to read the data present in the socket without removing the data. This allows the caller to view the data, but leaves it in the socket for future **recvmsg()** calls.

Returns

If **recvmsg()** is successful, a count of the number of characters received is returned. A return value of 0 indicates an end-of-file condition; that is, the connection has been closed. If an error occurs, a value of -1 is returned, and a more specific message is returned in the global variables **socket_errno** and **vmserrno**.

select()

select() — Examines the OpenVMS Channel sets whose addresses are passed in **ReadFds**, **WriteFds**, and **ExceptFds** to see if some of their Channels are ready for reading, ready for writing, or have an exceptional condition pending. On return, **select()** replaces the given Channel sets with subsets consisting of the Channels that are ready for the requested operation. The total number of ready Channels in all the sets is returned.

Description

The **select()** function is only useful for NETWORK file descriptors and cannot be used for any other OpenVMS I/O device.

The Channel sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such Channel sets: FD_ZERO(&fdset) initializes a Channel set **fdset** to the null set; FD_SET(VMS_Channel, &fdset) includes a particular Channel VMS_Channel in **fdset**; FD_CLR(VMS_Channel, &fdset) removes VMS_Channel from **fdset**; FD_ISSET(VMS_Channel, &fdset) is nonzero if VMS_Channel is a member of **fdset**, otherwise it is zero. The behavior of these macros is undefined if a Channel value is less than zero or greater than or equal to **FD_SETSIZE * CHANNELSIZE**, which is normally at least equal to the maximum number of Channels supported by the system. Make sure that the definition of these macros comes from the VSI TCP/IP `types.h` file, as the definitions differ from the UNIX definitions.

Note

Do not change the value of *`FD_SETSIZE`*. However, if you must change it, make sure its value is equal to the maximum number of channels your system can handle.

The VSI TCP/IP socket library is not reentrant. If you call into it from an AST (interrupt) routine, the results are unpredictable. The **select()** call must not be used while ASTs have been disabled. If the **select()** call is performed with ASTs disabled, the **select()** call will never return and will hang the program from which it was called. Instances when this improper call to **select()** can occur are as follows:

- A call to **select()** is performed within an AST routine (that is, executing an AST routine disables the delivery of other ASTs at the same (user-mode) priority).
- You have explicitly disabled AST delivery in normal (non-AST) code using the `$SETAST` system service.

Format

Status = int select(int Width, fd_set, *ReadFds, fd_set, *WriteFds, fd_set, *ExceptFds,

struct timeval, *Timeout);

FD_SET (VMS_Channel, &fdset)

FD_CLR (VMS_Channel, &fdset)

FD_ISSET (VMS_Channel, &fdset)

FD_ZERO (&fdset)

fd_set fdset;

Arguments

Width	
VMS Usage:	**channel count**

type:	**long (unsigned)**
access:	**read only**
mechanism:	**by value**

The number of bits to be checked in each bit mask that represents a Channel; the Channels from 0 through **Width**-1 in the Channel sets are examined. Typically, **width** has the value returned by **getdtablesize** for the maximum number of Channels.

ReadFds	
VMS Usage:	**channel bitmask**
type:	**struct fd_set**
access:	**modify**
mechanism:	**by reference**

A bit-mask of the Channels that **select()** should test for the ready for reading status. May be specified as a NULL pointer if no Channels are of interest. Selecting true for reading on a Channel on which a **listen()** call has been performed indicates that a subsequent **accept()** call on that Channel will not block.

WriteFds	
VMS Usage:	**channel bitmask**
type:	**struct fd_set**
access:	**modify**
mechanism:	**by reference**

A bit-mask of the Channels that **select()** should test for the ready for writing status. May be specified as a NULL pointer if no Channels are of interest.

ExceptFds	
VMS Usage:	**channel bitmask**
type:	**struct fd_set**
access:	**modify**
mechanism:	**by reference**

A bit-mask of the Channels that **select()** should test for exceptional conditions pending. May be specified as a NULL pointer if no Channels are of interest. Selecting true for exception conditions indicates that out-of-band data is present in the Channel's input buffers.

Timeout	
VMS Usage:	**timeout**
type:	**struct timeval**
access:	**read only**
mechanism:	**by reference**

A maximum interval to wait for the selection to complete. If **Timeout** is a NULL pointer, the select blocks indefinitely. To effect a poll, the **Timeout** argument should be a non-NULL pointer, pointing to a zero-valued timeval structure.

Returns

select() returns the number of ready Channels that are contained in the Channel sets, or -1 if an error occurred. If the time limit expires, **select()** returns 0. If **select()** returns with an error, the Channel sets are unmodified.

select_wake()

select_wake() — Wakes a process waiting in a **select()** call, aborting the **select()** operation. This function may be called from an AST (interrupt) routine, in which case the **select()** call will be aborted when the AST routine completes.

Format

select_wake();

send()/send_44()

send()/send_44() — Transmits a message to another socket. This function is equivalent to a **sendto()** called with the **To** and **ToLen** arguments specified as zero. The **socket_write()** function is equivalent to a **send()** function called with **Flags** specified as zero. Use the **send()** function only when a socket has been connected with **connect()**; however, you can use **sendto()** at any time. If no message space is available at the socket to hold the message to be transmitted, **send()** blocks unless the socket has been placed in non-blocking I/O mode via the socket ioctl FIONBIO. If the socket is type SOCK_DGRAM and the message is too long to pass through the underlying protocol in a single unit, the error EMSGSIZE is returned and the message is not transmitted. The **send_44()** function is the BSD 4.4 sockaddr variant of this call. This call is used automatically when `IP$ROOT:[IP.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

Format

Status = int send (short VMS_Channel, char *Buffer, int Size[, int Flags]);

If Flags are not specified, then 0 (zero) is used.

Arguments

VMS_Channel	
VMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Buffer	
VMS Usage:	**arbitrary**
type:	**byte buffer**

access:	**read only**
mechanism:	**by reference**

The address of a buffer containing the data to send.

Size	
VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by value**

The length of the buffer specified by **Buffer**.

Returns

If the **send()** function is successful, the count of the number of characters sent is returned. If an error occurs, a value of -1 is returned, and a more specific message is returned in the global variables **socket_errno** and **vmserrno**.

sendmsg()/sendmsg_44()

sendmsg()/sendmsg_44() — Transmits a message to another socket. It is equivalent to **sendto()**, but takes its arguments in a different fashion and can send noncontiguous data. If no message space is available at the socket to hold the message to be transmitted, **sendmsg()** blocks unless the socket has been placed in non-blocking I/O mode via the socket ioctl FIONBIO. If the socket is type SOCK_DGRAM and the message is too long to pass through the underlying protocol in a single unit, the error EMSGSIZE is returned and the message is not transmitted. The **sendmsg_44()** function is the BSD 4.4 sockaddr variant of this call. This call is used automatically when `IP$ROOT:[IP.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

Format

Status = sendmsg(VMS_Channel, Message, Flags);

short VMS_Channel;

struct msghdr *Message;

unsigned int Flags;

Arguments

VMS_Channel	
VMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Message	
VMS Usage:	**message header**
type:	**struct msghdr**
access:	**read only**
mechanism:	**by reference**

A pointer to a "msghdr" structure that describes the data to be sent and the address to send it to. The access rights portion of the structure is unused.

Flags	
VMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Control information that affects the **sendto()** function. The Flags argument can be zero or the following:

```
#define MSG_OOB 0x1 /* process out-of-band data */
```

The MSG_OOB flag causes **sendto()** to send out-of-band data on sockets that support this operation (such as SOCK_STREAM).

Returns

If the **sendmsg()** function is successful, the count of the number of characters sent is returned. If an error occurs, a value of -1 is returned, and a more specific message is returned in the global variables **socket_errno** and **vmserrno**.

sendto()/sendto_44

sendto()/sendto_44 — Transmits a message to another socket. It is equivalent to **send()**, but also allows the caller to specify the address to which to send the message. The **sendto()** function can be used on unconnected sockets, while **send()** and **socket_write()** cannot. If no message space is available at the socket to hold the message to be transmitted, **sendto()** blocks unless the socket has been placed in non-blocking I/O mode via the socket ioctl FIONBIO. If the socket is type SOCK_DGRAM and the message is too long to pass through the underlying protocol in a single unit, the error EMSGSIZE is returned and the message is not transmitted. The **sendto_44()** function is the BSD 4.4 sockaddr variant of this call. This call is used automatically when `IP$ROOT:[IP.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

Format

Status = sendto(VMS_Channel, Buffer, Size, Flags, To, ToLen);

short VMS_Channel;

char *Buffer;

int Size;

unsigned short Flags;

struct sockaddr *To;

unsigned int ToLen;

Arguments

VMS_Channel	
VMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Buffer	
VMS Usage:	**arbitrary**
type:	**byte buffer**
access:	**read only**
mechanism:	**by reference**

The address of a buffer containing the data to send.

Size	
VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by value**

The length of the buffer specified by **Buffer**.

Flags	
VMS Usage:	**mask_word**
type:	**word (unsigned)**
access:	**read only**
mechanism:	**by value**

Control information that affects the **sendto()** function. The Flags argument can be zero or the following:

```
#define MSG_OOB 0x1 /* process out-of-band data */
```

The **MSG_OOB** flag causes **sendto()** to send out-of-band data on sockets that support this operation (such as **SOCK_STREAM**).

To	
VMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**read only**
mechanism:	**by reference**

This optional argument is a pointer to the address to which the packet should be transmitted. The exact format of the **Address** argument is determined by the domain in which the communication is occurring.

ToLen	
VMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

This optional argument contains the length of the address pointed to by the **To** argument.

Returns

If the **sendto()** function is successful, the count of the number of characters sent is returned. If an error occurs, a value of -1 is returned, and a more specific message is returned in the global variables **socket_errno** and **vmserrno**.

sethostent()

sethostent() — Initializes the host table and DNS Name Server routines. It is usually unnecessary to call this function because the host table and Name Server routines are initialized automatically when any of the other host table routines are called.

Format

(void) sethostent(StayOpen);

unsigned int StayOpen;

Arguments

StayOpen	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Specifies whether the DNS Name Resolver should use TCP or UDP to communicate with the DNS Name Server. A nonzero value indicates TCP, and a value of 0 (the default if **sethostent()** is not called) indicates UDP.

setnetent()

setnetent() — Initializes the host table and DNS Name Server routines. It is usually unnecessary to call this function because the host table and Name Server routines are initialized automatically when any of the other host table routines are called.

Format

(void) setnetent(StayOpen);

unsigned int StayOpen;

Arguments

StayOpen	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Specifies whether the DNS Name Resolver should use TCP or UDP to communicate with the DNS Name Server. A nonzero value indicates TCP, and a value of 0 (the default if **setnetent()** is not called) indicates UDP.

setprotoent()

setprotoent() — Initializes the host table routines and sets the next protocol entry returned by **getprotoent()** to be the first entry.

Format

(void) setprotoent(StayOpen);

unsigned int StayOpen;

Arguments

StayOpen	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Provided only for compatibility with UNIX 4.3BSD, and is ignored by the VSI TCP/IP software.

setservent()

setservent() — Initializes the host table routines and sets the next service entry returned by **getservent()** to be the first entry.

Format

(void) setservent(StayOpen);

unsigned int StayOpen;

Arguments

StayOpen	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Provided only for compatibility with UNIX 4.3BSD, and is ignored by the VSI TCP/IP software.

setsockopt()

setsockopt() — Manipulates options associated with a socket. Options may exist at multiple protocol levels; however, they are always present at the uppermost socket level. When manipulating socket options, you must specify the level at which the option resides and the name of the option. To manipulate options at the socket level, specify **Level** as SOL_SOCKET. To manipulate options at any other level, specify the protocol number of the appropriate protocol controlling the option. For example, to indicate that an option is to be interpreted by the TCP protocol, set **Level** to the protocol number of TCP; see **getprotobyname()**. **OptName** and any specified options are passed without modification to the appropriate protocol module for interpretation. The include file `IP$root:[IP.include.sys]socket.h` contains definitions for socket-level options. Options at other protocol levels vary in format and name.

Format

Status = setsockopt(VMS_Channel, Level, OptName, OptVal, OptLen);

short VMS_Channel;

unsigned int Level, OptName, OptLen;

char *OptVal;

Arguments

VMS_Channel	
VMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Level	

VMS Usage:	**option_level**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The protocol level at which the option is to be manipulated. **Level** can be specified as SOL_SOCKET, or a protocol number as returned by **getprotobyname()**.

OptName	
VMS Usage:	**option_name**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The option that is to be manipulated.

OptVal	
VMS Usage:	**dependent on OptName**
type:	**byte buffer**
access:	**read only**
mechanism:	**by reference**

A pointer to a buffer that contains the new value of the option. The format of this buffer depends on the option requested.

OptLen	
VMS Usage:	**option_length**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The length of the buffer pointed to by **OptVal**.

Returns

If the **setsockopt()** is successful, a value of 0 is returned. If an error occurs, a value of -1 is returned, and a more specific message is returned in the global variables **socket_errno** and **vmserrno**.

shutdown()

shutdown() — Shuts down all or part of a full-duplex connection on the socket associated with VMS_Channel. This function is usually used to signal an end-of-file to the peer without closing the socket, which would prevent further data from being received.

Format

Status = shutdown(VMS_Channel, How);

short VMS_Channel;

unsigned int How;

Arguments

VMS_Channel	
VMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

How	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Controls which part of the full-duplex connection to shut down. If **How** is 0, further receive operations are disallowed. If **How** is 1, further send operations are disallowed. If **How** is 2, further send and receive operations are disallowed.

Returns

If **shutdown()** is successful, a value of 0 is returned. If an error occurs, a value of -1 is returned, and a more specific error message is returned in the global variables **socket_errno** and **vmserrno**.

socket()

socket() — Creates an end point for communication and returns an OpenVMS channel that describes the end point.

Format

VMS_Channel = socket(Address_Family, Type, Protocol);

short VMS_Channel;

unsigned int Address_Family, Type, Protocol;

Arguments

Address_Family	
VMS Usage:	**address_family**
type:	**longword (unsigned)**
access:	**read only**

mechanism:	**by value**

An address family with which addresses specified in later operations using the socket should be interpreted. The following formats are currently supported; they are defined in the include file `IP $root:[IP.include.sys]socket.h: AF_INET`, Internet (TCP/IP) addresses.

Type	
VMS Usage:	**socket_type**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The semantics of communication using the created socket. The following types are currently defined: `SOCK_STREAM`, `SOCK_DGRAM, SOCK_RAW`.

A SOCK_STREAM socket provides a sequenced, reliable, two-way connection-oriented byte stream with an out- of-band data transmission mechanism. A SOCK_DGRAM socket supports communication by connectionless, unreliable messages of a fixed (typically small) maximum length. SOCK_RAW sockets provide access to internal network interfaces. The type SOCK_RAW is available only to users with SYSPRV privilege.

The **Type** argument, together with the **Address_Family** argument, specifies the protocol to be used. For example, a socket created with AF_INET and SOCK_STREAM is a TCP socket, and a socket created with AF_INET and SOCK_DGRAM is a UDP socket.

Protocol	
VMS Usage:	**protocol_number**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

A particular protocol to be used with the socket. Normally, only a single protocol exists to support a particular socket type using a given address format. However, it is possible that many protocols may exist, in which case a particular protocol must be specified by **Protocol**. The protocol number to use depends on the communication domain in which communication will take place.

For TCP and UDP sockets, the protocol number MUST be specified as 0. For SOCK_RAW sockets, the protocol number should be the value returned by **getprotobyname()**.

Returns

If the **socket()** is successful, an OpenVMS channel is returned. If an error occurs, a value of -1 is returned, and a more specific error message is returned in the global variables **socket_errno** and **vmserrno**.

socket_close()

socket_close() — Deassigns the OpenVMS channel from the VSI TCP/IP INET: device. When the last channel assigned to the device is deassigned, the device and attached socket are deleted. If

the SO_LINGER socket option is set and data remains in the socket's output queue, **socket_close()** deletes only the device. The attached socket remains in the system until the data is sent, after which it is deleted. Once **socket_close()** is called, there is no way to reference this socket. Normally, channels are automatically deassigned at image exit. However, because there is a limit on the number of open channels per process, the **socket_close()** function is necessary for programs that deal with many connections.

Format

Status = socket_close(VMS_Channel);

short VMS_Channel;

Arguments

VMS_Channel	
VMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket to close.

Returns

If the **socket_close()** is successful, a value of 0 is returned. If an error occurs, a value of -1 is returned, and a more specific error message is returned in the global variables **socket_errno** and **vmserrno**.

socket_ioctl()

socket_ioctl() — Performs a variety of functions on the network. In particular, it manipulates socket characteristics, routing tables, ARP tables, and interface characteristics. A **socket_ioctl()** request has encoded in it whether the argument is an input or output parameter, and the size of the argument, in bytes. Macro and define statements used in specifying a **socket_ioctl()** request are located in the file `IP$root:[IP.include.sys]ioctl.h`.

Format

Status = socket_ioctl(VMS_Channel, Request, ArgP);

short VMS_Channel;

unsigned int Request;

char *ArgP;

Arguments

VMS_Channel	
VMS Usage:	**channel**

type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Request	
VMS Usage:	**ioctl_request**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Which **socket_ioctl()** function to perform.

ArgP	
VMS Usage:	**arbitrary**
type:	**byte buffer**
access:	**read, write, or modify depending on Request**
mechanism:	**by reference**

A pointer to a buffer whose format and function depend on the **Request** specified.

Returns

If the **socket_ioctl()** is successful, a value of 0 is returned. If an error occurs, a value of -1 is returned, and a more specific error message is returned in the global variables **socket_errno** and **vmserrno**.

For a list of the **socket_ioctl()** functions supported by VSI TCP/IP, see the following pages.

socket ioctl FIONBIO

socket ioctl FIONBIO — Controls nonblocking I/O on a socket. If nonblocking I/O is enabled and another function is called that would have to wait for a connection, for data to arrive, or for data to be transmitted, the function completes with a -1 error return, and the global variable **socket_errno** is set to EWOULDBLOCK.

Format

Status = socket_ioctl(VMS_Channel, FIONBIO, Enable);

unsigned int *Enable;

Arguments

Enable	
VMS Usage:	**longword_unsigned**

type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

A pointer to an integer that specifies whether nonblocking I/O is enabled or disabled. A value of 1 enables nonblocking I/O, and a value of 0 disables nonblocking I/O. By default, nonblocking I/O is disabled when a socket is created.

socket ioctl FIONREAD

socket ioctl FIONREAD — Retrieves the number of bytes waiting to be read. A return of 0 indicates that no data is buffered.

Format

Status = socket_ioctl(VMS_Channel, FIONREAD, Count);

unsigned int *Count;

Arguments

Count	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

A pointer to an integer buffer that will receive a count of the number of characters waiting to be read.

socket ioctl SIOCADDRT

socket ioctl SIOCADDRT — Adds routing information to the network routing tables. This function does not modify the socket itself, but rather modifies the operation of the network in general. It does not matter what the state of the socket is. Normally, to modify Internet routing tables, you use a socket created with the AF_INET and SOCK_DGRAM arguments.

Format

Status = socket_ioctl(VMS_Channel, SIOCADDRT, Route);

struct rtentry *Route;

Arguments

Route	
VMS Usage:	**routing_entry**
type:	**struct rtentry**

access:	**read only**
mechanism:	**by reference**

A pointer to the address of a **rtentry** structure that describes the route to be added. The **rtentry** structure is defined in `IP$root:[ip.include.net]route.h` as follows:

```
struct rtentry {
     u_long  rt_hash;
   struct  sockaddr rt_dst;
   struct  sockaddr rt_gateway;
   short   rt_flags;
   short   rt_refcnt;
   u_long  rt_use;
   struct  ifnet *rt_ifp;
};
```

Field	Description
rt_hash, **rt_refcnt**, **rt_use**, and **rt_ifp**	Are ignored by SIOCADDRT and should be set to zero.
rt_dst	Specifies the address of the destination host or network.
rt_gateway	Specifies the address of the local gateway to this host or network.

rt_flags	Specifies one or more of the following flags that affect a routing entry:
	
	`#define RTF_UP      0x1  /* route useable */`
	`#define RTF_GATEWAY 0x2  /* destination is a gateway */`
	`#define RTF_HOST    0x4  /* host entry (net otherwise)*/`
	
	RTF_UP — Indicates that the route is usable. It should always be specified.
	
	RTF_GATEWAY — Indicates that the next hop to the destination is a gateway, so that the output routines know to address the gateway rather than the destination directly.
	
	RTF_HOST — Indicates that the address specified in **rt_dst** is an Internet host, rather than an Internet network (the default).

socket ioctl SIOCDELRT

socket ioctl SIOCDELRT — Deletes routing information from the network routing tables. This function does not modify the socket itself, but rather modifies the operation of the network in general. It does not matter what the state of the socket is. Normally, to modify Internet routing tables, you use a socket created with the AF_INET and SOCK_DGRAM arguments. It is impossible to obtain a list of the routes installed via **socket_ioctl()**. To delete a route, you must either know it already exists or use **ip_kernel_nlist()** to read the routing tables directly from the networking kernel.

Format

Status = socket_ioctl(VMS_Channel, SIOCDELRT, Route);

struct rtentry *Route;

Arguments

Route	
VMS Usage:	**routing_entry**
type:	**struct rtentry**
access:	**read only**
mechanism:	**by reference**

A pointer to the address of a **rtentry** structure that describes the route to be deleted. The rtentry structure is defined in `IP$root:[ip.include.net]route.h` as follows:

```
struct rtentry {
   u_long  rt_hash;
   struct  sockaddr rt_dst;
   struct  sockaddr rt_gateway;
   short   rt_flags;
   short   rt_refcnt;
   u_long  rt_use;
   struct  ifnet *rt_ifp;
};
```

Field	**Description**
rt_hash, **rt_refcnt**, **rt_use**, and **rt_ifp**	Are ignored by SIOCDELRT and should be set to zero.
rt_dst	Specifies the address of the destination host or network.
rt_gateway	Specifies the address of the local gateway to this host or network.

rt_flags	Specifies one or more of the following flags that affect a routing entry:
	
	`#define RTF_UP      0x1  /* route useable */`
	`#define RTF_GATEWAY 0x2  /* destination is a gateway */`
	`#define RTF_HOST    0x4  /* host entry (net otherwise) */`
	
	RTF_UP — Indicates that the route is usable. It should always be specified.
	
	RTF_GATEWAY — Indicates that the next hop to the destination is a gateway, so that the output routines know to address the gateway rather than the destination directly.
	
	RTF_HOST — Indicates that the address specified in **rt_dst** is an Internet host, rather than an Internet network (the default).

socket ioctl SIOCATMARK

socket ioctl SIOCATMARK — Retrieves an indication as to whether the next byte in the stream coincides with an out-of-band or URGENT data mark.

Format

Status = socket_ioctl(VMS_Channel, SIOCATMARK, AtMark);

unsigned int *AtMark;

Arguments

AtMark	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

A pointer to an integer buffer that will receive the indication. The buffer is set to 0 if the socket is not at the out-of-band mark. It is set to nonzero if the socket is at the out-of-band mark.

socket ioctl SIOCDARP

socket ioctl SIOCDARP — Deletes an entry from the ARP table. This format is compatible with the UNIX 4.3BSD function of the same name.

Format

Status = socket_ioctl (VMS_Channel, SIOCDARP, ARP_Req);

struct arpreq *ARP_Req;

Arguments

ARP_Req	
VMS Usage:	**arp_request**
type:	**struct arpreq**
access:	**read only**
mechanism:	**by reference**

The address of an **arpreq** structure that contains the protocol address and the hardware address. The **arpreq** structure is defined in `IP$root:[ip.include.net]if_arp.h` as follows:

```
struct arpreq {
        struct  sockaddr arp_pa;                /* protocol address */
        struct  sockaddr arp_ha;                /* hardware address */
        int     arp_flags;                      /* flags */

};
/*  arp_flags and at_flags field values */
#define ATF_INUSE       0x01    /* entry in use */
#define ATF_COM         0x02    /* completed entry (enaddr valid) */
#define ATF_PERM        0x04    /* permanent entry */
```

```
#define ATF_PUBL        0x08    /* publish entry (respond for other host)
*/

#define ATF_USETRAILERS 0x10    /* has requested trailers */
#define ATF_PROXY       0x20    /* Do PROXY arp */
```

The **arp_pa** field is a sockaddr field that is set to the ip address the remote interface uses.

The **arp_ha.sa_data** field is 6 bytes of binary data that represents the Ethernet address of the remote interface.

socket ioctl SIOCGARP

socket ioctl SIOCGARP — Displays an entry in the ARP table. This function is compatible with the UNIX 4.3BSD function of the same name.

Format

Status = socket_ioctl (VMS_Channel, SIOCGARP, ARP_Req);

struct arpreq *ARP_Req;

Arguments

ARP_Req	
VMS Usage:	**arp_request**
type:	**struct arpreq**
access:	**modify**
mechanism:	**by reference**

The address of an **arpreq** structure that contains the protocol address and the hardware address. The **arpreq** structure is defined in `IP$root:[ip.include.net]if_arp.h` as follows:

```
struct arpreq {
        struct  sockaddr arp_pa;                /* protocol address */
        struct  sockaddr arp_ha;                /* hardware address */
        int     arp_flags;                      /* flags */

};
/*  arp_flags and at_flags field values */
#define ATF_INUSE       0x01    /* entry in use */
#define ATF_COM         0x02    /* completed entry (enaddr valid) */
#define ATF_PERM        0x04    /* permanent entry */
#define ATF_PUBL        0x08    /* publish entry (respond for other host)
*/

#define ATF_USETRAILERS 0x10    /* has requested trailers */
#define ATF_PROXY       0x20    /* Do PROXY arp */
```

The **arp_pa** field is a sockaddr field that is set to the ip address the remote interface uses.

The **arp_ha.sa_data** field is 6 bytes of binary data that represents the Ethernet address of the remote interface.

socket ioctl SIOCSARP

socket ioctl SIOCSARP — Adds an entry to the ARP table. This function is compatible with the UNIX 4.3BSD function of the same name.

Format

Status = socket_ioctl (VMS_Channel, SIOCSARP, ARP_Req);

struct arpreq *ARP_Req;

Arguments

ARP_Req	
VMS Usage:	**arp_request**
type:	**struct arpreq**
access:	**read only**
mechanism:	**by reference**

The address of an **arpreq** structure that contains the protocol address and the hardware address. The **arpreq** structure is defined in `IP$root:[ip.include.net]if_arp.h` as follows:

```
struct arpreq {
        struct  sockaddr arp_pa;                /* protocol address */
        struct  sockaddr arp_ha;                /* hardware address */
        int     arp_flags;                      /* flags */

};
/*  arp_flags and at_flags field values */
#define ATF_INUSE       0x01    /* entry in use */
#define ATF_COM         0x02    /* completed entry (enaddr valid) */
#define ATF_PERM        0x04    /* permanent entry */
#define ATF_PUBL        0x08    /* publish entry (respond for other host)
*/

#define ATF_USETRAILERS 0x10    /* has requested trailers */
#define ATF_PROXY       0x20    /* Do PROXY arp */
```

The **arp_pa** field is a sockaddr field that is set to the ip address the remote interface uses.

The **arp_ha.sa_data** field is 6 bytes of binary data that represents the Ethernet address of the remote interface.

socket ioctl SIOCGIFADDR

socket ioctl SIOCGIFADDR — Retrieves the Internet address of a network interface. This function does not modify the socket itself, but rather examines the operation of the network in general. It does not matter what the state of the socket is. Normally, to examine Internet addresses, you use a socket created with the AF_INET and SOCK_DGRAM arguments.

Format

Status = socket_ioctl(VMS_Channel, SIOCGIFADDR, Interface_Req);

struct ifreq *Interface_Req;

Arguments

Interface_Req	
VMS Usage:	**interface_request**
type:	**struct ifreq**
access:	**modify**
mechanism:	**by reference**

The address of an **ifreq** structure that describes the interface from which to retrieve the address. The **ifreq** structure is defined in `IP$root:[ip.include.net]if.h` as follows:

```
struct ifreq {
    char ifr_name[16];
    struct sockaddr ifr_addr;
};
```

The **ifr_name** field is a null-terminated string specifying the name of the interface to be examined, such as "se0".

The **ifr_addr** field is a **sockaddr** structure that is set to the address of the interface.

socket ioctl SIOCSIFADDR

socket ioctl SIOCSIFADDR — Sets the Internet address of a network interface. Normally, this is done using the **IP SET/INTERFACE** command. This function does not modify the socket itself, but rather modifies the operation of the network in general. It does not matter what the state of the socket is. Normally, to modify Internet addresses, you use a socket created with the AF_INET and SOCK_DGRAM arguments.

Format

Status = socket_ioctl(VMS_Channel, SIOCSIFADDR, Interface_Req);

struct ifreq *Interface_Req;

Arguments

Interface_Req	
VMS Usage:	**interface_request**
type:	**struct ifreq**
access:	**read only**
mechanism:	**by reference**

The address of an **ifreq** structure that describes the address to be set. The **ifreq** structure is defined in `IP$root:[ip.include.net]if.h` as follows:

```
struct ifreq {
   char ifr_name[16];
   struct sockaddr ifr_addr;
};
```

The **ifr_name** field is a null-terminated string specifying the name of the interface to be modified, such as "se0".

The **ifr_addr** field is a **sockaddr** structure specifying the address to be set.

socket ioctl SIOCGIFBRDADDR

socket ioctl SIOCGIFBRDADDR — Retrieves the Internet broadcast address of a network interface. This function does not modify the socket itself, but rather examines the operation of the network in general. It does not matter what the state of the socket is. Normally, to examine Internet broadcast addresses, you use a socket created with the AF_INET and SOCK_DGRAM arguments.

Format

Status = socket_ioctl(VMS_Channel, SIOCGIFBRDADDR, Interface_Req);

struct ifreq *Interface_Req;

Arguments

Interface_Req	
VMS Usage:	**interface_request**
type:	**struct ifreq**
access:	**modify**
mechanism:	**by reference**

The address of an **ifreq** structure that describes the interface from which to retrieve the broadcast address. The **ifreq** structure is defined in `IP$root:[ip.include.net]if.h` as follows:

```
struct ifreq {
   char ifr_name[16];
   struct sockaddr ifr_broadaddr;
};
```

The **ifr_name** field is a null-terminated string specifying the name of the interface to be examined, such as "se0".

The **ifr_broadaddr** field is a **sockaddr** structure that is set to the broadcast address of the interface.

socket ioctl SIOCSIFBRDADDR

socket ioctl SIOCSIFBRDADDR — Sets the Internet broadcast address of a network interface. Normally, this is done using the **IP SET/INTERFACE** command. This function does not modify the socket itself, but rather modifies the operation of the network in general. It does not matter what the state of the socket is. Normally, to modify Internet broadcast addresses, you use a socket created with the AF_INET and SOCK_DGRAM arguments.

Format

Status = socket_ioctl(VMS_Channel, SIOCSIFBRDADDR, Interface_Req);

struct ifreq *Interface_Req;

Arguments

Interface_Req	
VMS Usage:	**interface_request**
type:	**struct ifreq**
access:	**read only**
mechanism:	**by reference**

The address of an **ifreq** structure that describes the interface on which to set the broadcast address. The **ifreq** structure is defined in `IP$root:[ip.include.net]if.h` as follows:

```
struct ifreq {
   char ifr_name[16];
   struct sockaddr ifr_broadaddr;
};
```

The **ifr_nam**e field is a null-terminated string specifying the name of the interface to be modified, such as "se0".

The **ifr_broadaddr** field is a **sockaddr** structure specifying the broadcast address to be set.

socket ioctl SIOCGIFCONF

socket ioctl SIOCGIFCONF — Retrieves the list of network interfaces from the networking kernel for further examination by the other SIOCGxxxx functions. This function does not modify the socket itself, but rather examines the operation of the network in general. It does not matter what the state of the socket is. Normally, to examine the network configuration, you use a socket created with the AF_INET and SOCK_DGRAM arguments.

Format

Status = socket_ioctl(VMS_Channel, SIOCGIFCONF, Interface_Config);

struct ifconf *Interface_Config;

Arguments

Interface_Config	
VMS Usage:	**interface_configuration_request**
type:	**struct ifconf**
access:	**modify**
mechanism:	**by reference**

The address of an **ifconf** structure describing a buffer in which to return the interface configuration. The **ifconf** structure is defined in `IP$root:[ip.include.net]if.h` as follows:

```
struct  ifconf {
    int     ifc_len;                /* size of buffer */
    union {
            caddr_t ifcu_buf;
            struct  ifreq *ifcu_req;
```

```
    } ifc_ifcu;
#define ifc_buf ifc_ifcu.ifcu_buf   /* buffer address */
#define ifc_req ifc_ifcu.ifcu_req   /* array of structures */
};
```

The **ifc_len** field should be set to the length of the buffer specified by **ifc_buf**. Upon return, the **ifc_len** field contains the actual number of bytes written into the buffer.

The **ifc_buf** field should be set to a buffer large enough to hold the entire network configuration. Upon return, if VMS_Channel is an AF_INET socket the **ifc_req** buffer contains an array of **ifreq** structures, one for each interface and address. If VMS_Channel is an AF_INET6 socket, then the **ifc_req** buffer contains an array of **ifreq6** structures, one for each address present. The array of **ifreq6** structures may contain both IPv4 and IPv6 addresses.

The following short fragment of C-language code prints all Internet family interfaces and shows how to decode the **ifconf** structure:

```
n = ifc.ifc_len/sizeof(struct ifreq);
for (ifr = ifc.ifc_req; n > 0; n--, ifr++) {
    if (ifr->ifr_addr.sa_family != AF_INET) continue;
   printf("%s\n",ifr->ifr_name);
}
```

The **ifreq6** structure is defined in `IP$root:[ip.include.net]if.h` as follows:

```
struct ifreq6 {
   char ifr_name[16];
   struct sockaddr_in6 ifr_addr;
};
```

socket ioctl SIOCGIFDSTADDR

socket ioctl SIOCGIFDSTADDR — Retrieves the destination Internet address of a point-to-point network interface. This function does not modify the socket itself, but rather examines the operation of the network in general. It does not matter what the state of the socket is. Normally, to examine Internet addresses, you use a socket created with the AF_INET and SOCK_DGRAM arguments.

Format

Status = socket_ioctl(VMS_Channel, SIOCGIFDSTADDR, Interface_Req);

struct ifreq *Interface_Req;

Arguments

Interface_Req	
VMS Usage:	**interface_request**
type:	**struct ifreq**
access:	**modify**
mechanism:	**by reference**

The address of an **ifreq** structure that describes the interface from which to retrieve the destination address. The **ifreq** structure is defined in `IP$root:[ip.include.net]if.h` as follows:

```
struct ifreq {
    char ifr_name[16];
    struct sockaddr ifr_dstaddr;
};
```

The **ifr_name** field is a null-terminated string specifying the name of the interface to be examined, such as "se0".

The **ifr_dstaddr** field is a **sockaddr** structure that is set to the destination address of the interface.

socket ioctl SIOCSIFDSTADDR

socket ioctl SIOCSIFDSTADDR — Sets the destination Internet address of a point-to-point network interface. Normally, this is done using the **IP SET/INTERFACE** command. This function does not modify the socket itself, but rather modifies the operation of the network in general. It does not matter what the state of the socket is. Normally, to modify Internet addresses, you use a socket created with the AF_INET and SOCK_DGRAM arguments.

Format

Status = socket_ioctl(VMS_Channel, SIOCSIFDSTADDR, Interface_Req);

struct ifreq *Interface_Req;

Arguments

Interface_Req	
VMS Usage:	**interface_request**
type:	**struct ifreq**
access:	**read only**
mechanism:	**by reference**

The address of an **ifreq** structure that describes the interface on which to set the destination address. The **ifreq** structure is defined in `IP$root:[ip.include.net]if.h` as follows:

```
struct ifreq {
    char ifr_name[16];
    struct sockaddr ifr_dstaddr;
};
```

The **ifr_name** field is a null-terminated string specifying the name of the interface to be modified, such as "se0".

The **ifr_dstaddr** field is a **sockaddr** structure specifying the destination address to be set.

socket ioctl SIOCGIFFLAGS

socket ioctl SIOCGIFFLAGS — Retrieves various control flags from a network interface. This function does not modify the socket itself, but rather examines the operation of the network in general. It does not matter what the state of the socket is. Normally, to examine interface flags, you use a socket created with the AF_INET and SOCK_DGRAM arguments.

Format

Status = socket_ioctl(VMS_Channel, SIOCSIFFLAGS, Interface_Req);

struct ifreq *Interface_Req;

Arguments

Interface_Req	
VMS Usage:	**interface_request**
type:	**struct ifreq**
access:	**modify**
mechanism:	**by reference**

The address of an **ifreq** structure that describes the state of the flags. The **ifreq** structure is defined in `IP$root:[ip.include.net]if.h` as follows:

```
struct ifreq {
    char ifr_name[16];
    short ifr_flags;
    char Xfill[14];
};
```

The **ifr_name** field is a null-terminated string specifying the name of the interface to be examined, such as "se0".

The **ifr_flag**s field receives the state of the interface flags. The following flag bits are valid:

```
#define IFF_UP          0x1   /* interface is up */
#define IFF_BROADCAST   0x2   /* broadcast address valid */
#define IFF_DEBUG       0x4   /* turn on debugging */
#define IFF_LOOPBACK    0x8   /* is a loopback net */
#define IFF_POINTOPOINT 0x10  /* interface is ptp link */
#define IFF_NOTRAILERS  0x20  /* avoid use of trailers */
#define IFF_RUNNING     0x40  /* resources allocated */
#define IFF_NOARP       0x80  /* no ARP protocol */
```

socket ioctl SIOCSIFFLAGS

socket ioctl SIOCSIFFLAGS — Sets various control flags on a network interface. Normally this is done using the **IP SET /INTERFACE** command. To modify the state of a flag, first call the **SIOCGIFFLAGS socket_ioctl()** function, change whichever bits are necessary, and then reset the flags by calling **SIOCSIFFLAGS socket_ioctl()**. This function does not modify the socket itself, but rather modifies the operation of the network in general. It does not matter what the state of the socket is. Normally, to modify interface flags, you use a socket created with the AF_INET and SOCK_DGRAM arguments.

Format

Status = socket_ioctl(VMS_Channel, SIOCSIFFLAGS, Interface_Req);

struct ifreq *Interface_Req;

Arguments

Interface_Req	
VMS Usage:	**interface_request**
type:	**struct ifreq**
access:	**read only**
mechanism:	**by reference**

The address of an **ifreq** structure that describes the new state of the flags. The **ifreq** structure is defined in `IP$root:[ip.include.net]if.h` as follows:

```
struct ifreq {
   char ifr_name[16];
   short ifr_flags;
   char Xfill[14];
};
```

The **ifr_name** field is a null-terminated string specifying the name of the interface to be modified, such as "se0".

The **ifr_flags** field specifies the new state of the interface flags. The following flags can be set or cleared:

```
#define IFF_UP          0x1     /* interface is up */
#define IFF_DEBUG       0x4     /* turn on debugging */
#define IFF_NOTRAILERS  0x20    /* avoid use of trailers */
#define IFF_NOAR        0x80    /* no ARP protocol */
```

socket ioctl SIOCGIFMETRIC

socket ioctl SIOCGIFMETRIC — Retrieves the network interface metric, or cost. The interface metric is ignored by the VSI TCP/IP software, and is not documented further here.

Format

socket ioctl SIOCGIFMETRIC

socket ioctl SIOCSIFMETRIC

socket ioctl SIOCSIFMETRIC — Sets the network interface metric, or cost. The interface metric is ignored by the VSI TCP/IP software, and is not documented further here.

Format

socket ioctl SIOCSIFMETRIC

socket ioctl SIOCGIFNETMASK

socket ioctl SIOCGIFNETMASK — Retrieves the Internet address mask of a network interface. This function does not modify the socket itself, but rather examines the operation of the network in general. It does not matter what the state of the socket is. Normally, to examine Internet address masks, you use a socket created with the AF_INET and SOCK_DGRAM arguments.

Format

Status = socket_ioctl(VMS_Channel, SIOCGIFNETMASK, Interface_Req);

struct ifreq *Interface_Req;

Arguments

Interface_Req	
VMS Usage:	**interface_request**
type:	**struct ifreq**
access:	**modify**
mechanism:	**by reference**

The address of an **ifreq** structure that describes the interface from which to retrieve the address mask. The **ifreq** structure is defined in `IP$root:[ip.include.net]if.h` as follows:

```
struct ifreq {
    char ifr_name[16];
    struct sockaddr ifr_addr;
};
```

The **ifr_name** field is a null-terminated string specifying the name of the interface to be examined, such as "se0".

The **ifr_addr** field is a **sockaddr** structure that is set to the address mask of the interface.

socket ioctl SIOCSIFNETMASK

socket ioctl SIOCSIFNETMASK — Sets the Internet address mask of a network interface. Normally, this is done using the **IP SET/INTERFACE** command. This function does not modify the socket itself, but rather modifies the operation of the network in general. It does not matter what the state of the socket is. Normally, to modify Internet address masks, you use a socket created with the AF_INET and SOCK_DGRAM arguments.

Format

Status = socket_ioctl(VMS_Channel, SIOCSIFNETMASK, Interface_Req);

struct ifreq *Interface_Req;

Arguments

Interface_Req	
VMS Usage:	**interface_request**
type:	**struct ifreq**
access:	**read only**
mechanism:	**by reference**

The address of an **ifreq** structure that describes the interface on which to set the address mask. The **ifreq** structure is defined in `IP$root:[ip.include.net]if.h` as follows:

```
struct ifreq {
   char ifr_name[16];
   struct sockaddr ifr_addr;
};
```

The **ifr_nam**e field is a null-terminated string specifying the name of the interface to be modified, such as "se0".

The **ifr_addr** field is a **sockaddr** structure specifying the address mask to be set.

socket option SO_BROADCAST

socket option SO_BROADCAST — Enables transmission of broadcast messages on the specified socket.

Format

Status = setsockopt(VMS_Channel, SOL_SOCKET, SO_BROADCAST, On, sizeof(*On));

unsigned int *On;

Arguments

On	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

A pointer to an integer buffer that specifies whether the transmission of broadcast messages is enabled or disabled. A nonzero value enables the transmission of broadcast messages, a value of 0 disables the transmission.

socket option SO_DEBUG

socket option SO_DEBUG — Controls the recording of debugging information by the VSI TCP/IP networking kernel.

Format

Status = setsockopt(VMS_Channel, SOL_SOCKET, SO_DEBUG, On, sizeof(*On));

unsigned int *On;

Arguments

On	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**

mechanism:	**by reference**

A pointer to an integer buffer that specifies whether debugging is enabled or disabled. A nonzero value enables debugging. A value of 0 disables debugging.

socket option SO_DONTROUTE

socket option SO_DONTROUTE — Indicates that outgoing messages bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface, as determined by the network portion of the destination address.

Format

Status = setsockopt(VMS_Channel, SOL_SOCKET, SO_DONTROUTE, On, sizeof(*On));

unsigned int *On;

Arguments

On	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

A pointer to an integer buffer that specifies whether SO_DONTROUTE is enabled or disabled. A nonzero value enables SO_DONTROUTE. A value of 0 disables SO_DONTROUTE.

socket option SO_ERROR

socket option SO_ERROR — Retrieves and clears any error status pending on the socket. This function is only valid with the **getsockopt()** function.

Format

Status = getsockopt(VMS_Channel, SOL_SOCKET, SO_ERROR, Value, Length);

unsigned int *Value;

unsigned int *Length;

Arguments

Value	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

A pointer to an integer buffer that receives the value of **errno** (the error number) that is pending on the socket.

Length	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

On entry, contains the length of the space pointed to by **Value**, in bytes. On return, it contains the actual length, in bytes, of the **Value** returned.

socket option SO_KEEPALIVE

socket option SO_KEEPALIVE — Enables periodic transmission of messages on an idle connected socket. If the connected party fails to respond to these messages, the connection is considered broken and processes using the socket are notified via an error returned by a read. Keepalives are a questionable use of the network in that they cause idle connections to add network traffic by constantly probing their peer. Avoid keepalives if another mechanism is available to detect the loss of a peer, such as timeouts.

Format

Status = setsockopt(VMS_Channel, SOL_SOCKET, SO_KEEPALIVE, On, sizeof(*On));

unsigned int *On;

Arguments

On	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

A pointer to an integer buffer that specifies whether keepalives are enabled or disabled. A nonzero value enables keepalives. A value of 0 disables keepalives.

socket option SO_LINGER

socket option SO_LINGER — Controls the action taken when unsent messages are queued on a socket and a **socket_close()** function call is issued. If the socket promises reliable delivery of data and SO_LINGER is set, **socket_close()** deletes only the device. The attached socket remains in the system until this data is sent or until it determines that it cannot deliver the information (a timeout period, termed the linger interval, is specified in the **setsockopt()** function). Only then is the attached socket deleted.

Format

Status = setsockopt(VMS_Channel, SOL_SOCKET, SO_LINGER, Linger, sizeof(*Linger));

struct linger *Linger;

Arguments

Linger	
VMS Usage:	**linger_structure**
type:	**struct linger**
access:	**read only**
mechanism:	**by reference**

A pointer to a structure describing whether the SO_LINGER option is enabled or disabled.

```
struct  linger {
   int     l_onoff;        /* option on/off */
   int     l_linger;       /* linger time */
};
```

When the **l_onoff** field is nonzero, SO_LINGER is enabled. When it is 0, SO_LINGER is disabled. If SO_LINGER is being enabled, the **l_linger** field specifies the timeout period, in seconds.

socket option SO_OOBINLINE

socket option SO_OOBINLINE — Enables receipt of out-of-band data along with the regular data stream. You can use this option instead of specifying the MSG_OOB flag to the **recv()** or **recvfrom()** functions.

Format

Status = setsockopt(VMS_Channel, SOL_SOCKET, SO_OOBINLINE, On, sizeof(*On));

unsigned int *On;

Arguments

On	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

A pointer to an integer buffer that specifies whether the SO_OOBINLINE option is enabled or disabled. A nonzero value enables SO_OOBINLINE. A value of 0 disables SO_OOBINLINE.

socket option SO_RCVBUF

socket option SO_RCVBUF — Specifies the amount of buffer space that can be used to buffer received data on the socket. The default value is 6144. You can specify this option to raise the TCP window size, increase the maximum size of UDP datagrams that can be received, or increase buffer space in general.

Format

Status = setsockopt(VMS_Channel, SOL_SOCKET, SO_RCVBUF, Value, sizeof(*Value));

unsigned int *Value;

Arguments

Value	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

A pointer to an integer buffer that specifies the new size of the receive buffer, in bytes.

socket option SO_RCVLOWAT

socket option SO_RCVLOWAT — This option exists only for compatibility with UNIX 4.3BSD and has no effect on VSI TCP/IP sockets.

Format

socket option SO_RCVLOWAT

socket option SO_RCVTIMEO

socket option SO_RCVTIMEO — This option exists only for compatibility with UNIX 4.3BSD and has no effect on VSI TCP/IP sockets.

Format

socket option SO_RCVTIMEO

socket option SO_REUSEADDR

socket option SO_REUSEADDR — Specifies how to reuse local addresses. When SO_REUSEADDR is enabled, **bind()** allows a local port number to be used even if sockets using the same local port number already exist, provided that these sockets are connected to a unique remote port. This option allows a server to **bind()** to a socket to listen for new connections, even if connections are already in progress on this port.

Format

Status = setsockopt(VMS_Channel, SOL_SOCKET, SO_REUSEADDR, On, sizeof(*On));

unsigned int *On;

Arguments

On	

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

A pointer to an integer buffer that specifies whether SO_REUSEADDR is enabled or disabled. A nonzero value enables SO_REUSEADDR. A value of 0 disables SO_REUSEADDR.

socket option SO_SNDBUF

socket option SO_SNDBUF — Specifies the amount of buffer space that can be used to buffer transmitted data on the socket. The default value is 6144 for TCP and 2048 for UDP. You can specify this option to raise the TCP window size, increase the maximum size of UDP datagrams that can be transmitted, or increase buffer space in general.

Format

Status = setsockopt(VMS_Channel, SOL_SOCKET, SO_SNDBUF, Value, sizeof(*Value));

unsigned int *Value;

Arguments

Value	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

A pointer to an integer buffer that specifies the new size of the transmit buffer, in bytes.

socket option SO_SNDLOWAT

socket option SO_SNDLOWAT — This option exists only for compatibility with UNIX 4.3BSD and has no effect on VSI TCP/IP sockets.

Format

socket option SO_SNDLOWAT

socket option SO_SNDTIMEO

socket option SO_SNDTIMEO — This option exists only for compatibility with UNIX 4.3BSD and has no effect on VSI TCP/IP sockets.

Format

socket option SO_SNDTIMEO

socket option SO_TYPE

socket option SO_TYPE — Retrieves the socket type (such as SOCK_DGRAM or SOCK_STREAM). This function is only valid with the **getsockopt()** function.

Format

Status = getsockopt(VMS_Channel, SOL_SOCKET, SO_TYPE, sizeof(*Value));

unsigned int *Value;

Arguments

Value	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by reference**

A pointer to an integer buffer that receives the socket type.

socket option TCP_KEEPALIVE

socket option TCP_KEEPALIVE — Lets you specify how long an idle socket remains open if the SO_KEEPALIVE option is enabled.

Format

Status = setsockopt(VMS_Channel, IPPROTO_TCP, TCP_KEEPALIVE, keepalive), sizeof(struct tcp_keepalive));

struct tcp_keepalive *keepalive

Description

If **SO_KEEPALIVE** is enabled, TCP_KEEPALIVE lets you specify:

Idle time	The amount of time a TCP socket should remain idle before sending the first keepalive packet.
Probe interval	The amount of time between keepalive packets.
Probe count	The number of keepalive packets to be sent before the connection is closed.

This feature is available to both the INETDRIVER and the UCXDRIVER, although it is usually accessed through the UCXDRIVER.

Arguments

Keepalive	
VMS Usage:	**keepalive_structure**

type:	**struct tcp_keepalive**
access:	**read only**
mechanism:	**by reference**

A pointer to a structure specifying the keepalive parameter values **idle_time**, **probe_intvl**, and **probe_count**.

The structure TCP_KEEPALIVE definition can be found in the include file `TCP.H`, as follows:

```
struct tcp_keepalive {
   int idle_time;    /*Time before first probe */
   int probe_intvl;  /*Time between probes */
   int probe_count;  /*Number of probes before closing connection */
};
```

The **idle_time** and **probe_intvl** values are specified in seconds; probe_count is the number of probes to send before closing the connection.

The minimum value for **idle_time** is 75 seconds. If a value less than 75 is specified, 75 is used.

If a value of 0 (zero) is specified for any of the entries in the structure, the current value is retained.

Note

The system default values are an **idle_time** value of 120 minutes, a **probe_intvl** value of 75 seconds, and a **probe_count** value of 8.

socket option TCP_NODELAY

socket option TCP_NODELAY — Disables the Nagle algorithm (RFC 896) which causes TCP to have, at most, one outstanding unacknowledged small segment. By default, the Nagle algorithm is enabled, delaying small segments of output data up to 200 ms so that they can be packaged into larger segments. If you enable TCP_NODELAY, TCP sends small segments as soon as possible, without waiting for acknowledgments from the receiver or for the 200 ms TCP fast timer to expire.

Format

Status = setsockopt(VMS_Channel, IPPROTO_TCP, TCP_NODELAY, On, sizeof(*On));

unsigned int *On;

Arguments

On	
VMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

A pointer to an integer buffer that specifies whether the TCP_NODELAY option is enabled or disabled. A value of 0 disables TCP_NODELAY.

socket_perror()

socket_perror() — Formats and prints the error code that is placed in the global variables **socket_errno** and **vmserrno** when an error occurs in one of the other socket functions. The error message is printed on the OpenVMS equivalent to the UNIX "stdout" device (normally SYS $OUTPUT), and is prefixed by the specified string.

Description

A typical use of **socket_perror()** might be the following:

```
if (connect(s, &sin, sizeof(sin)) < 0) {
    socket_perror("connect failed");
    exit(1);
}
```

Format

(void) socket_perror(String);

char *String;

Arguments

String	
VMS Usage:	**arbitrary_string**
type:	**ASCIZ string**
access:	**read only**
mechanism:	**by reference**

A C-language string with information about the last call to fail. This is printed as a prefix to the error message.

socket_read()

socket_read() — Reads messages from a socket. See also **recv()/recv_44()** and **recvfrom()recvfrom_44()**. This function is equivalent to a **recv()** function called with **Flags** specified as zero. The length of the message received is returned as the status. If a message is too long to fit in the supplied buffer and the socket is type SOCK_DGRAM, excess bytes are discarded. If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking (see **socket_ioctl()**). In this case, a status of -1 is returned, and the global variable **socket_errno** is set to EWOULDBLOCK.

Format

int socket_read (short VMS_Channel, char *Buffer, int Size);

Arguments

VMS_Channel	

VMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Buffer	
VMS Usage:	**arbitrary**
type:	**byte buffer**
access:	**write only**
mechanism:	**by reference**

The address of a buffer into which to place the data read.

Size	
VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by value**

The length of the buffer specified by **Buffer**. The actual number of bytes read is returned in the **Status**.

Returns

If the **socket_read()** routine is successful, the count of the number of characters received is returned. A return value of 0 indicates an end-of-file condition; that is, the connection has been closed. If an error occurs, a value of -1 is returned, and a more specific message is returned in the global variables **socket_errno** and **vmserrno**.

socket_write()

socket_write() — Writes a message to another socket. This function is equivalent to a **send()** function called with **Flags** specified as zero. This function can be used only when a socket has been connected with **connect()**. If no message space is available at the socket to hold the message to be transmitted, **socket_write()** blocks unless the socket has been placed in non-blocking I/O mode via the socket ioctl FIONBIO. If the socket is type SOCK_DGRAM and the message is too long to pass through the underlying protocol in a single unit, the error EMSGSIZE is returned and the message is not transmitted.

Format

int socket_write (short VMS_Channel, char *Buffer, int Size);

Arguments

VMS_Channel	

VMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Buffer	
VMS Usage:	**arbitrary**
type:	**byte buffer**
access:	**read only**
mechanism:	**by reference**

The address of a buffer containing the data to send.

Size	
VMS Usage:	**longword_signed**
type:	**longword (signed)**
access:	**read only**
mechanism:	**by value**

The length of the buffer specified by **Buffer**.

Returns

If the **socket_write()** routine is successful, the count of the number of characters sent is returned. If an error occurs, a value of -1 is returned, and a more specific error message is returned in the global variables **socket_errno** and **vmserrno**.

vms_errno_string()

vms_errno_string() — Formats a string corresponding to the error code that is placed in **socket_errno** and **vmserrno** when an error occurs in one of the other socket functions.

Format

(char *) vms_errno_string();

Returns

The **vms_errno_string()** function returns a pointer to the string.

2.4. SCTP

Support for SCTP (Stream Control Transport Protocol) has been added to the VSI TCP/IP C socket library, with the shareable image `IP$:TCPIP$SCTP_SHR.EXE`. SCTP provides end-to-end guaranteed delivery without the potential of blocking that TCP can encounter. SCTP also allows for

multiple streams within a conventional pairing of sockets between two IP addresses. Messages on one stream can be sent and received independently of other streams on the connection. See RFC 4960 for more information about SCTP.

Definitions for routines and constants are in

- `IP$ROOT:[IP.INCLUDE.NETINET]SCTP.H`
- `IP$ROOT:[IP.INCLUDE.NETINET]SCTP_CONSTANTS.H`
- `IP$ROOT:[IP.INCLUDE.NETINET]SCTP_UIO.H`

To use SCTP create a socket with the following parameters:

- socket (AF_INET, SOCK_STREAM, IPPROTO_SCTP)

The following routines are supported:

- int sctp_opt_info(int sd, sctp_assoc_t id, int opt, void *arg, short *size)

Description

sctp_opt_info is a wrapper library function that can be used to get SCTP level options on a socket.

Parameter Usage

sd is the socket descriptor for which the option is requested. For one-to-many style sockets, id specifies the association to query. For one-to-one style sockets, id is ignored.

opt specifes the SCTP socket option to get.

arg is an option-specific structure buffer provided by the caller. size is a value-result parameter, initially containing the size of the buffer pointed to by arg and modified on return to indicate the actual size of the value returned.

Returns

On success, **sctp_opt_info** returns 0 and on failure -1 is returned with errno set to the appropriate error code.

Supported Options:

SCTP_RTOINFO

SCTP_ASSOCINFO

SCTP_INITMSG

SCTP_NODELAY

SCTP_AUTOCLOSE

SCTP_PRIMARY_ADDR

SCTP_DISABLE_FRAGMENTS

SCTP_PEER_ADDR_PARAMS

SCTP_EVENTS

SCTP_I_WANT_MAPPED_V4_ADDR

SCTP_MAXSEG

SCTP_STATUS

SCTP_GET_PEER_ADDR_INFO

int sctp_bindx(int sd, struct sockaddr *addrs, int addrcnt, int flags)

int sctp_bindx(int sd, struct sockaddr *addrs, int addrcnt, int flags) — sctp_bindx adds or removes a set of bind addresses passed in the array addrs to/from the socket sd. addrcnt is the number of addresses in the array and the flags parameter indicates if the addresses need to be added or removed.

Description

An application can use `SCTP_BINDX_ADD_ADDR` to associate additional addresses with an endpoint after calling bind(2). `SCTP_BINDX_REM_ADDR` directs SCTP to remove the given addresses from the association. A caller may not remove all addresses from an association. It will fail with EINVAL.

Parameter Usage

If sd is an IPv4 socket, the addresses passed must be IPv4 addresses. If sd is an IPv6 socket, the addresses passed can be either IPv4 or IPv6 addresses.

addrs is a pointer to an array of one or more socket addresses. Each address is contained in its appropriate structure (i.e. **struct sockaddr_in** or **struct sockaddr_in6**). The family of the address type must be used to distinguish the address length.

The caller specifies the number of addresses in the array with addrcnt.

The flags parameter can be either `SCTP_BINDX_ADD_ADDR` or `SCTP_BINDX_REM_ADDR`.

Return Value

On success, 0 is returned. On failure, -1 is returned, and errno is set appropriately.

Errors

- EBADF - sd is not a valid descriptor.
- ENOTSOCK - sd is a descriptor for a file, not a socket.
- EFAULT - Error while copying in or out from the user address space.
- EINVAL - Invalid port or address or trying to remove all addresses from an association.

- EACCES - The address is protected, and the user is not the super-user.

int sctp_getpaddrs(int sd, sctp_assoc_t id, struct sockaddr **addrs)

int sctp_getpaddrs(int sd, sctp_assoc_t id, struct sockaddr **addrs) — **sctp_getpaddrs** returns all peer addresses in an association.

Description

On return, addrs will point to a dynamically allocated packed array of sockaddr structures of the appropriate type for each address. The caller should use **sctp_freepaddrs** to free the memory. Note that the in/out parameter addrs must not be NULL.

Parameter Usage

If sd is an IPv4 socket, the addresses returned will be all IPv4 addresses. If sd is an IPv6 socket, the addresses returned can be a mix of IPv4 or IPv6 addresses.

For one-to-many style sockets, id specifies the association to query. For one-to-one style sockets, id is ignored.

sctp_freepaddrs frees all the resources allocated by **sctp_getpaddrs**.

Return Value

On success, **sctp_getpaddrs** returns the number of peer addresses in the association. If there is no association on this socket, 0 is returned and the value of *addrs is undefined. On error, **sctp_getpaddrs** returns -1 and the value of *addrs is undefined.

sctp_freepaddrs (struct sockaddr *addrs)

sctp_freepaddrs (struct sockaddr *addrs) — The **sctp_freepaddrs()** and **sctp_freeladdrs()** functions are used to release the memory allocated by previous calls to **sctp_getpaddrs()** or **sctp_getladdrs()** respectively.

Format

sctp_freepaddrs (struct sockaddr *addrs)

sctp_getladdrs (int sd, sctp_assoc_t id, struct sockaddr **addrs)

sctp_getladdrs(int sd, sctp_assoc_t id, struct sockaddr **addrs) — **sctp_getladdrs** returns all locally bound addresses on a socket.

Description

On return, addrs will point to a dynamically allocated packed array of sockaddr structures of the appropriate type for each local address. The caller should use **sctp_freeladdrs** to free the memory. Note that the in/out parameter addrs must not be NULL.

Parameter Usage

If sd is an IPv4 socket, the addresses returned will be all IPv4 addresses. If sd is an IPv6 socket, the addresses returned can be a mix of IPv4 or IPv6 addresses.

For one-to-many style sockets, id specifies the association to query. For one-to-one style sockets, id is ignored. If the id field is set to 0, then the locally bound addresses are returned without regard to any particular association.

sctp_freeladdrs frees all the resources allocated by **sctp_getladdrs**.

Return Value

On success, **sctp_getladdrs** returns the number of local addresses bound to the socket. If the socket is unbound, 0 is returned and the value of *addrs is undefined. On error, **sctp_getladdrs** returns -1 and the value of *addrs is undefined.

sctp_freeladdrs (struct sockaddr *addrs)

sctp_freeladdrs(struct sockaddr *addrs) — The **sctp_freepaddrs()** and **sctp_freeladdrs()** functions are used to release the memory allocated by previous calls to **sctp_getpaddrs()** or **sctp_getladdrs()** respectively.

Format

sctp_freeladdrs(struct sockaddr *addrs)

int sctp_connectx(int sd, struct sockaddr *addrs, int addrcnt)

int sctp_connectx(int sd, struct sockaddr *addrs, int addrcnt) — **sctp_connectx** initiates a connection to a set of addresses passed in the array addrs to/from the socket sd. addrcnt is the number of addresses in the array.

Paramter Usage

If sd is an IPv4 socket, the addresses passed must be IPv4 addresses. If sd is an IPv6 socket, the addresses passed can be either IPv4 or IPv6 addresses.

addrs is a pointer to an array of one or more socket addresses. Each address is contained in its appropriate structure (i.e. **struct sockaddr_in** or **struct sockaddr_in6**). The family of the address type must be used to distinguish the address length. The caller specifies the number of addresses in the array with addrcnt.

Return Value

On success, 0 is returned. On failure, -1 is returned, and errno is set appropriately.

Errors

EBADF - sd is not a valid descriptor.

ENOTSOCK - sd is a descriptor for a file, not a socket.

EFAULT - Error while copying in or out from the user address space.

EINVAL - Invalid port or address.

EACCES - The address is protected, and the user is not the super-user.

EISCONN - The socket is already connected.

ECONNREFUSED - No one listening on the remote address.

ETIMEDOUT - Timeout while attempting connection. The server may be too busy to accept new connections. Note that for IP sockets the timeout may be very long when syncookies are enabled on the server.

ENETUNREACH - Network is unreachable.

EADDRINUSE - Local address is already in use.

EINPROGRESS - The socket is non-blocking and the connection cannot be completed immediately. It is possible to select(2) or poll(2) for completion by selecting the socket for writing. After select indicates writability, use getsockopt(2) to read the SO_ERROR option at level SOL_SOCKET to determine whether connect completed successfully (SO_ERROR is zero) or unsuccessfully (SO_ERROR is one of the usual error codes listed here, explaining the reason for the failure).

EALREADY - The socket is non-blocking and a previous connection attempt has not yet been completed.

EAGAIN - No more free local ports or insufficient entries in the routing cache. For PF_INET see the net.ipv4.ip_local_port_range sysctl in ip(7) on how to increase the number of local ports.

EAFNOSUPPORT - The passed address did not have the correct address family in its **sa_family** field.

EACCES, EPERM - The user tried to connect to a broadcast address without having the socket broadcast flag enabled or the connection request failed because of a local firewall rule.

sctp_assoc_t sctp_getassocid(int sd, struct sockaddr *addr)

sctp_assoc_t sctp_getassocid(int sd, struct sockaddr *addr) — return an association id for a specified socket address. The **sctp_getassocid()** call attempts to look up the specified socket address addr and find the respective association identification.

Return Values

The call returns the association id upon success and 0 is returned upon failure.

Errors

The **sctp_getassocid()** function can return the following errors.

ENOENT- The address does not have an association setup to it.

EBADF - The argument s is not a valid descriptor.

ENOTSOCK - The argument s is not a socket.

int sctp_getaddrlen (int family)

int sctp_getaddrlen(int family) — return the address length of an address family.

Description

The **sctp_getaddrlen()** function returns the size of a specific address family. This function is provided for application binary compatibility since it provides the application with the size the operating system thinks the specific address family is. Note that the function will actually create an SCTP socket and then gather the information via a **getsockopt()** system calls. If for some reason a SCTP socket cannot be created or the **getsockopt()** call fails, an error will be returned with errno set as specified in the **socket()** or **getsockopt()** system call.

Return Values

The call returns the number of bytes that the operating system expects for the specific address family or SOCKET_ERROR (-1).

Errors

The **sctp_getaddrlen()** function can return the following errors:

EINVAL - The address family specified does NOT exist.

Chapter 3. Using the $QIO System Service

This chapter describes how to use the $QIO system service and its data structures with TCP/IP Services.

After you create a network pseudodevice (BG:) and assign a channel to it, use the $QIO system service for I/O operations.

3.1. $QIO System Service Variations

The two variations of the $QIO system service are:

- Queue I/O Request ($QIO) — Completes asynchronously. It returns to the caller immediately after queuing the I/O request, without waiting for the I/O operation to complete.
- Queue I/O Request and Wait ($QIOW) — Completes synchronously. It returns to the caller after the I/O operation completes. The only difference between the $QIO and $QIOW calling sequences is the service name. The system service arguments are the same.

3.2. $QIO Format

The $QIO calling sequence has the following format:

```
SYS$QIO [efn],chan,func,[iosb],[astadr],[astprm],[p1],[p2],[p3],[p4],[p5],
[p6]
```

The following table describes each argument.

Table 3.1. $QIO Arguments

Argument	Description
astadr	AST (asynchronous system trap) service routine
astprm	AST parameter to be passed
chan	I/O channel
efn	Event flag number
func	Network pseudodevice function code and/or function modifier
iosb	I/O status block
p1, p2, p3, p4, p5, p6	Function-specific I/O request parameters

3.2.1. Symbol Definition Files

The following table lists the symbol definition files for the $QIO arguments p1 through p6. Use the standard mechanism for the programming language you are using to include the appropriate symbol definition files in your program.

Table 3.2. Network Symbol Definition Files

File Name	File Name
TCPIP$INETDEF.H	C
TCPIP$INETDEF.FOR	VAX Fortran
TCPIP$INETDEF.PAS	VAX PASCAL
TCPIP$INETDEF.MAR	MACRO-32
TCPIP$INETDEF.PLI	VAX PL/1
TCPIP$INETDEF.R32	BLISS-32
TCPIP$INETDEF.ADA	VAX Ada
TCPIP$INETDEF.BAS	VAX BASIC

3.3. $QIO Functions

The following table lists the $QIO function codes commonly used in a network application.

Note

The IO$_SETMODE and IO$_SETCHAR function codes are identical. All references to the IO $_SETMODE function code, its arguments, options, function modifiers, and condition values returned also apply to the IO$_SETCHAR function code, which is not explicitly described in this manual.

The IO$_SENSEMODE and IO$_SENSECHAR function codes are identical. All references to the IO$_SENSEMODE function code, its arguments, options, function modifiers, and condition values returned also apply to the IO$_SENSECHAR function code, which is not explicitly described in this manual.

Table 3.3. $QIO Function Codes

$QIO Function Codes	Description
$QIO(IO$_SETMODE) $QIO(IO$_SETCHAR)	Creates the socket by setting the internet domain, protocol (socket) type, and protocol of the socket.
	Binds a name (local address and port) to the socket.
	Defines a network pseudodevice as a listener on a TCP/IP server.
	Specifies socket options.
$QIO(IO$_ACCESS)	Initiates a connection request from a client to a remote host using TCP.
	Specifies the peer where you can send datagrams.
	Accepts a connection request from a TCP/IP client when used with the IO$M_ACCEPT function modifier.
$QIO(IO$_WRITEVBLK)	Writes data (virtual block) from the local host to the remote host for stream sockets, datagrams, and raw IP.
$QIO(IO$_READVBLK)	Reads data (virtual block) from the remote host to the local host for stream sockets, datagrams, and raw IP.
$QIO(IO$_DEACCESS)	Disconnects the link established between two communication agents through an IO$_DEACCESS function.

$QIO Function Codes	Description
	Shuts down the communication link when used with the IO $M_SHUTDOWN function modifier. You can shut down the receive or transmit portion of the link, or both.
$QIO(IO$_SENSECHAR) $QIO(IO$_SENSEMODE)	Obtains socket information.

3.4. $QIO Arguments

You pass two types of arguments with the $QIO system service: functionindependent arguments and function-dependent arguments. The following sections provide information about $QIO system service arguments.

3.4.1. $QIO Function-Independent Arguments

The following table describes the $QIO function-independent arguments.

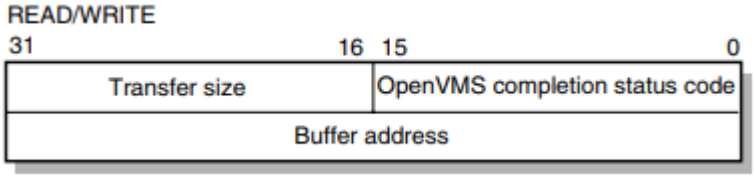
Table 3.4. $QIO Function-Independent Arguments

Argument	Description
astadr	Address of the asynchronous system trap (AST) routine to be executed when the I/O operation is completed.
astprm	A quadword (Alpha) or longword (VAX) containing the value to be passed to the AST routine.
chan	A longword value that contains the number of the I/O channel. The $QIO system service uses only the low-order word.
efn	A longword value of the event flag number that the $QIO system service sets when the I/O operation completes. The $QIO system service uses only the low-order byte.
func	A longword value that specifies the network pseudodevice function code and function modifiers that specify the operation to be performed.
	Function modifiers affect the operation of a specified function code. In MACRO-32, you use the exclamation point (!) to logically OR the function code and its modifier. In Compaq C, you use the vertical bar (|). This manual uses the vertical bar (|) in text.
iosb	The I/O status block that receives the final status message for the I/O operation. The iosb argument is the address of the quadword I/O status block. (For the format of the I/O status block, see nextsection

3.4.2. I/O Status Block

The system returns the status of a $QIO operation in the I/O status block (IOSB) supplied as an argument to the $QIO call. In the case of a successful IO$_READVBLK or IO$_WRITEVBLK operation, the second word of the I/O status block contains the number of bytes transferred during the operation (see Figure 5–1).

Figure 3.1. I/O Status Block for a Successful READ or WRITE Operation



With an unsuccessful IO$_READVBLK or IO$_WRITEVBLK operation, in most cases, the system returns a UNIX error code in the second word of the I/O status block.

For C programs, the OpenVMS completion codes are defined in the SSDEF.H header file. The UNIX error codes are defined in the ERRNO.H header file and in the TCPIP$INETDEF.H header file. For other language variants, see xxxTable 5–2.

3.4.3. $QIO Function-Dependent Arguments

Arguments **p1, p2, p3, p4, p5,** and **p6** to the $QIO system service are used to pass function-dependent arguments. Table 5–5 lists arguments p1 through p6 for the $QIO system service and indicates whether the parameter is passed by value, by reference, or by descriptor.

Table 3.5. $QIO Function-Dependent Arguments

$QIO	p1	p2	p3	p4	p5	p6
IO$_ACCESS	Not used	Not used	Remote socket name[a]	Not used	Not used	Not used
IO$_ACCESS | IO $M_ACCEPT	Not used	Not used	Remote socket name[b]	Channel number[c]	Not used	Not used
IO$_ACPCONTROL	Subfunction code[d]	Input parameter[d]	Buffer length[c]	Buffer[d]	Not used	Not used
IO$_DEACCESS	Not used	Not used	Not used	Not used	Not used	Not used
IO$_DEACCESS | IO $M_SHUTDOWN	Not used	Not used	Not used	Shutdown flags[e]	Not used	Not used
IO$_READVBLK	Buffer[c]	Buffer size[e]	Remote socket name[b]	Flags[e]	Not used	Output buffer list[d]
IO$_READVBLK \| IO$M_INTERRUPT	Buffer[c]	Buffer size[e]	Not used	Not used	Not used	Not used
IO$_WRITEVBLK	Buffer[c]	Buffer size[e]	Remote socket name[a]	Flags[e]	Input buffer list[d]	Not used
IO$_WRITEVBLK \| IO$M_INTERRUPT	Buffer[c]	Buffer size[e]	Not used	Not used	Not used	Not used
IO$_SETMODE	Socket char[c]	Not used	Local socket name	Backlog limit[e]	Input parameter list[a]	Not used
IO$_SETMODE | IO $_OUTBAND	AST procedure[c]	User argument[e]	Access mode[e]	Not used	Not used	Not used

$QIO	p1	p2	p3	p4	p5	p6
IO$_SETMODE | IO $_READATTN	AST procedure[c]	User argument[e]	Access mode[e]	Not used	Not used	Not used
IO$_SETMODE | IO $WRTATTN	AST procedure[c]	User argument[e]	Access mode[e]	Not used	Not used	Not used
IO$_SENSEMODE	Not used	Not used	Local socket name[b]	Remote socket name[b]	Not used	Output parameter list[a]

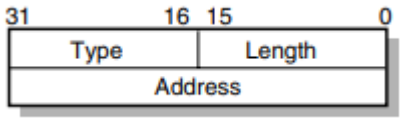
[a]By item_list_2 descriptor.
[b]By item_list_3 descriptor
[c]By reference.
[d]By descriptor
[e]By value

3.5. Passing Arguments by Descriptor

In addition to OpenVMS argument descriptors, I/O functions specific to TCP/IP Services also pass arguments by using item_list_2 and item_list_3 argument descriptors. The format of these argument descriptors is unique to TCP/IP Services, and they supplement argument descriptors defined in the OpenVMS Calling Standard.

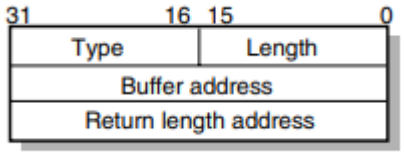
Use of an item_list_2 or item_list_3 argument descriptor is indicated when the argument's passing mechanism is specified as an item_list_2 descriptor or an item_list_3 descriptor.

The item_list_2 argument descriptors describe the size, data type, and starting address of a service parameter. An item_list_2 argument descriptor contains three fields, as depicted in the following diagram:



The first field is a word containing the length (in bytes) of the parameter being described. The second field is a word containing a symbolic code specifying the data type of the parameter. The third field is a longword containing the starting address of the parameter.

The item_list_3 argument descriptors describe the size, data type, and address of a buffer in which a service writes parameter information returned from a get operation. An item_list_3 argument descriptor contains four fields, as depicted in the following diagram:



The first field is a word containing the length (in bytes) of the buffer in which a service writes information. The length of the buffer needed depends on the data type specified in the type field. If the value of buffer length is too small, the service truncates the data. The second field is a word

containing a symbolic code specifying the type of information that a service is to return. The third field is a longword containing the address of the buffer in which a service writes the information. The fourth field is a longword containing the address of a longword in which a service writes the length (in bytes) of the information it actually returned.

Note

When a parameter specified as a descriptor is described as ‘‘read-only’’, the descriptor itself is only read, and TCP/IP Services does not modify the memory described. However, system service postprocessing requires that the described memory must be both readable and writable.

3.5.1. Specifying an Input Parameter List

Use the **p5** argument with the IO$_SETMODE function to specify input parameter lists. The **p5** argument specifies the address of a item_list_2 descriptor that points to and identifies the type of input parameter list.

To initialize an item_list_2 descriptor, you need to:

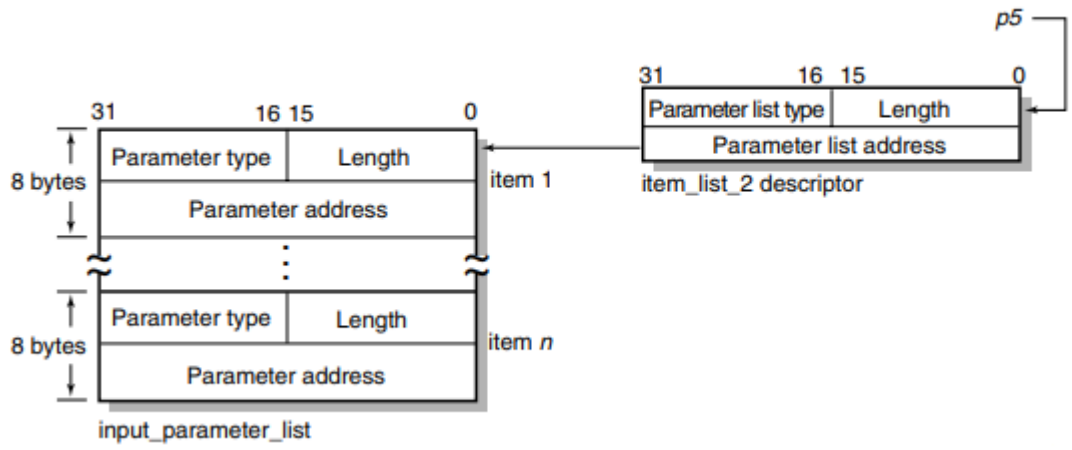
1. Set the descriptor’s type field to one of the following symbolic codes to specify the type of input parameter list:

Symbolic Name	Input Parameter List Type
TCPIP$C_SOCKOPT	Socket options
TCPIP$C_TCPOPT	TCP protocol options
TCPIP$C_IPOPT	IP protocol options
TCPIP$C_IOCTL	I/O control commands

2. Set the descriptor’s length field to specify the length of the input parameter list.
3. Set the descriptor’s address field to specify the starting address of the input parameter list.

The following figure illustrates how the **p5** argument specifies an input parameter list.

Figure 3.2. Specifying an Input Parameter List



As the name implies, input parameter lists consist of one or more contiguous item_list_2 or ioctl_comm structures. The length of a input parameter list is determined solely from the length field of its associated argument descriptor. Input parameter lists are never terminated by a longword containing a zero.

Each item_list_2 structure that appears in an input parameter list describes an individual parameter or item to set. Such items include socket or protocol options as identified by the item's type field. To initialize an item_list_2 descriptor, you need to:

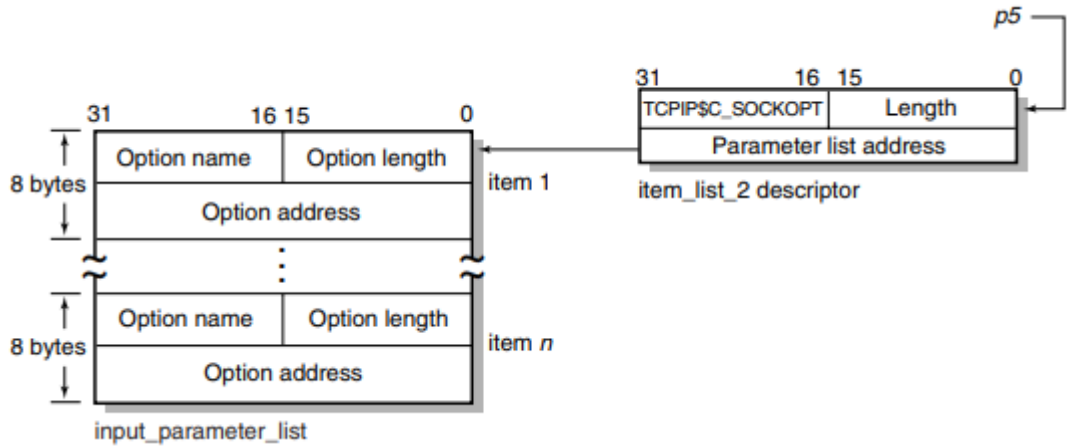
1. Set the item's type field to one of the symbolic codes in Appendix A.

2. Set the item's length field to specify the length of the item.

3. Set the item's address field to specify the starting address of its data.

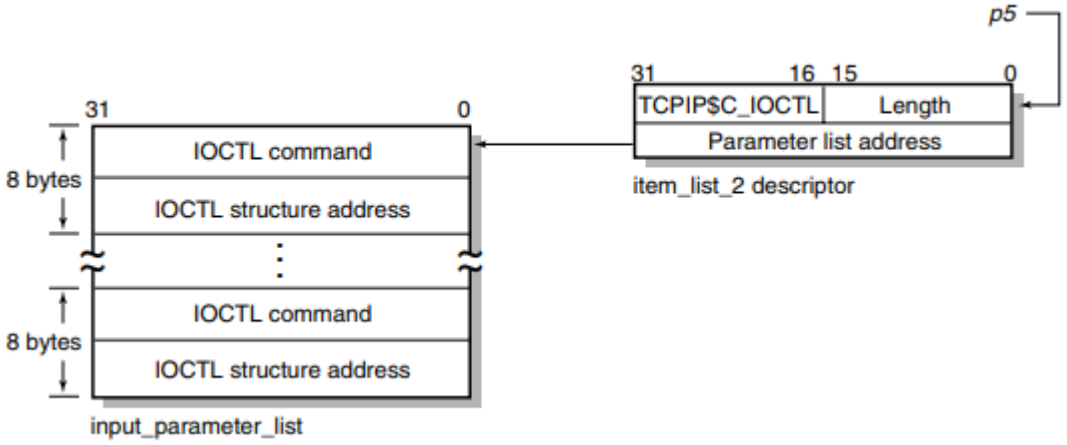
The following figure illustrates how to specify setting socket options.

Figure 3.3. Setting Socket Options



Each ioctl_comm structure appearing in an input parameter list contains an I/O control command---the IOCTL request code (as defined by $SIOCDEF) and its associated IOCTL structure address. The following figure illustrates how to specify (set) I/O control (IOCTL) commands.

Figure 3.4. Setting IOCTL Parameters



3.5.2. Specifying an Output Parameter List

Use the **p6** argument with the IO$_SENSEMODE function to specify output parameter lists. The **p6** argument specifies the address of an item_list_2 descriptor that points to and identifies the type of output parameter list.

To initialize an item_list_2 descriptor, you need to:

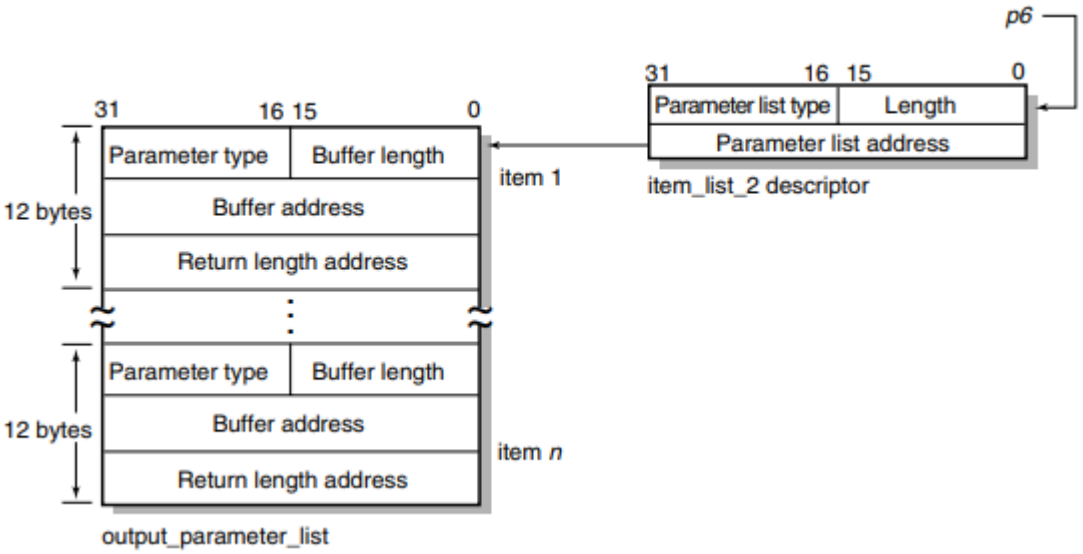
1. Set the descriptor's type field to one of the following symbolic codes to specify the type of output parameter list:

Symbolic Name	Output Parameter List Type
TCPIP$C_SOCKOPT	Socket options
TCPIP$C_TCPOPT	TCP protocol options
TCPIP$C_IPOPT	IP protocol options
TCPIP$C_IOCTL	I/O control commands

2. Set the descriptor's length field to specify the length of the output parameter list.
3. Set the descriptor's address field to specify the starting address of the output parameter list.

The following figure illustrates how the **p6** argument specifies an output parameter list:

Figure 3.5. Specifying an Output Parameter List



As the name implies, output parameter lists consist of one or more contiguous item_list_3 or ioctl_comm structures. The length of an output parameter list is determined solely from the length field of its associated argument descriptor. Output parameter lists are never terminated by a longword containing a zero.

Each item_list_3 structure that appears in an output parameter list describes an individual parameter or item to return. Such items include socket or protocol options as identified by the item's type field.

To initialize an item_list_3 structure, you need to:

1. Set the item's type field to one of symbolic codes found in Appendix A.

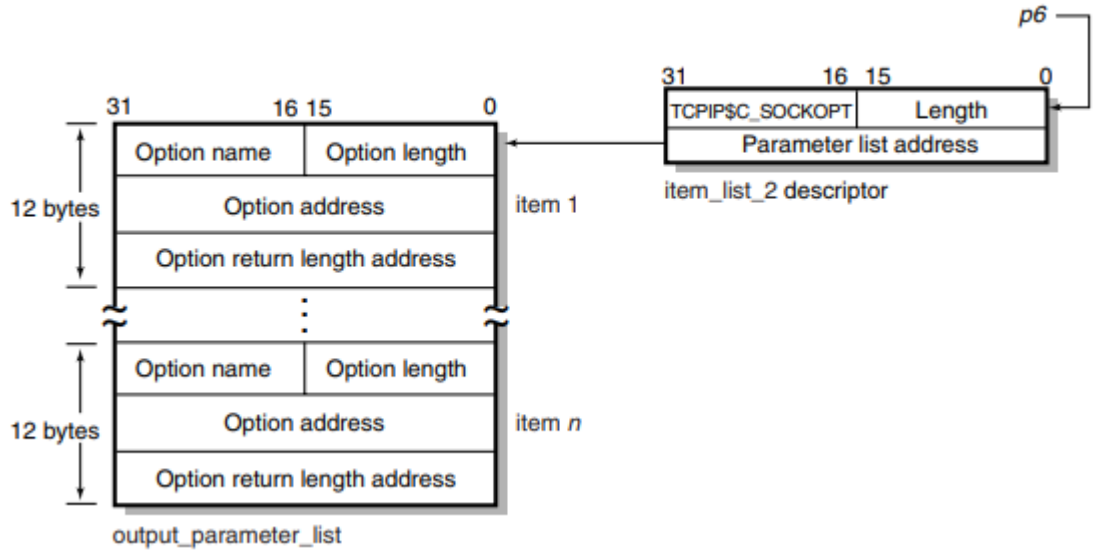
2. Set the item's buffer length field to specify the length of its buffer.

3. Set the item's buffer address field to specify the starting address of its buffer.

4. Set the item's returned length address field to specify the address of a longword to receive the length in bytes of the information actually returned for this item.

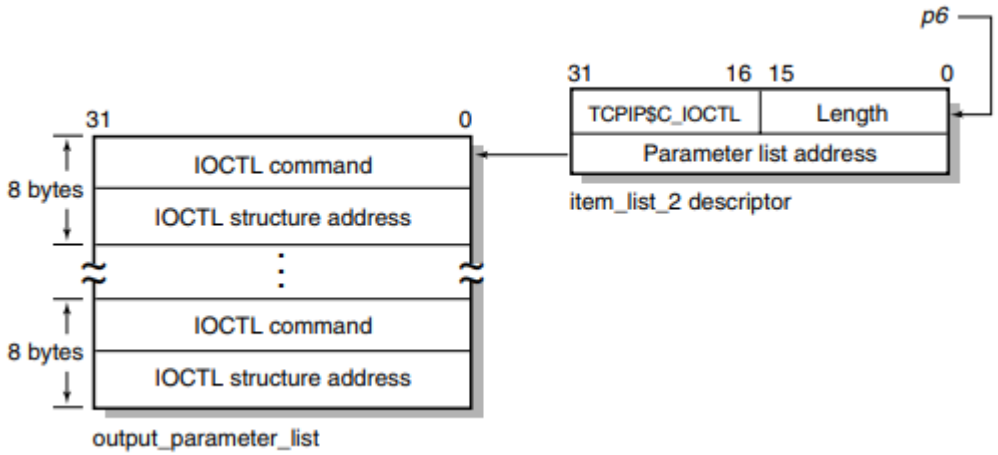
The following figure illustrates how to specify getting socket options.

Figure 3.6. Getting Socket Options



Each ioctl_comm structure appearing in a output parameter list contains an I/O control command---the IOCTL request code (as defined by $SIOCDEF) and its associated IOCTL structure address. The following figure illustrates how to specify (get) I/O control (IOCTL) commands.

Figure 3.7. Getting IOCTL Parameters



3.5.3. Specifying a Socket Name

Use the **p3** or **p4** argument with the IO$_ACCESS, IO$_READVBLK, IO$_SENSEMODE, IO$_SETMODE, and IO$_WRITEVBLK functions to specify a socket name.

The **p3** and **p4** arguments specify the address of an item_list_2 or item_list_3 descriptor that points to a socket name structure. The socket name structure contains address domain, port number, and host internet address.

Note

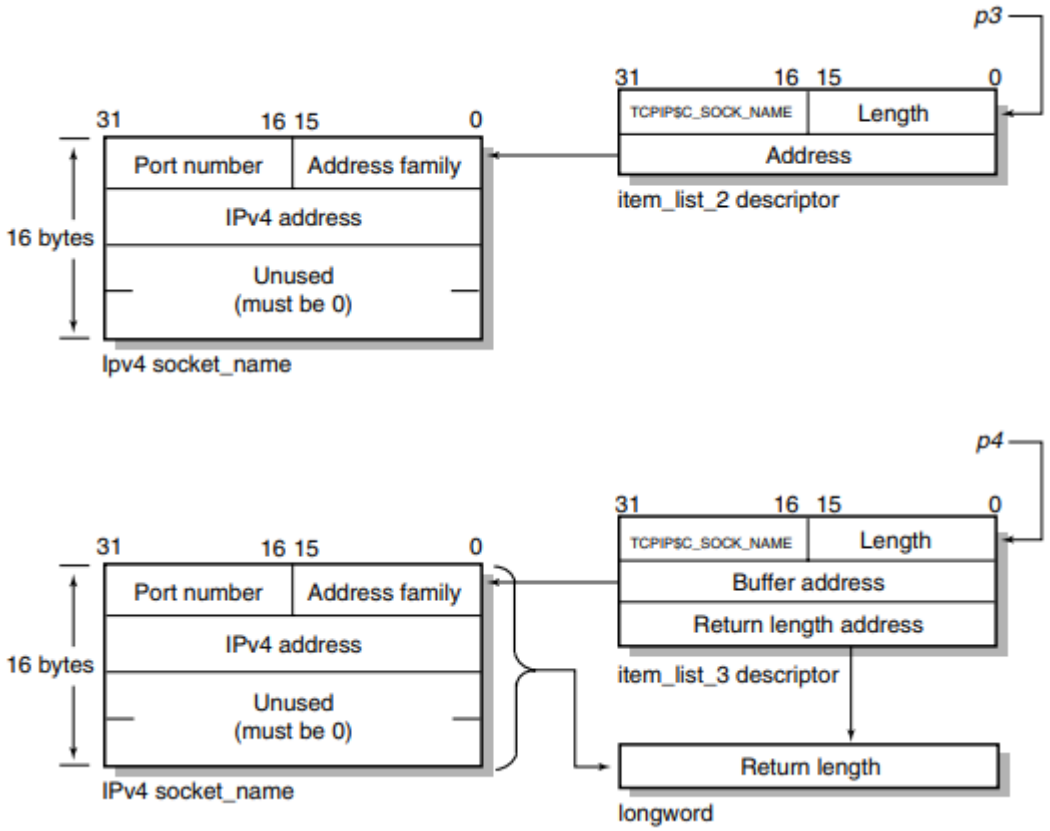
NotePort numbers 1 to 1023 require a system UIC or a UIC with SYSPRV and BYPASS privileges when assigned. If you specify zero when binding a socket name, the system assigns an available port.

Use an item_list_2 argument descriptor with the IO$_ACCESS, IO$_WRITEVBLK, and IO$_SETMODE functions to specify (set) a socket name. The descriptor's parameter type is TCPIP$C_SOCK_NAME.

Use an item_list_3 argument descriptor with the IO$_ACCESS|IO$M_ACCEPT, IO$_READVBLK, and IO$_SENSEMODE functions to specify (get) a socket name. The descriptor's parameter type is TCPIP$C_SOCK_NAME.

With BSD Version 4.3, specify socket names as illustrated in the following figure:

Figure 3.8. Specifying a Socket Name



3.5.4. Specifying a Buffer List

Use the **p5** argument with the IO$_WRITEVBLK function to specify input buffer lists. The **p5** argument specifies the address of a 32- or 64-bit fixed-length descriptor (on Alpha systems) or a 32-bit fixed-length descriptor (on VAX systems) pointing to an input buffer list.

Use the **p6** argument with the IO$_READVBLK function to specify output buffer lists. The **p6** argument specifies the address of a 32- or 64-bit fixed-length descriptor (on Alpha systems) or a 32-bit fixed-length descriptor (on VAX systems) pointing to an output buffer list.

To initialize the **p5** or **p6** arugment descriptor, you need to:

1. Set the descriptor's data-type code (the DTYPE field) to DSC$K_DTYPE_DSC to specify a buffer list containing one or more descriptors defining the length and starting address of user buffers.

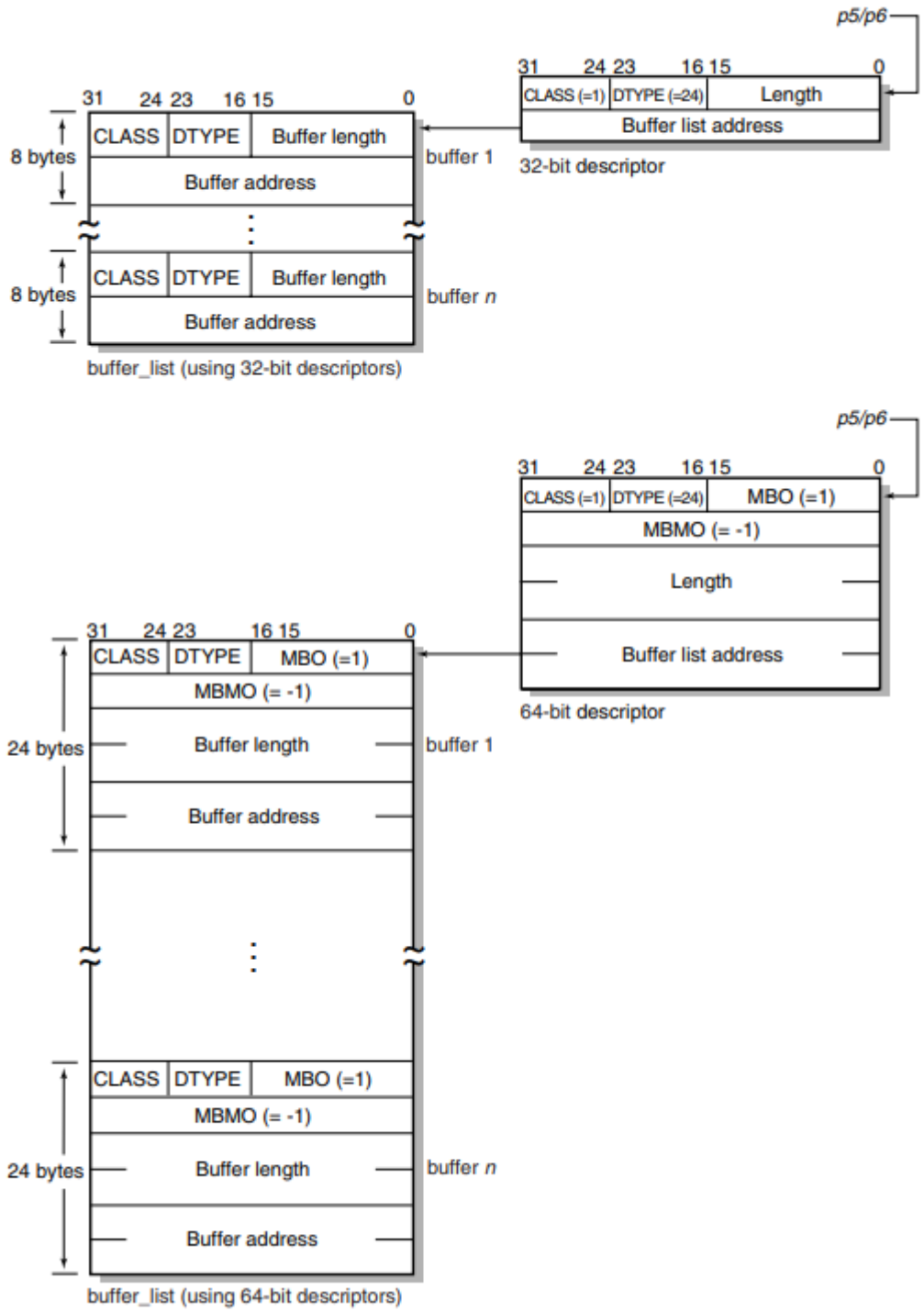
2. Set the descriptor's class code (the CLASS field) to DSC$K_CLASS_S.

3. Set the descriptor's length field to specify the length of the buffer list.

4. Set the descriptor's MBO field to 1 and the MBMO field to all 1s if this is a 64-bit argument descriptor.

The following figire illustrates how to specify a buffer list:

Figure 3.9. Specifying a Buffer List



Buffer lists, as the name implies, consist of one or more contiguous 32- or 64-bit fixed-length descriptors (on Alpha systems) or 32-bit fixed-length descriptors (on VAX systems).

Each 32- or 64-bit descriptor that appears in a buffer list describes one user buffer. Initialize each descriptor by setting its data type, class, length, and address fields as appropriate for 32- and 64-bit descriptors.

For more information about using 32-bit and 64-bit descriptors, refer to the *OpenVMS Calling Standard*.

Chapter 4. $QIO Interface

The $QIO interface allows programmers to use more sophisticated programming techniques than available with the socket library. Using the $QIO interface, you can perform fully asynchronous I/O to the network and receive Asynchronous System Traps (ASTs) when out-of-band data arrives (similar to the UNIX `SIGURG` signal). In general, there is a one-to-one mapping between the socket library functions and $QIO calls.

The $QIO interface returns an OpenVMS error code in the first word of the Input/Output Status Block (`IOSB`). If the low bit of the OpenVMS error code is clear, an error has been returned by the network. The OpenVMS error code is generated from the UNIX **errno** code by multiplying the UNIX code by 8 (eight) and logical ORing it with 0x8000.

You can mix and match the socket library function and the $QIO calls. For example, you can use **socket()** and **connect()** to establish a connection, then use IO$_SEND and IO$_RECEIVE to send and receive data on it.

Note

If more than one $QIO operation is pending on a socket at any one time, there is no guarantee that the $QIO calls will complete in the order they are queued. In particular, if more than one read or write operation is pending at any one time, the data may be interleaved. You do not need to use multiple read or write operations concurrently on the same socket to increase performance because of the network buffering.

The function codes for the VSI TCP/IP-specific $QIO functions are defined in the include file `IP_root:[IP.include.vms]inetiodef.h`.

If the compile time constant `USE_BSD44_ENTRIES` is defined, then the BSD 4.4 variant of the `IO$_ACCEPT`, `IO$_BIND`, `IO$_CONNECT`, `IO$_GETPEERNAME`, `IO$_GETSOCKNAME`, `IO$_RECEIVE`, `IO$_SEND` is selected.

The following are the interface functions:

IO$_ACCEPT	**IO$_SEND**
IO$_ACCEPT_WAIT	**IO$_SENSEMODE**
IO$_BIND	**IO$_SENSEMODE \| IO$M_CTRL**
IO$_CONNECT	**IO$_SETCHAR**
IO$_GETPEERNAME	**IO$_SETMODE\|IO$M_ATTNAST**
IO$_GETSOCKNAME	**IO$_SETSOCKOPT**
IO$_GETSOCKOPT	**IO$_SHUTDOWN**
IO$_IOCTL	**IO$_SOCKET**
IO$_LISTEN	**SYS$CANCEL**
IO$_RECEIVE (IO$_READVBLK)	**SYS$DASSGN**
IO$_SELECT	

IO$_ACCEPT

IO$_ACCEPT — Extracts the first connection from the queue of pending connections on a socket, creates a new socket with the same properties as the original socket, and associates an OpenVMS channel to the new socket. **IO$_ACCEPT** is equivalent to the **accept()** socket library function. Normally, instead of calling **IO$_ACCEPT** to wait for a connection to become available, **IO $_ACCEPT**_WAIT is used. This allows your process to wait for the connection without holding the extra network channel and tying up system resources. When the **IO$_ACCEPT**_WAIT completes, it indicates that a connection is available. **IO$_ACCEPT** is then called to accept it.

Format

Status = SYS$QIOW(Efn, New_VMS_Channel, **IO$_ACCEPT**, IOSB, AstAdr, AstPrm, Address, AddrLen, VMS_Channel, 0, 0, 0);

Arguments

New_VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

An OpenVMS channel to a newly-created INET device. Create this channel by using SYS$ASSIGN to assign a fresh channel to INET0: before issuing the **IO$_ACCEPT** call. The accepted connection is accessed using this channel.

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

The OpenVMS channel to the INET: device on which the **IO$_LISTEN** call was performed. After accepting the connection, this device remains available to accept new connections.

Address	
OpenVMS Usage:	**special_structure**
type:	**structure defined below**
access:	**write only**
mechanism:	**by reference**

An optional pointer to a structure that, following the completion of the **IO$_ACCEPT** call, contains the address of the socket that made the connection. This structure is defined as follows:

```
struct {
 unsigned long Length;
 struct sockaddr Address;
```

```
};
```

AddrLen	
OpenVMS Usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**read only**
mechanism:	**by value**

The length of the buffer pointed to by the **Address** argument, in bytes. It must be at least 20 bytes.

IO$_ACCEPT_WAIT

IO$_ACCEPT_WAIT — Used to wait for an incoming connection without accepting it. This allows your process to wait for the connection without holding the extra network channel and tying up system resources. When the **IO$_ACCEPT_WAIT** call completes, it indicates that a connection is available. **IO$_ACCEPT** is then called to accept it. The **IO$_ACCEPT_WAIT** call takes no function-specific parameters.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_ACCEPT_WAIT, IOSB, AstAdr, AstPrm, 0, 0, 0, 0, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

The OpenVMS channel to the INET: device on which the **IO$_LISTEN** call was performed.

IO$_BIND

IO$_BIND — Assigns an address to an unnamed socket. When a socket is created with **IO $_SOCKET**, it exists in a name space (address family) but has no assigned address. IO$_BIND requests that the address be assigned to the socket. **IO$_BIND** is equivalent to the bind() socket library function.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_BIND, IOSB, AstAdr, AstPrm, Name, NameLen, 0, 0, 0, 0);

Arguments

VMS_Channel	

OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Name	
OpenVMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**read only**
mechanism:	**by reference**

The address to which the socket should be bound. The exact format of the **Address** argument is determined by the domain in which the socket was created.

NameLen	
OpenVMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The length of the **Name** argument, in bytes.

IO$_CONNECT

IO$_CONNECT — When used on a **SOCK_STREAM** socket, this function attempts to make a connection to another socket. When used on a **SOCK_DGRAM** socket, this function permanently specifies the peer to which datagrams are sent to and received from. The peer socket is specified by name, which is an address in the communications domain of the socket. Each communications domain interprets the name parameter in its own way. **IO$_CONNECT** is equivalent to the **connect()** socket library function. If the address of the local socket has not yet been specified with **IO$_BIND**, the local address is also set to an unused port number when **IO$_CONNECT** is called.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_CONNECT, IOSB, AstAdr, AstPrm, Name, NameLen, 0, 0, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**

mechanism:	**by value**

A channel to the socket.

Name	
OpenVMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**read only**
mechanism:	**by reference**

The address of the peer to which the socket should be connected. The exact format of the **Address** argument is determined by the domain in which the socket was created.

NameLen	
OpenVMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The length of the Name argument, in bytes.

IO$_GETPEERNAME

IO$_GETPEERNAME — Returns the name of the peer connected to the specified socket. It is equivalent to the **getpeername()** socket library function.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_GETPEERNAME, IOSB, AstAdr, AstPrm, Address, AddrLen, 0, 0, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Address	
OpenVMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**write only**
mechanism:	**by reference**

A result parameter filled in with the address of the peer, as known to the communications layer. The exact format of the **Address** argument is determined by the domain in which the communication is occurring.

AddrLen	
OpenVMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

On entry, contains the length of the space pointed to by **Address**, in bytes. On return, it contains the actual length, in bytes, of the address returned.

IO$_GETSOCKNAME

IO$_GETSOCKNAME — Returns the current name of the specified socket. Equivalent to the **getsockname()** socket library function.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_GETSOCKNAME, IOSB, AstAdr, AstPrm, Address, AddrLen, 0, 0, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Address	
OpenVMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**write only**
mechanism:	**by reference**

A result parameter filled in with the address of the local socket, as known to the communications layer. The exact format of the **Address** argument is determined by the domain in which the communication is occurring.

AddrLen	
OpenVMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**modify**

mechanism:	**by reference**

On entry, contains the length of the space pointed to by **Address**, in bytes. On return, it contains the actual length, in bytes, of the address returned.

IO$_GETSOCKOPT

IO$_GETSOCKOPT — Retrieves options associated with a socket. It is equivalent to the **getsockopt()** library routine. Options can exist at multiple protocol levels; however, they are always present at the uppermost socket level. When manipulating socket options, you must specify the level at which the option resides and the name of the option. To manipulate options at the socket level, specify level as **SOL_SOCKET**. To manipulate options at any other level, specify the protocol number of the appropriate protocol controlling the option. For example, to indicate that an option is to be interpreted by the TCP protocol, set **Level** to the protocol number of TCP, as determined by calling **getprotobyname()**. **OptName** and any specified options are passed without modification to the appropriate protocol module for interpretation. The include file `IP_root:[IP.include.sys]socket.h` contains definitions for socket-level options. Options at other protocol levels vary in format and name. For more information on what socket options may be retrieved with **IO$_GETSOCKOPT**, see **setsockopt()**.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_GETSOCKOPT, IOSB, AstAdr, AstPrm, Level, OptName, OptVal, OptLen, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Level	
OpenVMS Usage:	**option_level**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The protocol level at which the option will be manipulated. Specify **Level** as **SOL_SOCKET** or a protocol number as returned by **getprotoent()**.

OptName	
OpenVMS Usage:	**option_name**
type:	**longword (unsigned)**
access:	**read only**

mechanism:	**by value**

The option that is to be manipulated.

OptVal	
OpenVMS Usage:	**dependent on OptName**
type:	**byte buffer**
access:	**write only**
mechanism:	**by reference**

A pointer to a buffer that is to receive the current value of the option. The format of this buffer is dependent on the option requested.

OptLen	
OpenVMS Usage:	**option_length**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

On entry, contains the length of the space pointed to by **OptVal**, in bytes. On return, it contains the actual length, in bytes, of the option returned.

IO$_IOCTL

IO$_IOCTL — Performs a variety of functions on the network; in particular, it manipulates socket characteristics, routing tables, ARP tables, and interface characteristics. The **IO$_IOCTL** call is equivalent to the **socket_ioctl()** library routine. A **IO$_IOCTL** request has encoded in it whether the argument is an input or output parameter, and the size of the argument, in bytes. Macro and define statements used in specifying an IO$_IOCTL request are located in the file `IP_root:[IP.include.sys]ioctl.h`.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_IOCTL, IOSB, AstAdr, AstPrm, Request, ArgP, 0, 0, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Request	

OpenVMS Usage:	**ioctl_request**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Which **IO$_IOCTL** function to perform. The available **IO$_IOCTL** functions are documented in the **socket ioctl** sections.

ArgP	
OpenVMS Usage:	**arbitrary**
type:	**byte buffer**
access:	**read, write, or modify depending on Request**
mechanism:	**by reference**

A pointer to a buffer whose format and function is dependent on the **Request** specified.

IO$_LISTEN

IO$_LISTEN — Specifies the number of incoming connections that may be queued while waiting to be accepted. This backlog must be specified before accepting a connection on a socket. The **IO $_LISTEN** function applies only to sockets of type **SOCK_STREAM**. The **IO$_LISTEN** call is equivalent to the **listen()** socket library function.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_LISTEN, IOSB, AstAdr, AstPrm, BackLog, 0, 0, 0, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Backlog	
OpenVMS Usage:	**connection_backlog**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Defines the maximum length of the queue of pending connections. If a connection request arrives when the queue is full, the request is ignored. The backlog queue length is limited to 5.

IO$_RECEIVE (IO$_READVBLK)

IO$_RECEIVE — Receives messages from a socket. This call is equivalent to the **recvfrom()**, **recv()**, and **socket_read()** socket library functions. The length of the message received is returned in the second and third word of the I/O Status Block (IOSB). A count of 0 indicates an end-of-file condition; that is, the connection has been closed. If a message is too long to fit in the supplied buffer and the socket is type **SOCK_DGRAM**, excess bytes are discarded. If no messages are available at the socket, the **IO$_RECEIVE** call waits for a message to arrive, unless the socket is nonblocking (see **socket_ioctl()**).

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_RECEIVE, IOSB, AstAdr, AstPrm, Buffer, Size, Flags, From, FromLen, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Buffer	
OpenVMS Usage:	**arbitrary**
type:	**byte buffer**
access:	**write only**
mechanism:	**by reference**

The address of a buffer in which to place the data read.

Size	
OpenVMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The length of the buffer specified by **Buffer**. The actual number of bytes read is returned in the **Status**.

Flags	
OpenVMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Control information that affects the **IO$_RECEIVE** call. The **Flags** argument is formed by ORing one or more of the following values:

```
#define   MSG_OOB    0x1   /* process out-of-band data */
#define   MSG_PEEK   0x2   /* peek at incoming message */
```

The MSG_OOB flag causes **IO$_RECEIVE** to read any out-of-band data that has arrived on the socket.

The MSG_PEEK flag causes **IO$_RECEIVE** to read the data present in the socket without removing the data. This allows the caller to view the data, but leaves it in the socket for future **IO$_RECEIVE** calls.

From	
OpenVMS Usage:	**special_structure**
type:	**structure defined below**
access:	**write only**
mechanism:	**by reference**

An optional pointer to a structure that, following the completion of the **IO$_RECEIVE**, contains the address of the socket that sent the packet. This structure is defined as follows:

```
struct {
 unsigned short Length;
 struct sockaddr Address;
};
```

FromLen	
OpenVMS Usage:	***word_unsigned***
type:	**word (unsigned)**
access:	**read only**
mechanism:	**by value**

The length of the buffer pointed to by the **From** argument, in bytes. It must be at least 18 bytes.

IO$_SELECT

IO$_SELECT — Examines the specified channel to see if it is ready for reading, ready for writing, or has an exception condition pending (the presence of out-of-band data is an exception condition). The UNIX **select()** system call can be emulated by posting multiple **IO$_SELECT** calls on different channels. **IO$_SELECT** is only useful for channels assigned to the INET: device. It cannot be used for any other VMS I/O device.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_SELECT, IOSB, AstAdr, AstPrm, Modes, 0, 0, 0, 0, 0);

Arguments

VMS_Channel	

OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Modes	
OpenVMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

On input, the **Modes** argument is a bit mask of one or more of the following values:

```
#define SELECT_DONTWAIT    (1<<0)
#define SELECT_READABLE    (1<<1)
#define SELECT_WRITEABLE   (1<<2)
#define SELECT_EXCEPTION   (1<<3)
```

If the SELECT_DONTWAIT bit is set, the **IO$_SELECT** call will complete immediately, whether or not the socket is ready for any I/O operations. If this bit is not set, the **IO$_SELECT** call will wait until the socket is ready to perform one of the requested operations.

If the SELECT_READABLE bit is set, the **IO$_SELECT** call will check if the socket is ready for reading or a connecting has been received and is ready to be accepted.

If the SELECT_WRITEABLE bit is set, the **IO$_SELECT** call will check if the socket is ready for writing or a connect request has been completed.

If the SELECT_EXCEPTION bit is set, the **IO$_SELECT** call will check if the socket has out-of-band data ready to read.

On output, the **Modes** argument is a bit mask that indicates which operations the socket is ready to perform. If the SELECT_DONTWAIT operation was specified, the Modes value may be zero; if SELECT_DONTWAIT is not specified, then one or more of the SELECT_READABLE, SELECT_WRITABLE, or SELECT_EXCEPTION bits will be set.

IO$_SEND

IO$_SEND — Transmits a message to another socket. It is equivalent to the **sendto()**, **send()**, and **socket_write()** socket library functions. If no message space is available at the socket to hold the message to be transmitted, **IO$_SEND** blocks unless the socket has been placed in non-blocking I/O mode via **IO$_IOCTL**. If the message is too long to pass through the underlying protocol in a single unit, the error EMSGSIZE is returned and the message is not transmitted.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_SEND, IOSB, AstAdr, AstPrm, Buffer, Size, Flags, To, ToLen, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Buffer	
OpenVMS Usage:	**arbitrary**
type:	**byte buffer**
access:	**read only**
mechanism:	**by reference**

The address of a buffer containing the data to send.

Size	
OpenVMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The length of the buffer specified by **Buffer**.

Flags	
OpenVMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Control information that affects the **IO$_SEND** call. The Flags argument can be zero or the following:

```
#define MSG_OOB 0x1 /* process out-of-band data */
```

The MSG_OOB flag causes IO$_SEND to send out-of-band data on sockets that support this operation (such as SOCK_STREAM).

To	
OpenVMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**read only**
mechanism:	**by reference**

An optional pointer to the address to which the packet should be transmitted. The exact format of the **Address** argument is determined by the domain in which the communication is occurring.

ToLen	
OpenVMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

An optional argument that contains the length of the address pointed to by the **To** argument.

IO$_SENSEMODE

IO$_SENSEMODE — Reads the active connections status and returns status information for all of the active and listening connections.

Format

Status = SYS$QIO(efn, chan, IO$_SENSEMODE, iosb, astadr, astprm, buffer, address, conn_type, 0, 0, 0)

Arguments

p1=buffer	
OpenVMS Usage:	**vector_byte_unsigned**
type:	**byte (unsigned)**
access:	**write only**
mechanism:	**by reference**

Optional address of the 8-byte device characteristics buffer. Data returned is: the device class (`DC$_SCOM`) in the first byte, the device type (0) in the second byte, and the default buffer size, which is the maximum datagram size, in the high-order word of the first longword. **IO$_SENSEMODE** returns the second longword as 0.

p2=address	
OpenVMS Usage:	**vector_word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by descriptor**

Address of the descriptor for the buffer to receive the status information on the active connections.

P3=value	
OpenVMS Usage:	**Longword_unsigned**
type:	**Longword (unsigned)**
access:	**Read only**

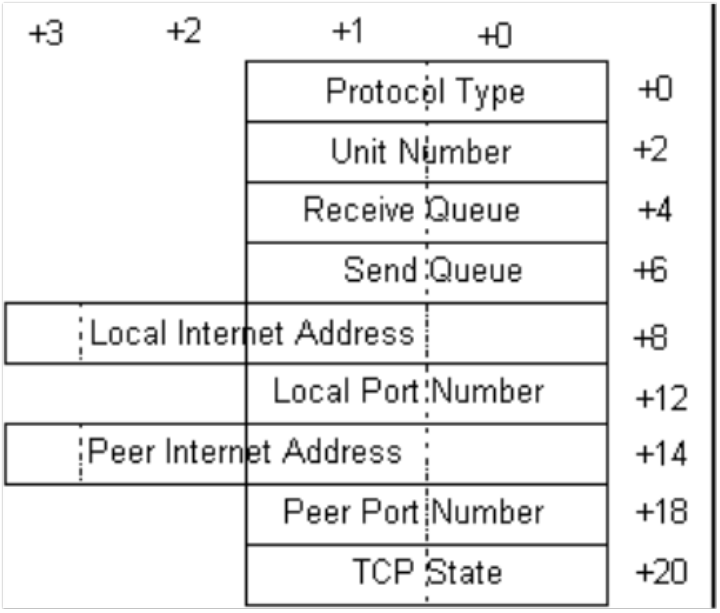
mechanism:	**by value**

0 to get information about TCP connections, non-zero to get information about UDP connections.

Figure 4.1, “Connection Status Information” shows the 22 bytes of information returned for each connection.

Protocol type	Word value is 4 for INETDRIVER stream sockets, and 5 for BGDRIVER stream sockets.
Unit number	Word value is the INETDRIVER, or BGDRIVER device unit number for the connection.
Receive queue	Word value is the number of bytes received from the peer waiting to be delivered to the user through the **IO$_READVBLK** function.
Send queue	Word value is the number of bytes waiting to be transmitted to or to be acknowledged by the peer.
Local internet address	Longword value is the local internet address (or 0 if the connection is not open and no local internet address was specified for the connection).
Local port number	Word value is the local port number.
Peer internet address	Longword value is the peer's internet address (or 0 if the connection is not open and no peer internet address was specified for the connection).
Peer port number	Word value is the peer's port number, or 0 if the connection is not open and you did not specify a peer port number for the connection.
TCP state	Word value is the Transmission Control Protocol connection state mask. See Table 4.1, “TCP State Mask Values” for the mask value definitions.

Figure 4.1. Connection Status Information



Status

SS$_BUFFEROVF	Buffer too small for all connections Truncated buffer returned
SS$_DEVINACT	Device not active Contact system manager why VSI TCP/IP (or INETDRIVER) not started
SS$_NORMAL	Success Status information returned

The byte count for the status information buffer is returned in the high-order word of the first longword of the I/O status block. This may be less than the bytes requested. See Figure 4.2, “I/O Status Block” for more information.

The size in bytes of each connection's record (22 bytes) is returned in the low order word of the second longword of the I/O status block.

The total number of active connections is returned in the high-order word of the second longword of the I/O status block. This can be greater than the number of reported connections if the buffer is full.

Figure 4.2. I/O Status Block

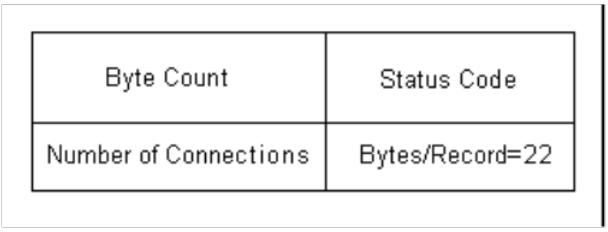


Table 4.1. TCP State Mask Values

Mask Value	State	Mask Value	State	Mask Value	State
1	LISTEN	16	FIN-WAIT-1	256	LAST-ACK
2	SYN-SENT	32	FIN-WAIT-2	512	TIME-WAIT
4	SYN-RECEIVED	64	CLOSE-WAIT	1024	CLOSED
8	ESTABLISHED	128	CLOSING		

IO$_SENSEMODE | IO$M_CTRL

IO$_SENSEMODE | IO$M_CTRL

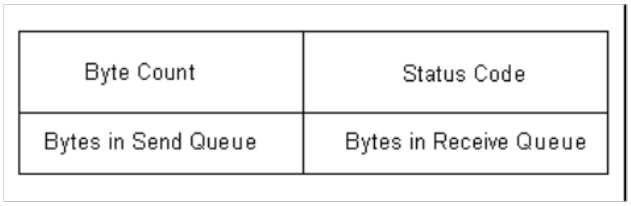
Description

SS$_BUFFEROVF	Buffer too small for all characteristics Truncated characteristics buffer is returned

SS$_DEVINACT	Device not active Contact system manager why VSI TCP/IP (or TCPDRIVER) not started
SS$_NORMAL	Success Characteristics returned

The byte count for the characteristics buffer is returned in the high-order word of the first longword of the I/O status block. This may be less than the bytes requested. The number of bytes in the receive queue is returned in the low order word of the second longword in the I/O status block. The number of bytes in the read queue is returned in the high-order word of the second longword in the I/O status block. Figure 4.3, “I/O Status Block” shows the I/O Status Block.

Figure 4.3. I/O Status Block



Note

You can use the `SYS$GETDVI` system service to obtain the local port number, peer port number, and peer internet address. The DEVDEPEND field stores the local port number (low order word) and peer port number (high-order word). The DEVDEPEND2 field stores the peer internet address.

Performs the following functions:

- Reads network device information
- Reads the routing table
- Reads the ARP information
- Reads the IP SNMP information
- Reads the ICMP SNMP information
- Reads the TCP SNMP information
- Reads the UDP SNMP information

Format

Status = SYS$QIO(efn, chan, IO$_SENSEMODE | IO$M_CTRL, iosb, astadr, astprm, buffer, address, function, line-id, 0, 0)

Arguments

p1=buffer	

OpenVMS Usage:	**vector_byte_unsigned**
type:	**byte (unsigned)**
access:	**write only**
mechanism:	**by reference**

Optional address of the 8-byte device characteristics buffer. The data returned is the device class (`DC$_SCOM`) in the first byte, the device type (0) in the second byte, and the default buffer size (0) in the high-order word of the first longword. The second longword is returned as 0.

p2=address	
OpenVMS Usage:	**vector_word_unsigned**
type:	**Word (unsigned)**
access:	**write only**
mechanism:	**by descriptor**

Address of the descriptor for the buffer to receive the information. The format of the buffer depends on the information requested. Each buffer format is described separately in the section that follows.

If bit 12 (mask 4096) is set in the parameter identifier (PID), the PID is followed by a counted string. If bit 12 is clear, the PID is followed by a longword value. While VSI TCP/IP currently never returns a counted string for a parameter, this may change in the future.

p3=function	
OpenVMS Usage:	**Longword-unsigned**
type:	**Longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Code that designates the function.

The function codes are shown in the Table 4.2, “P3 Function Codes”.

Table 4.2. P3 Function Codes

Code	Function
1	P1 of the QIO is not used
2	VMS descriptor of the space to put the return information
3	10
4	Not used
5	Not used
6	Not used
7	Read UDP SNMP counters
8	Read routing table
10	Read interface throughput information

p4=line-id	

OpenVMS Usage:	**Longword-unsigned**
type:	**Longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Specify this argument only if you are reading a network device's ARP table function.

Reading Network Device Information

Use **IO$_SENSEMODE | IO$M_CTRL** with p3=1 to read network device information. The information returned in the buffer (specified by p2=address) can consist of multiple records. Each record consists of nine longwords, and one record is returned for each device.

When you read network device information, the data in each record is returned in the order presented below. All are longword values.

1	Line id (see the description of the line-id argument)
2	Line's local internet address
3	Line's internet address network mask
4	Line's maximum transmission unit (MTU) in the low-order word, and the line flags in the high-order word
5	Number of packets transmitted (includes ARP packets for Ethernet lines)
6	Number of transmit errors
7	Number of packets received (includes ARP and trailer packets for Ethernet lines)
8	Number of receive errors
9	Number of received packets discarded due to insufficient buffer space

Reading the Routing Table

Use **IO$_SENSEMODE | IO$M_CTRL** with p3=8 to read the routing table. The information returned in the buffer (specified by p2=address) can consist of multiple records. Each record consists of five longwords, and one record is returned for each table entry.

The **p3=*8*** function returns full routing information and is a superset of **p3=*2,*** which was retained for backwards compatibility with existing programs. **p3=*2*** and **p3=*8*** return the same table of routing entries, in the following order, except that **p3=*2*** does not return items 7 and 8 (address mask and Path MTU):

1	Destination internet address.	Destination host or network to which the datagram is bound. Returned as a longword value.
2	Gateway internet address.	Internet address to which the datagram for this route is transmitted. Returned as a longword value.
3	Flags.	Routing table entry's flag bits. Returned as a word value: Mask 1, name GATEWAY, if set, the route is to a gateway (the datagram is sent to the gateway internet address). If clear, the route is a direct route.

		Mask 2, name HOST, if set, the route is for a host. If clear, the route is for a network. Mask 4, name DYNAMIC, if set, the route was created by a received ICMP redirect message. Mask 8, name AUTOMATIC, if set, this route was added by IP_RAPD process and will be modified or remoted by that process as appropriate. Mask 16, name LOCKED, if set, the route cannot be changed by an ICMP redirect message. Mask 32, name INTERFACE, if set, the route is for a network interface. Mask 64, name DELETED, if set, the route is marked for deletion (it is deleted when the reference count reaches 0). Mask 128, name POSSDOWN, if set, the route is marked as possibly down.
4	Reference count.	Number of connections currently using the route. Returned as a word value.
5	Use count.	Number of times the route has been used for outgoing traffic. Returned as a longword value.
6	Line ID.	Line identification for the network device used to transmit the datagram to the destination. See the description of the line-id argument later in this section for the line ID codes. Table 4.3, “Line ID Values” shows the line identification values.
7	Address mask.	Address mask for the destination address. Returned as a longword value.
8	Path MTU.	Path maximum transmission unit. Returned as a longword value.

Table 4.3. Line ID Values

Line ID	Line ID Value	Line ID	Line ID Value	Line ID	Line ID Value
LO-0	^X00000001	DN-*n*	^X00*nn0241*	PD-n	^X00*nn0042*
PSI-*n*	^X00*nn0006*	PPP-*n*	^X00*nn0341*		
SL-*n*	^X00*nn0141*	SE-*n*	^X00*nn0402*		

Note

The I/O status block (iosb) returns routing table entry size information for the p3=8 function to assist in diagnosing buffer overflow situations. See the Status section for details.

Reading Interface Throughput Information

Use **IO$_SENSEMODE | IO$M_CTRL** with p3=10 to read network device information. The information returned in the buffer (specified by p2=descriptor) can consist of multiple records. Each record consists of nine longwords, and one record is returned for each device.

When you read network device information, the data in each record is returned in the order presented below. All are longword values.

Table 4.4. QIO Parameters

Code	Function
1	P1 of the QIO is not used
2	is a VMS descriptor of the space to put the return information
3	10
4	Not used
5	Not used
6	Not used

The returned data is in the following format (all values are integers):

1	Line ID
2	Average Out Bytes (for the last 6 seconds)
3	Average In Bytes
4	Average Out Packets
5	Average In Packets

Reading the ARP Table Function

Use **IO$_SENSEMODE | IO$M_CTRL** with function=3 to read a network device's ARP table function. The information returned in the buffer (specified by p2=address) depends on the line id specified in line-id.

The line-id argument is the line id and is a longword value. The least significant byte of the line id is the major device type code. The next byte is the device type subcode. The next byte is the controller unit number. The most significant byte is ignored.

The information returned in the buffer can consist of multiple records. Each record consists of 12 bytes, and one record is returned for each ARP table entry.

When reading a table function, the data in each record is returned in the following order:

1. Internet address. Returned as a longword value.

2. Physical address. Returned as a 6 byte value.

3. Flags. Returned as a word value. The ARP table entry’s flag bits are shown in Table 4.5, “ARP Table Entry Flag Bits”.

Table 4.5. ARP Table Entry Flag Bits

Mask	Name	Description
1	PERMANENT	If set, the entry can only be removed by a NETCU REMOVE ARP command and if RARP is enabled, the local host responds if a RARP request is received for this address. If clear, the entry can be removed if not used within a short period.

Mask	Name	Description
2	PUBLISH	If set, the local host responds to ARP requests for the internet address (this bit is usually only set for the local hosts's entry). If clear, the local host does not respond to received ARP requests for this address.
4	LOCKED	If set, the physical address cannot be changed by received ARP requests/replies.
4096	LASTUSED	If set, last reference to entry was a use rather than an update.
8192	CONFNEED	If set, confirmation needed on next use.
16384	CONFPEND	If set, confirmation pending.
32768	RESOLVED	If set, the physical address is valid.

Status

SS$_BADPARAM	Code specified in *function argument invalid.*
SS$_BUFFEROVF	Buffer too small for all information Truncated buffer returned.
SS$_DEVINACT	Device not active Contact your system manager to determine why VSI TCP/IP was not started.
SS$_NORMAL	Success Requested information returned.
SS$_NOSUCHDEV	Line identification specified in *arp argument does not exist.*

The byte count for the information or counters buffer is returned in the high-order word of the first longword of the I/O status block. This can be less than the bytes requested.

- For the p3=2 routing table function, in the second longword of the I/O status block, bit 0 is always set, bit 1 is set if the forwarding capability is enabled, and bit 2 is set if ARP replies for non-local internet addresses are enabled.
- For the p3=8 routing table function, the IOSB contains the following:

Status Code	SS$_NORMAL or SS$_BUFFEROVF
Transfer Byte Count	Number of bytes of returned information
Entry Size	Number of bytes in each entry
Number of Entries	Number of entries in the routing table

If the status is SS$_BUFFEROVF, you can determine the number of routing entries actually returned by calculating (Transfer Byte Count) DIV (Entry Size) and comparing that with the Number of Entries value. Be sure to check the Entry Size in the IO status block.

Reading the IP SNMP Counters Function

Use **IO$_SENSEMODE | IO$M_CTRL** with function=4 to read the IP SNMP counters.

The data returned is an array of longwords in the following format:

- Indicates whether or not this entity is acting as an IP router.
- The default value inserted in the IP header's time-to-live field.
- The total number of input datagrams received.
- The number of input datagrams discarded due to errors in their IP headers.
- The number of input datagrams discarded because the IP address in their IP header's destination field was not a valid address to be received at this entity.
- The number of IP datagrams for which this entity was not their final destination, and for which forwarding to another entity was required.
- The number of datagrams received but discarded because of an unknown or unsupported protocol.
- The number of input datagrams received but discarded for reasons other than errors.
- The total number of input datagrams successfully delivered to IP user protocols, including ICMP.
- The total number of IP datagrams that local IP user protocols (including ICMP) supplied to IP in request for transmission.
- The number of output IP datagrams that were discarded for reasons other than errors.
- The number of IP datagrams discarded because no route could be found to transmit them to their destination.
- The maximum number of seconds that received fragments are held while they are awaiting reassembly at this entity.
- The number of IP fragments received that needed to be reassembled at this entity.
- The number of IP datagrams successfully reassembled.
- The number of failures detected by the IP reassembly algorithm.
- The number of IP datagrams that have been successfully fragmented at this entity.
- The number of IP datagrams that have been discarded at this entity because they could not be fragmented.
- The number of IP datagrams that have been created as a result of fragmentation at this entity.

Reading the ICMP SNMP Counters Function

Use **IO$_SENSEMODE | IO$M_CTRL** with function=5 to read the ICMP SNMP counters.

The data returned is an array of longwords in the following format:

- The total number of ICMP messages received.
- The number of ICMP messages received but determined as having ICMP-specific errors.
- The number of ICMP Destination Unreachable messages received.
- The number of ICMP Time Exceeded messages received.

- The number of ICMP Parameter Problem messages received.
- The number of ICMP Source Quench messages received.
- The number of ICMP Redirect messages received.
- The number of ICMP Echo (request) messages received.
- The number of ICMP Echo reply messages received.
- The number of ICMP Timestamp (request) messages received.
- The number of ICMP Timestamp Reply messages received.
- The number of ICMP Address Mask Request messages received.
- The number of ICMP Address Mask Reply messages received.
- The total number of ICMP messages that this entity attempted to send.
- The number of ICMP messages that this entity did not send because of ICMP-related problems.
- The number of ICMP Destination Unreachable messages sent.
- The number of ICMP Time Exceeded messages sent.
- The number of ICMP Parameter Problem messages sent.
- The number of ICMP Source Quench messages sent.
- The number of ICMP Redirect messages sent.
- The number of ICMP Echo (request) messages sent.
- The number of ICMP Echo reply messages sent.
- The number of ICMP Timestamp (request) messages sent.
- The number of ICMP Timestamp Reply messages sent.
- The number of ICMP Address Mask Request messages sent.
- The number of ICMP Address Mask Reply messages sent.

Reading the TCP SNMP Counters Function

Use **IO$_SENSEMODE | IO$M_CTRL** with function=6 to read TCP SNMP counters.

The data returned is an array of longwords in the following format:

- The algorithm used to determine the timeout value for retransmitting unacknowledged octets.
- The minimum value (measured in milliseconds) permitted by a TCP implementation for the retransmission timeout.
- The maximum value (measured in milliseconds) permitted by a TCP implementation for the retransmission timeout.

- The limit on the total number of TCP connections supported.
- The number of times TCP connections have made a transition to the SYN-SENT state from the CLOSED state.
- The number of times TCP connections have made a direct transition to the SYN-REVD state from the LISTEN state.
- The number of failed connection attempts.
- The number of resets that have occurred.

The number of TCP connections having a current state of either ESTABLISHED or CLOSE-WAIT.

- The total number of segments received.
- The total number of segments sent.
- The total number of segments retransmitted.

Reading the UDP SNMP Counters Function

Use **IO$_SENSEMODE | IO$M_CTRL** with function=7 to read the UDP SNMP counters.

The data returned is an array of longwords in the following format:

- The total number of IDP datagrams delivered to UDP users.
- The total number of received UDP datagrams for which there was not an application at the destination port.
- The number of received UDP datagrams that could not be delivered for reasons other than the lack of an application at the destination port.
- The total number of UDP datagrams sent from this entity.

IO$_SETCHAR

IO$_SETCHAR — Sets special characteristics that control the operation of the INET: device, rather than the socket attached to it. These operations are normally used by only the `IP_SERVER` process to hand off a connection to a process that it creates to handle the connection.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_SETCHAR, IOSB, AstAdr, AstPrm, Flags, 0, 0, 0, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**

access:	**read only**
mechanism:	**by value**

A channel to the socket.

Flags	
OpenVMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

A bit mask of one or more of the following values. If **IO$_SETCHAR** is not called, all options are set to **OFF**.

```
#define SETCHAR_PERMANENT   (1<<0)
#define SETCHAR_SHAREABLE   (1<<1)
#define SETCHAR_HANDOFF     (1<<2)
```

If the SETCHAR_PERMANENT bit is set when the last channel to the socket device is deassigned using the SYS$DASSGN system service, the socket is not closed and the socket device is not deleted. Normally, the last deassign closes the socket. If this bit has been set, it must be explicitly cleared before the socket can be deleted.

If the SETCHAR_SHAREABLE bit is set, the socket becomes a shareable device and any process can assign a channel to it.

If the SETCHAR_HANDOFF bit is set, the socket is not closed and the socket device is not deleted when the last channel to the socket device is deassigned. After this occurs, the socket reverts to a normal socket, and if a new channel is assigned and deassigned, the socket is closed. The SETCHAR_HANDOFF bit is a safer version of the SETCHAR_PERMANENT bit because it allows a single hand-off to another process without the risk of a socket getting permanently stuck on your system.

IO$_SETMODE|IO$M_ATTNAST

IO$_SETMODE|IO$M_ATTNAST — Enables an AST to be delivered to your process when out-of-band data arrives on a socket. This is similar to the UNIX 4.3BSD SIGURG signal being delivered. You cannot enable the delivery of the AST through the socket library functions. After the AST is delivered, you must explicitly reenable it using this call if you want the AST to be delivered when future out-of-band data arrives.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_SETMODE|IO$M_ATTNAST, IOSB, AstAdr, AstPrm, Routine, Parameter, 0, 0, 0, 0);

Arguments

Routine	
OpenVMS Usage:	**ast_procedure**

type:	**procedure entry mask**
access:	**call without stack unwinding**
mechanism:	**by reference**

The address of the AST routine to call when out-of-band data arrives on the socket. To disable AST delivery, set **Routine** to 0.

Parameter	
OpenVMS Usage:	**user_arg**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The argument with which to call the AST routine.

IO$_SETSOCKOPT

IO$_SETSOCKOPT — Manipulates options associated with a socket. It is equivalent to the **setsockopt()** socket library function. Options may exist at multiple protocol levels; however, they are always present at the uppermost socket level. When manipulating socket options, you must specify the level at which the option resides and the name of the option. To manipulate options at the socket level, specify **Level** as **SOL_SOCKET**. To manipulate options at any other level, specify the protocol number of the appropriate protocol controlling the option. For example, to indicate that an option is to be interpreted by the TCP protocol, set **Level** to the protocol number of TCP; see **getprotobyname()**. **OptName** and any specified options are passed without modification to the appropriate protocol module for interpretation. The include file `IP_root:[IP.include.sys]socket.h` contains definitions for socket-level options. Options at other protocol levels vary in format and name.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_SETSOCKOPT, IOSB, AstAdr, AstPrm, Level, OptName, OptVal, OptLen, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Level	
OpenVMS Usage:	**option_level**
type:	**longword (unsigned)**

access:	**read only**
mechanism:	**by value**

The protocol level at which the option will be manipulated. Specify **Level** as **SOL_SOCKET**, or a protocol number as returned by **getprotobyname()**.

OptName	
OpenVMS Usage:	**option_name**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The option that is to be manipulated. For a description of each of the valid options for **IO $_SETSOCKOPT**, see the **socket option** sections.

OptVal	
OpenVMS Usage:	**dependent on OptName**
type:	**byte buffer**
access:	**read only**
mechanism:	**by reference**

A pointer to a buffer that contains the new value of the option. The format of this buffer depends on the option requested.

OptLen	
OpenVMS Usage:	**option_length**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The length of the buffer pointed to by **OptVal**.

IO$_SHUTDOWN

IO$_SHUTDOWN — Shuts down all or part of a full-duplex connection on the socket associated with VMS_Channel. This function is usually used to signal an end-of-file to the peer without closing the socket itself, which would prevent further data from being received. It is equivalent to the **shutdown()** socket library function.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_SHUTDOWN, IOSB, AstAdr, AstPrm, How, 0, 0, 0, 0, 0);

Arguments

VMS_Channel	

OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

How	
OpenVMS Usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Controls which part of the full-duplex connection to shut down, as follows: if **How** is 0, further receive operations are disallowed; if **How** is 1, further send operations are disallowed; if **How** is 2, further send and receive operations are disallowed.

IO$_SOCKET

IO$_SOCKET — Creates an end point for communication and returns an OpenVMS channel that describes the end point. It is equivalent to the **socket()** socket library function. Before issuing the **IO $_SOCKET** call, an OpenVMS channel must first be assigned to the INET0: device to get a new channel to the network.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_SOCKET, IOSB, AstAdr, AstPrm, Address_Family, Type, Protocol, 0, 0, 0);

Arguments

Address_Family	
OpenVMS Usage:	**address_family**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

An address family with which addresses specified in later operations using the socket will be interpreted. The following formats are currently supported; they are defined in the include file `IP_root:[IP.include.sys]socket.h`:

AF_INET	Internet (TCP/IP) addresses
AF_PUP	Xerox PUP addresses
AF_CHAOS	CHAOSnet addresses

Type	

OpenVMS Usage:	**socket_type**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The semantics of communication using the created socket. The following types are currently defined:

SOCK_STREAM	SOCK_DGRAM	SOCK_RAW

A **SOCK_STREAM** socket provides a sequenced, reliable, two-way connection-oriented byte stream with an out-of-band data transmission mechanism. A **SOCK_DGRAM** socket supports communication by connectionless, unreliable messages of a fixed (typically small) maximum length. **SOCK_RAW** sockets provide access to internal network interfaces. The type **SOCK_RAW** is available only to users with **SYSPRV** privilege.

The **Type** argument, together with the **Address_Family** argument, specifies the protocol to be used. For example, a socket created with **AF_INET** and **SOCK_STREAM** is a TCP socket, and a socket created with **AF_INET** and **SOCK_DGRAM** is a UDP socket.

Protocol	
OpenVMS Usage:	**protocol_number**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

A protocol to be used with the socket. Normally, only a single protocol exists to support a particular socket type using a given address format. However, many protocols may exist, in which case a particular protocol must be specified by **Protocol**. The protocol number to use depends on the communication domain in which communication will take place.

For TCP and UDP sockets, the protocol number MUST be specified as 0. For **SOCK_RAW** sockets, the protocol number should be the value returned by **getprotobyname()**.

SYS$CANCEL

SYS$CANCEL — Cancels any I/O IOSB status of **SS$_CANCEL**. Outstanding I/O operations are automatically cancelled at image exit. For more information on **SYS$CANCEL**, see the *OpenVMS System Services Reference Manual*.

Format

Status = SYS$CANCEL(VMS_Channel);

SYS$DASSGN

SYS$DASSGN — Equivalent to the **socket_close()** function. When you deassign a channel, any outstanding I/O is completed with an **IOSB** status of **SS$_CANCEL**. Deassigning a channel closes the network connection. I/O channels are automatically deassigned at image exit. For more information on **SYS$DASSGN**, see the *OpenVMS System Services Reference Manual*.

Format

Status = SYS$DASSGN(VMS_Channel);

Chapter 5. SNMP Extensible Agent API Routines

This chapter is for application programmers. It describes the Application Programming Interface (API) routines required for an application program to export private Management Information Bases (MIBs) using the VSI TCP/IP SNMP agent.

To be able to use your private Management Information Base (MIB) with VSI TCP/IP's SNMP agent, develop a shareable image that exports the following application programming interface routines, in addition to routines you may need to access the MIB variables:

SnmpExtensionInit	Called by the SNMPD agent after startup to initialize the MIB subagent
SnmpExtensionInitEx	Registers multiple subtrees with the subagent (called by the SNMPD agent at startup only implemented)
SnmpExtensionQuery	Completes the MIB subagent query (called by the SNMPD agent to handle a get, getnext, or set request)
SnmpExtensionTrap	Sends an enterprise-specific trap (called by the SNMPD agent when the subagent alerts the agent that a trap needs to be set)

Note

The routine names used in this API are taken from the Microsoft SNMP Extension Agent for Windows NT.

The SNMP shareable images need to be configured for the SNMP agent to interact with them.

SNMP subagent developers should use the include file `SNMP_COMMON.H.` found in the `IP $COMMON_ROOT:[IP.INCLUDE]` directory. This file defines the data structures the API uses.

For details on VSI TCP/IP's SNMP agent, see the *VSI TCP/IP Administrator's Guide: Volume II*.

5.1. Requirements

You require the following before using the SNMP extensible agent API routines:

- Working knowledge of SNMP; specifically the following RFCs:
  - RFC 1155, *Structure and Identification of Management Information for TCP/IP-based Internets*
  - RFC 1157, *A Simple Network Management Protocol (SNMP)*
  - RFC 1213, *Management Information Base for Network Management of TCP/IP-based internets: MIB-II*
- Working knowledge of OpenVMS shareable images

5.2. Linking the Extension Agent Image

To link the Extension Agent Image you need to create an option file. The example below is for Alpha Systems.

Alpha

```
!Note: Exclude SnmpExtensionInitEx if it is not needed.
!See the definition of this routine.
!
SYMBOL_VECTOR=( SnmpExtensionInit=PROCEDURE, -
SnmpExtensionQuery=PROCEDURE, -
SnmpExtensionTrap=PROCEDURE, -
SnmpExtensionInitEx=PROCEDURE)
!
!List your object/library files here
```

Your link statement should then look like this:

```
$ LINK /SHARE=image-nameoption-file /OPT
```

image-name is the name of the shareable image you want to build, and *option-file* is the option file mentioned above.

5.3. Installing the Extension Agent Image

You should copy the shareable image you build for your SNMP subagent to the SYS$SHARE.

Warning

Since the shareable image is loaded into the same process address space as the SNMPD server, an access violation by the subagent shareable image can crash the server application. Ensure the integrity of your shareable image by testing it thoroughly. Shareable image errors can also corrupt the server's memory space or may result in memory or resource leaks.

5.4. Debugging Code

SNMP subagent developers can use a debug logical, IP$SNMP_DEBUG, to set certain debug masks. Define the logical as follows and use the *mask* values in Table 5.1, “Debugging Mask Values”:

```
$ DEFINE IP$SNMP_DEBUG mask
```

Table 5.1. Debugging Mask Values

Mask Value	z
0010	Raw SNMP input
0020	Raw SNMP output
0040	ASN.1 encoded message input
0080	ASN.1 encoded message output
1000	SNMP Subagent Developer debug mask (prints events and statuses)

5.5. Subroutine Reference

The following pages include the subroutine descriptions.

SnmpExtensionInit

SnmpExtensionInit — Initializes the SNMP subagent and registers the subagent in the SNMPD agent. The subagent calls this routine at startup.

Format

status = SnmpExtensionInit (*trap-alert-routine*, *time-zero-reference*, *trap-event*, *supported-view*)

Return Values

TRUE	Subagent initialized successfully
FALSE	Subagent initialization failed

Arguments

trap-alert-routine	
OpenVMS usage:	address
type:	integer
access:	read only
mechanism:	by value

Address of the routine the subagent should call when it is ready to send a trap.

trap-event	
OpenVMS usage:	unsigned long
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Currently unused.

time-zero-reference	
OpenVMS usage:	unsigned long
type:	longword (unsigned)
access:	read only
mechanism:	by value

Time reference the SNMP agent provides, in hundredths of a second. Use C routines `time()` and `difftime()` to calculate MIB uptime (in hundredths of a second).

supported-view	
OpenVMS usage:	object identifier
type:	AsnOBJID (see the SNMP_COMMON.H file)
access:	write only

mechanism:	by reference

Prefix of the MIB tree the subagent supports.

SnmpExtensionInitEx

SnmpExtensionInitEx — Registers multiple MIB subtrees with agent. This routine is called multiple times, once for each MIB subtree that needs to be registered. If the routine passes back the first or next MIB subtree, return with TRUE. If all the MIB subtrees were passed back, return with FALSE. Only implement this routine if you have multiple MIB subtrees in your extendible agent. The VSI TCP/IP SNMP agent executes this routine if it exists and overwrites MIB information set by **SnmpExtensionInit**.

Format

status = SnmpExtentionInitEx (*supported-view*)

Return Values

TRUE	Returning first or next MIB subtree
FALSE	All MIB subtrees were passed back

Arguments

supported-view	
OpenVMS usage:	object identifier
type:	AsnOBJID (see the `SNMP_COMMON.H` file)
access:	write only
mechanism:	by reference

Prefix of the MIB tree the subagent supports.

Example

```
int SnmpExtensionInitEx (AsnOBJID *supportedView)
 {
  int view1[] = {1, 3, 6, 1, 4, 1, 12, 2, 1 };
  int view2[] = {1, 3, 6, 1, 4, 1, 12, 2, 2 };
  int view3[] = {1, 3, 6, 1, 4, 1, 12, 2, 5 };
  static int whichView = 0;
  switch ( whichView++) {
  case 0:
  supportedView->idLength = 9;
  memcpy (supportedView->ids, view1, 9* sizeof (int));
  break;
 case 1:
  supportedView->idLength = 9;
  memcpy (supportedView->ids, view2, 9* sizeof (int));
  break;
 case 2:
  supportedView->idLength = 9;
```

```
 memcpy (supportedView->ids, view3, 9* sizeof (int));
 break;
default:
 return (0);
}
return (1);
}
```

SnmpExtensionQuery

SnmpExtensionQuery — Queries the SNMP subagent to get or set a variable in the MIB tree served by the subagent. This routine is called by the SNMPD agent to handle a get, getnext, or set request.

Format

status = SnmpExtensionQuery (*request-type*, *var-bind-list*, *error-status*, *error-index*)

Return Values

TRUE	Operation successfully completed
FALSE	Operation could not be carried out by the subagent; use *error-status* and *error-index* to provide more information

Arguments

request-type	
OpenVMS usage:	byte
type:	unsigned char
access:	read only
mechanism:	by value

Identifies the type of request **GET**, **SET**, or **GET NEXT**.

var-bind-list	
OpenVMS usage:	user defined
type:	RFC1157VarBindList (see the SNMP_COMMON.H file)
access:	read-write
mechanism:	by value

The list of name-value pairs used in the request. For a **GET** request the value is filled by the subagent and for a **SET** request, the value is be used to change the current variable value in the subagent.

error-status	
OpenVMS usage:	integer
type:	integer
access:	write only
mechanism:	by reference

Status of a failed operation.

error-index	
OpenVMS usage:	integer
type:	integer
access:	write only
mechanism:	by reference

The index of the variable in the variable binding list for which the operation failed.

SnmpExtensionTrap

SnmpExtensionTrap — Sends a trap from the subagent. If the subagent wants to send a trap, it must first call the trap-alert-routine (see **SnmpExtensionInit** routine). The trap-alert-routine should be called with two parameters (objids, idlength). For example:

Description

If the VSI's DNS process wants to send trap information to all the communities that are interested then the DNS server must be running and the objectids passed are 1, 3, 6, 1, 4, 1, 105, 1, 2, 1, 1, 1, 3, 1, and the length of 14.

- 1,3,6,1,4,1 is the default prefix
- 105 is the enterprise id for VSI
- 1,2,1,1,1 are the Mib object ids for the DNS process
- 3,1 are the objectids for DNSUpTrap

The SNMP agent trap-alert-routine creates a table of all received trap mibs. For each of these entries, the agent then calls the subagent's **SnmpExtensionTrap** routine when it is ready to send the trap.

Note

The SNMP agent calls the subagent from inside the trap-alert-routine.

Format

status = SnmpExtensionTrap (*`enterprise`*, *`generic-trap`*, *`specific-trap`*, *`time-stamp`*, *`var-bind-list`*)

Return Values

TRUE	More traps to be generated
FALSE	No more traps to be generated

Arguments

enterprise	

OpenVMS usage:	array of object identifiers
type:	AsnOBJID (see the SNMP_COMMON.H file)
access:	write only
mechanism:	by reference

The prefix of the MIB for the enterprise sending the trap.

generic-trap	
OpenVMS usage:	integer
type:	integer
access:	write only
mechanism:	by reference

The generic enterprise trap id(6).

specific-trap	
OpenVMS usage:	integer
type:	integer
access:	write only
mechanism:	by reference

The enterprise-specific trap number.

Note

Since an enterprise can have many traps, the combination of enterprise id, generic trap, and specific trap should give a unique identification for a trap.

time-stamp	
OpenVMS usage:	integer
type:	integer (timeticks)
access:	write only
mechanism:	by reference

The time at which the trap was generated.

var-bind-list	
OpenVMS usage:	user defined
type:	RFC1157VarBindList (see the SNMP_COMMON.H file)
access:	read-write
mechanism:	by value

The list of name-value pairs. This list contains name and value of the MIB variable for which the trap is generated.

Chapter 6. RPC Fundamentals

6.1. Introduction

VSI TCP/IP RPC Services must be used with the VSI TCP/IP C Socket Library.

This chapter is for RPC programmers. It provides basic information you need to know before using RPC Services to write distributed applications, including:

- What RPC Services are
- What components are in RPC Services
- How RPC clients and servers communicate
- Important RPC concepts and terms

6.2. What Are RPC Services?

RPC Services are a set of software development tools that allow you to build distributed applications on OpenVMS systems.

6.2.1. VSI TCP/IP Implementation

RPC Services are based on the Open Network Computing Remote Procedure Call (RPC) protocols developed by Sun Microsystems, Inc. These protocols are defined in the following Requests for Comments (RFCs):

- *RPC: Remote Procedure Call Protocol Specification, Version 2* (RFC 1057)
- *XDR: External Data Representation Standard* (RFC 1014)

6.2.2. Distributed Applications

A distributed application executes different parts of its programs on different hosts in a network. Computers on the network share the processing workload, with each computer performing the tasks for which it is best equipped.

For example, a distributed database application might consist of a central database running on a system server and numerous client workstations. The workstations send requests to the server. The server carries out the requests and sends the results back to the workstations. The workstations use the results in other modules of the application.

RPCs allow programs to invoke procedures on remote hosts as if the procedures were local. RPC Services hides the networking details from the application.

RPC Services facilitates distributed processing because it relieves the application programmer of performing low-level network tasks such as establishing connections, addressing sockets, and converting data from one machine's format to another.

6.3. Components of RPC Services

RPC Services comprises the following components: Run-Time Libraries (RTLs), RPCGEN Compiler, Port Mapper, RPC Information.

6.3.1. Run-Time Libraries (RTLs)

RPC Services provides a single shareable RTL. The library contains:

- RPC client and server routines
- XDR routines

The Chapter 9, *RPC RTL Management Routines*, and the chapters that follow it describe the RTLs in detail.

6.3.2. RPCGEN Compiler

RPCGEN is a compiler that creates the network interface portion of a distributed application. It effectively hides from the programmer the details of writing and debugging low-level network interface code. Chapter 8, ***RPCGEN** Compiler* describes how to use RPCGEN.

6.3.3. Port Mapper

The Port Mapper helps RPC client programs connect to ports that are being used by RPC servers. A Port Mapper runs on each host that implements RPC Services. These steps summarize how the Port Mapper works:

1. RPC servers register with the Port Mapper by telling it which ports they are using.
2. When an RPC client needs to reach a particular server, it supplies the Port Mapper with the numbers of the remote program and program version it wants to reach. The client also specifies a transport protocol (UDP or TCP). (provides details on these numbers.)
3. The Port Mapper provides the correct port number for the requested service. This process is called binding.

Once binding has taken place, the client does not have to call the Port Mapper for subsequent calls to the same server. A service can register for different ports on different hosts. For example, a server can register for port 800 on Host A and port 1000 on Host B. The Port Mapper is itself an RPC server and uses the RPC RTL. The Port Mapper plays an important role in disseminating messages for broadcast RPC. The Port Mapper is part of the Master Server Process.

6.3.4. RPC Information

Use the RPC information command to:

- Request a listing of all programs that are registered with the Port Mapper

You enter this command at the DCL prompt. (See Chapter 7, *Building Distributed Applications with RPC* for more information.)

6.4. Client-Server Relationship

In RPC, the terms client and server do not describe particular hosts or software entities. Rather, they describe the roles of particular programs in a given transaction. Every RPC transaction has a client and a server. The client is the program that calls a remote procedure; the server is the program that executes the procedure on behalf of the caller.

A program can be a client or a server at different times. The program's role merely depends on whether it is making the call or servicing the call.

6.5. External Data Representation (XDR)

External Data Representation (XDR) is a standard that solves the problem of converting data from one machine's format to another.

RPC Services uses the XDR data description language to describe and encode data. Although similar to C language, XDR is not a programming language. It merely describes the format of data, using implicit typing. *XDR: External Data Representation Standard* (RFC 1014) defines the XDR language.

6.6. RPC Processing Flow

Remote and local procedure calls share some similarities. In both cases, a calling process makes arguments available to a procedure. The procedure uses the arguments to compute a result, then returns the result to the caller. The caller uses the results of the procedure and resumes execution.

Figure 6.1, “RPC Processing Flow” shows the underlying processing that makes a remote procedure call different from a local call.

The following steps describe the processing flow during a remote procedure call:

1. The client program passes arguments to the client stub procedure. (See Chapter 8, ***RPCGEN** Compiler* for details on how to create stubs.)

2. The client stub marshals the data by:

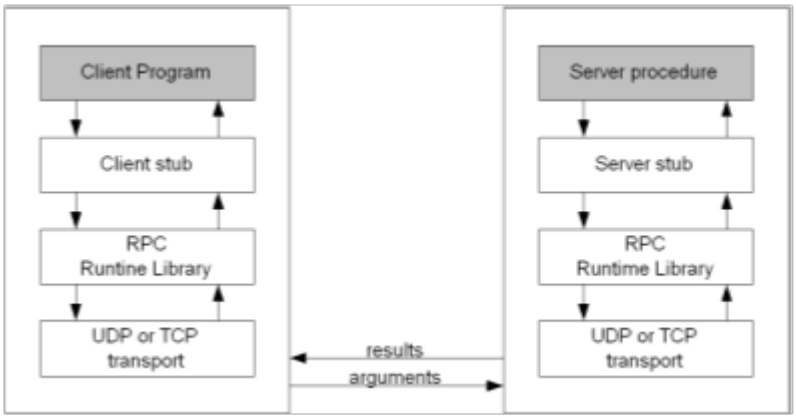
   - Calling the XDR routines to convert the arguments from the local representation to XDR
   - Placing the results in a packet

   Using RPC RTL calls, the client stub sends the packet to the UDP or TCP layer for transmission to the server.

3. The packet travels on the network to the server, up through the layers to the server stub.

4. The server stub un-marshals the packet by converting the arguments from XDR to the local representation. Then it passes the arguments to the server procedure.

Figure 6.1. RPC Processing Flow



6.7. Local Calls versus Remote Calls

This section describes some of the ways in which local and remote procedure calls handle system crashes, errors, and call semantics.

6.7.1. Handling System Crashes

Local procedure calls involve programs that reside on the same host. Therefore, the called procedure cannot crash independently of the calling program.

Remote procedure calls involve programs that reside on different hosts. Therefore, the client program does not necessarily know when the remote host has crashed.

6.7.2. Handling Errors

If a local procedure call encounters a condition that prevents the call from executing, the local operating system usually tells the calling procedure what happened.

If a remote procedure call cannot be executed for some reason (e.g., errors occur on the network or remote host), the client might not be informed of what happened. You may want to build a signaling or condition-handling mechanism into the application to inform the client of such errors.

RPC returns certain types of errors to the client, such as those that occur when it cannot decode arguments. The RPC server must be able to return processing-related errors, such as those that occur when arguments are invalid, to the client. However, the RPC server may not return errors during batch processing or broadcast RPC.

6.7.3. Call Semantics

Call semantics determine how many times a procedure executes.

Local procedures are guaranteed to execute once and only once.

Remote procedures have different guarantees, depending on which transport protocol is used.

The TCP transport guarantees execution once and only once as long as the server does not crash. The UDP transport guarantees execution at least once. It relies on the XID cache to prevent a remote procedure from executing multiple times.

See Section 6.10, “XID Cache” for details on the XID cache.

6.8. Programming Interface

The RPC RTL is the programming interface to RPC. You may think of this interface as containing multiple levels.

The RPC RTL reference chapters describe each routine.

6.8.1. High-Level Routines

The higher-level RPC routines provide the simplest RPC programming interface. These routines call lower-level RPC routines using default arguments, effectively hiding the networking details from the application programmer.

When you use high-level routines, you sacrifice control over such tasks as client authentication, port registration, and socket manipulation, but you gain the benefits of using a simpler programming interface. Programmers using high-level routines can usually develop applications faster than they can using low-level RPC routines.

You can use the RPCGEN compiler only when you use the highest-level RPC programming interface.

6.8.2. Mid-Level Routines

The mid-level routines provide the most commonly used RPC interface. They give the programmer some control over networking tasks, but not as much control as the low-level routines permit.

For example, you can control memory allocation, authentication, ports, and sockets using mid-level routines.

The mid-level routines require you to know procedure, program, and version numbers, as well as input and output types. Output data is available for future use. You can use the registerrpc and callrpc routines.

6.8.3. Low-Level Routines

The low-level routines provide the most complicated RPC interface, but they also give you the most control over networking tasks such as client authentication, port registration, and socket manipulation. These routines are used for the most sophisticated distributed applications.

6.9. Transport Protocols

RPC Services uses the transport protocols listed in the following table. The RPC client and server must use the same transport protocol for a given transaction.

Table 6.1. RPC Transport Protocols

Protocols	Characteristics
UDP	Unreliable datagram service
	Connectionless
	Used for broadcast RPC
	Maximum broadcast message size in either direction on an Ethernet line: 1500
	Execution is guaranteed at least once (see Section 6.10, “XID Cache”)
	Calls cannot be processed in batch
TCP	Reliable
	Connection-oriented
	Can send an unlimited number of bytes per RPC call
	Execution is guaranteed once and only once
	Calls can be processed in batch
	No broadcasting

You must use the VSI TCP/IP C Socket Library with RPC Services.

6.10. XID Cache

The XID cache stores responses the server has sent. When the XID cache is enabled, the server does not have to recreate every response to every request. Instead, the server can use the responses in the cache. Thus, the XID cache saves computing resources and improves the performance of the server.

Only the UDP transports can use the XID cache. The reliability of the TCP transport generally makes the XID cache unnecessary. UDP is inherently unreliable.

Table 6.2, “XID Cache Differences” shows how the XID caches differ for the UDP and UDPA/TCPA transports.

Table 6.2. XID Cache Differences

UDP Transport	UDPA/TCPA Transports
Places every response in the XID cache	Allows the server to specify which responses are to be cached, using the svcudp_enablecache and svctcpa_enablecache routines
XID cache cannot be disabled	Requires you to disable the XID cache after use

6.10.1. Cache Entries

Each entry in the XID cache contains:

- The encoded response that was sent over the network
- The internet address of the client that sent the request
- The transaction ID that the client assigned to the request

6.10.2. Cache Size

You determine the size of the XID cache. Consider these factors:

- How many clients are using the server.
- Approximately how long the cache should save the responses.
- How much memory you can allocate. Each entry requires about 8Kbytes.

The more active the server is, the less time the responses remain in the cache.

6.10.3. Execution Guarantees

As explained earlier in Local Calls versus Remote Callssection remote procedures have different execution guarantees, depending on which transport protocol is used. The XID cache affects the execution guarantee.

The TCP transport guarantees execution once and only once as long as the server does not crash. The UDP transport guarantees execution at least once. If the XID cache is enabled, a UDP procedure is unlikely to execute more than once.

6.10.4. Enabling XID Cache

Use the svcudp_enablecache routine to enable the XID cache. This routine is described in the RPC RTL reference chapters.

Not enabling the XID cache saves memory.

6.11. Broadcast RPC

Broadcast RPC allows the client to send a broadcast call to all Port Mappers on the network and wait for multiple replies from RPC servers.

For example, a host might use a broadcast RPC message to inform all hosts on a network of a system shutdown.

Table 6.3, “Normal RPC vs Broadcast RPC” shows the differences between normal RPC and broadcast RPC.

Table 6.3. Normal RPC vs Broadcast RPC

Normal RPC	Broadcast RPC
Client expects one answer	Client expects many answers
Can use TCP or UDP	Requires UDP
Server always responds to errors	Server does not respond to errors; Client does not know when errors occur
Port Mapper is desirable, but not required if you use fixed port numbers	Requires Port Mapper services

Broadcast RPC sends messages to only one port — the Port Mapper port — on every host in the network. On each host, the Port Mappers pass the messages to the target RPC server. The servers compute the results and send them back to the client.

6.12. Identifying Remote Programs and Procedures

The RPC client must uniquely identify the remote procedure it wants to reach. Therefore, all remote procedure calls must contain these three fields:

- A remote program number
- The version number of the remote program
- A remote procedure number

6.12.1. Remote Program Numbers

A remote program is a program that implements at least one remote procedure. Remote programs are identified by numbers that you assign during application development. Use Table 6.4, “Remote Program Numbers” to determine which program numbers are available. The numbers are in groups of hexadecimal 20000000.

Table 6.4. Remote Program Numbers

Range	Purpose
0 to 1FFFFFFF	Defined and administered by Sun Microsystems. Should be identical for all sites. Use only for applications of general interest to the Internet community.
20000000 to 3FFFFFFF	Defined by the client application program. Site-specific. Use primarily for new programs.
40000000 to 5FFFFFFF	Use for applications that generate program numbers dynamically.
60000000 to FFFFFFFF	Reserved for the future. Do not use.

6.12.2. Remote Version Numbers

Multiple versions of the same program may exist on a host or network. Version numbers distinguish one version of a program from another. Each time you alter a program, remember to increment its version number.

6.12.3. Remote Procedure Numbers

A remote program may contain many remote procedures. Remote procedures are identified by numbers that you assign during application development. Follow these guidelines when assigning procedure numbers:

- Use 1 for the first procedure in a program. (Procedure 0 should do nothing and require no authentication to the server.)
- For each additional procedure in a program, increment the procedure number by one.

6.13. Additional Terms

Before writing RPC applications, you should be familiar with the terms in the Table 6.5, “Additional Terms”.

Table 6.5. Additional Terms

Term	Definition
Channel	An OpenVMS term referring to a logical path that connects a process to a physical device, allowing the process to communicate with that device. A process requests OpenVMS to assign a channel to a device. Refer to Hewlett-Packard’s documentation for more information on channels.
Client handle	Information that uniquely identifies the server to which the client is sending the request. Consists of the server's host name, program number, program version number, and transport protocol. See the following routines in the Chapter 10, *RPC RTL Client Routines*:

authnone_create	clnt_create	clnt_perror / clnt_sperror

<table>
<tr><th>Term</th><th>Definition</th></tr>
<tr><td></td><td>
<table>
<tr><td>authunix_create</td><td>clnttcp_create</td><td></td></tr>
<tr><td>authunix_create_defa</td><td>clntudp_create / clntudp_bufcreate</td><td></td></tr>
</table>
</td></tr>
<tr><td>Port</td><td>An abstract point through which a datagram passes from the host layer to the application layer protocols.</td></tr>
<tr><td>Server handle</td><td>Information that uniquely identifies the server. Content varies according to the transport being used. See the following routines in Chapter 12, RPC RTL Server Routines:
<table>
<tr><td>svcudp_create /</td><td>svctcp_create</td><td>svc_destroy</td></tr>
<tr><td>svc_freeargs</td><td>svc_getargs</td><td></td></tr>
<tr><td>svc_register</td><td>svc_sendreply</td><td></td></tr>
</table>
</td></tr>
<tr><td>Socket</td><td>An abstract point through which a process gains access to the Internet. A process must open a socket and bind it to a specific destination.
Note
The VSI TCP/IP C Socket Library must be used with RPC Services.</td></tr>
</table>

Chapter 7. Building Distributed Applications with RPC

7.1. Introduction

This chapter is for RPC programmers. It explains:

- What components a distributed application contains
- How to use RPC to develop a distributed application, step by step
- How to get RPC information

7.2. Distributed Application Components

Table 7.1, “Application Components” lists the components of a distributed application.

Table 7.1. Application Components

Component	Description
Main program (client)	An ordinary main program that calls a remote procedure as if local
Network interface	Client and server stubs, header files, XDR routines for input arguments and results
Server procedure	Carries out the client's request (at least one is required)

These components may be written in any high-level language. The RPC Run-Time Library (RTL) routines are written in the C language.

7.3. What You Need to Do

The following steps summarize what you need to do to build a distributed application:

1. Design the application.
2. Write an RPC interface definition. Compile it using RPCGEN, then edit the output files as necessary. (This step is optional. An RPC interface definition is not required. If you do not write one, proceed to step 3.)
3. Write any necessary code that RPCGEN did not generate.
4. Compile the RPCGEN output files, server procedures, and main program using the appropriate language compiler(s). RPCGEN output files must be compiled using HP C.
5. Link the object code, making sure you link in the RPC RTL.
6. Start the Port Mapper on the server host.
7. Execute the client and server programs.

Step 1: Design the Application

You must write a main (client) program and at least one server procedure. The network interface, however, may be hand-written or created by RPCGEN. The network interface files contain client and server stubs, header files, and XDR routines. You may edit any files that RPCGEN creates.

When deciding whether to write the network interface yourself, consider these factors:

Is execution time critical?	Your hand-written code may execute faster than code that RPCGEN creates.
Which RPC interface layer do you want to use?	RPCGEN permits you to use only the highest layer interface. If you want to use the lower layers, you must write original code.
Which transport protocol do you want to use?	

You may write your own XDR programs, but it is usually best to let RPCGEN handle these.

Step 2: Write and Compile the Interface Definition

An interface definition is a program the RPCGEN compiler accepts as input. See Chapter 8, ***RPCGEN** Compiler* for more information about interface definitions.

Interface definitions are optional. If you write the all of the network interface code yourself, you do not need an interface definition.

You must write an interface definition if you want RPCGEN to generate network interface code.

After compiling the interface definition, edit the output file(s).

If you are not writing an interface definition, skip this step and proceed to Step 3.

Step 3: Write the Necessary Code

Write any necessary code that RPCGEN did not create for you. Table 7.2, “Coding References” lists the texts you may use as references.

Table 7.2. Coding References

Reference	Purpose
RFC 1057	Defines the RPC language. Use for writing interface definitions.
RFC 1014	Defines the XDR language. Use for writing XDR filter routines.
Chapter 10, *RPC RTL Client Routines*	Defines each routine in the RPC RTL. Use for writing stub procedures and XDR filter routines.

Step 4: Compile All Files

Compile the RPCGEN output files, server procedures, and main program separately.

HP C (Alpha):

```
$ CC /STANDARD=RELAXED /WARNING=DISABLE=(IMPLICITFUNC) filename.C
```

Step 5: Link the Object Code

Link the object code files. Make sure you link in the RPC RTL. Use the following command.

HP C (Alpha):

```
$ LINK filenames, SYS$INPUT /OPTIONS
TCPIP$RPCXDR_SHR /SHARE
SYS$SHARE:DECC$SHR /SHARE
Ctrl/Z
```

After entering the command, press **Ctrl/Z**.

To avoid repetitive data entry, you may create an OpenVMS command procedure to execute these commands.

Step 6: Start the Port Mapper

The Port Mapper must be running on the server host. If it is not running, use the **IP CONFIGURE / SERVER** command to start it. If you want to generate your own screen shot, you can use CRASH. Then all you have to do is change the user-entered items to **bold**, and change the V10.5 (42) to V10.5 (nnn) in the banner line.

Step 7: Execute the Client and Server Programs

Perform these steps:

1. Run the server program interactively to debug it, or using the **/DETACHED** qualifier. Refer to the VSI documentation for details.

2. Run the client main program.

7.4. Obtaining RPC Information

You can: request a listing of all programs registered with a Port Mapper.

7.4.1. Requesting a Program Listing

To request a listing of all programs that are registered with the Port Mapper, enter the **IP SHOW / RPC_PORTMAP** command in the following format at the DCL prompt:

```
$ IP SHOW /RPC_PORTMAP
```

If you add **/REMOTE_HOST=*hostname*** to this command:

```
$ IP SHOW /RPC_PORTMAP /REMOTE_HOST=[host-name]
```

Specify the domain name of the host on which the Port Mapper resides. If you omit this parameter, RPC uses the name of the local host. Example 7.1, “Sample RPC Information Output” shows an example.

Example 7.1. Sample RPC Information Output

```
$ IP SHOW/RPC_PORTMAP
```

```
VSI TCP/IP for OpenVMS registered RPC programs:
Program    Version   Protocol   Port
-------    -------   --------   ----
NLOCKMGR   3         TCP        2049
NLOCKMGR   1         TCP        2049
NLOCKMGR   3         UDP        2049
NLOCKMGR   1         UDP        2049
NFS        2         TCP        2049
NFS        2         UDP        2049
MOUNT      1         TCP        1024
MOUNT      1         UDP        1028
STATUS     1         TCP        1024
STATUS     1         UDP        1024
```

Chapter 8. RPCGEN Compiler

8.1. Introduction

This chapter is for RPC programmers.

8.2. What Is RPCGEN?

RPCGEN is the RPC Protocol Compiler. This compiler creates the network interface portion of a distributed application, effectively hiding from the programmer the details of writing and debugging low-level network interface code.

You are not required to use **RPCGEN** when developing a distributed application. If speed and flexibility are critical to your application, you can write the network interface code yourself, using RPC Run-Time Library (RTL) calls where they are needed.

Compiling with **RPCGEN** is one step in developing distributed applications. See Chapter 7, *Building Distributed Applications with RPC* for a complete description of the application development process.

RPCGEN allows you to use the highest layer of the RPC programming interface. The Chapter 6, *RPC Fundamentals* provides details on these layers.

8.3. Software Requirements

The following software must be installed on your system before you can use RPCGEN:

- OpenVMS Version 8.4-2L1
- HP C compiler Version 3.2 or later

8.4. Input Files

The **RPCGEN** compiler accepts as input programs called `interface definitions`, written in RPC Language (RPCL), an extension of XDR language. RFC 1057 and RFC 1014 describe these languages in detail.

An interface definition must always contain the following information:

- Remote program number
- Version number of the remote program
- Remote procedure number(s)
- Input and output arguments

Example 8.1, “Interface Definition” shows a sample interface definition.

Example 8.1. Interface Definition

```
/*
** RPCGEN input file for the print file RPC batching example.
**
```

```
** This file is used by RPCGEN to create the files PRINT.H and PRINT_XDR.C
** The client and server files were developed from scratch.
*/

const MAX_STRING_LEN = 1024;   /* maximum string length */

/*
** This is the information that the client sends to the server
*/
struct a_record
{
  string            ar_buffer< MAX_STRING_LEN>;
};

program PRINT_FILE_PROG
{   version PRINT_FILE_VERS_1
    {
    void              PRINT_RECORD( a_record) = 1;
    u_long SHOW_COUNT( void) = 2;
    } = 1;
} = 0x20000003;
/* end file PRINT.X */
```

The default extension for **RPCGEN** input files is .X.

You do not need to call the RPC RTL directly when writing an interface definition. **RPCGEN** inserts the necessary library calls in the output file.

8.5. Output Files

RPCGEN output files contain code in C language. Table 8.1, "**RPCGEN** Output Files" lists the **RPCGEN** output files and summarizes their purpose. You can edit **RPCGEN** output files during application development.

Table 8.1. RPCGEN Output Files

File	Purpose
Client and server stub calls	Interface between the network and the client and server programs. Stubs use RPC RTL to communicate with the network.
XDR routines	Convert data from a machine's local data format to XDR for mat, and vice versa.
Header	Contains common definitions, such as those needed for any structures being passed.

`Invoking RPC` explains how to request specific output files.

Table 8.2, "**RPCGEN** File Naming Conventions" shows the conventions you should use to name output files.

Table 8.2. RPCGEN File Naming Conventions

File	Output Filename
Client stub	*`inputname`*`_CLNT.C`
Server stub	*`inputname`*`_SVC.C`

File	Output Filename
Header file	*inputname*.H
XDR filter routines	*inputname*_XDR.C

inputname is the name of the input file. For example, if the input file is TEST.X, the server stub is TEST_SVC.C.

When you use the **RPCGEN** command to create all output files at once, **RPCGEN** creates the output filenames listed in Table 8.2, "**RPCGEN** File Naming Conventions" by default. When you want to create specific kinds of output files, you must specify the names of the output files in the command line.

8.6. Preprocessor Directives

RPCGEN runs the input files through the C preprocessor before compiling. You can use the macros listed in Table 8.3, "Macros" with the #ifdef preprocessor directive to indicate that specific lines of code in the input file are to be used only for specific **RPCGEN** output files.

Table 8.3. Macros

File	Macro
Client stub	RPC_CLNT
Server stub	RPC_SVC
Header file	RPC_HDR
XDR filter routines	RPC_XDR

8.7. Invoking RPCGEN

This section explains how to invoke **RPCGEN** to create:

- All output files at once
- Specific output files
- Server stubs for either the TCP or UDP transport

8.7.1. Creating All Output Files at Once

This command creates all four **RPCGEN** output files at once:

RPCGEN *input*

where *input* is the name of the file containing the interface definition.

In the following example, **RPCGEN** creates the output files PROGRAM.H, PROGRAM_CLNT.C, PROGRAM_SVC.C, and PROGRAM_XDR.C:

```
RPCGEN PROGRAM.X
```

8.7.2. Creating Specific Output Files

This command creates only the **RPCGEN** output file that you specify:

`RPCGEN` { -c | -h | -l | -m } [-o *`output`*] *`input`*

-c	Creates an XDR filter file (`_XDR.C`)
-h	Creates a header file (`.H`)
-l	Creates a client stub (`_CLNT.C`)
-m	Creates a server stub (`_SVC.C`) that uses both the UDP and TCP transports
-o	Specifies an output file (or the terminal if no output file is given)
output	Name of the output file
input	Name of an interface definition file with a `.X` extension

Follow these guidelines:

- Specify just one output file (-c, -h, -l, or -m) in a command line
- If you omit the output file, **RPCGEN** sends output to the terminal screen

8.7.3. Examples:

1. `RPCGEN -h PROGRAM`

   RPCGEN accepts the file `PROGRAM.X` as input and sends the header file output to the screen, because no output file is specified.

2. `RPCGEN -l -o PROGRAM_CLNT.C PROGRAM.X`

   RPCGEN accepts the `PROGRAM.X` file as input and creates the `PROGRAM_CLNT.C` client stub file.

3. `RPCGEN -m -o PROGRAM_SVC.C PROGRAM.X`

   RPCGEN accepts the `PROGRAM.X` file as input and creates the `PROGRAM_SVC.C` server stub file. The server can use both the UDP and TCP transports.

8.7.4. Creating Server Stubs for TCP or UDP Transports

This command creates a server stub file for either the TCP or UDP transport:

`RPCGEN` -s { udp | tcp } [-o *`output`*] *`input`*

-s	Creates a server (_SVC.C) that uses either the UDP or TCP transport (with -s, you must specify either **udp** or **tcp**; do not also use -m)
udp	Creates a UDP server
tcp	Creates a TCP server
-o	Specifies an output file (or the terminal if no output file is given)
output	Name of the output file
input	Name of an interface definition file with a .X extension

If you omit the output file, **RPCGEN** sends output to the terminal screen.

In this example, **RPCGEN** accepts the `PROGRAM.X` file as input and creates the `PROGRAM_SVC.C` output file, containing a TCP server stub:

RPCGEN -s tcp -o PROGRAM_SVC.C PROGRAM.X

8.8. Error Handling

RPCGEN stops processing when it encounters an error. It indicates which line the error is on.

8.9. Restrictions

RPCGEN does not support the following:

- The syntax int x, y; . You must write this as `int x int y;`

Chapter 9. RPC RTL Management Routines

9.1. Introduction

This chapter is for RPC programmers. It introduces RPC Run-Time Library (RTL) conventions and documents the management routines in the RPC RTL. These routines are the programming interface to RPC.

9.2. Management Routines

The RPC RTL contains:

- RPC management routines
- RPC client and server routines for the UDP and TCP transport layers
- On Alpha systems, RPC provides a single shareable image accessed via the `TCPIP$RPCXDR_SHR` logical. This shareable image contains routines for all of the HP C floating-point types. The correct routines will be called automatically based on the compiler options used to compile the RPC application. See the VSI C documentation for how to use the floating-point compiler options.

Chapter 7, *Building Distributed Applications with RPC* explains how to link in the RPC RTL.

9.3. Routine Name Conventions

In this chapter, all routines are documented according to their standard UNIX names.

9.4. Header Files

All RPC programs include the file named `RPC.H`. Locations for this file are `TCPIP$RPC:RPC.H`.

The `RPC.H` file includes the files listed in Table 9.1, “Header Files Included In RPC.H”.

Table 9.1. Header Files Included In RPC.H

Filename	Purpose
`AUTH.H`	Used for authentication.
`AUTH_UNIX.H`	Contains XDR definitions for UNIX-style authentication.
`CLNT.H`	Contains various RPC client definitions.
`IN.H`	Defines structures for the internet and socket addresses (**in_addrand** and **sockaddr_in**). This file is part of the C Socket Library.
`RPC_MSG.H`	Defines the RPC message format.
`SVC.H`	Contains various RPC server definitions.
`SVC_AUTH.H`	Used for server authentication.

Filename	Purpose
TYPES.H	Defines UNIX C data types.
XDR.H	Contains various XDR definitions.
NETDB.H	Defines structures and routines to parse /etc/rpc.

There is an additional header file not included by RPC.H that is used by **xdr_pmap** and **xdr_pmaplist** routines. The file name is pmap_prot.h, and the location is:

TCPIP$RPC:PMAP_PROT.H

9.5. Management Routines

RPC management routines retrieve and maintain information that describes how a process is using RPC.This section describes each management routine and function in detail. The following information is provided for each routine:

- Format
- Arguments
- Description
- Diagnostics, or status codes returned, if any

The management routines are

- **get_myaddress**
- **getrpcbynumber**
- **getrpcport**

get_myaddress

get_myaddress — Returns the internet address of the local host.

Format

```
#include
void get_myaddress (struct sockaddr_in *addr);
```

Argument

addr

Address of a sockaddr_in structure that will be loaded with the host internet address. The port number is always set to htons(PMAPPORT).

Description

The **get_myaddress** routine returns the internet address of the local host without doing any name translation or DNS lookups.

getrpcbynumber

getrpcbynumber — Gets an RPC entry.

Format

```
#include
struct rpcent *getrpcbynumber(number)
int number;
```

Argument

number

Program name or number.

Description

The **getrpcbynumber** routine returns a pointer to an object with the following structure containing the broken-out fields of a line in the RPC program number database, /etc/rpc.

```
struct rpcent {
 char *r_name;       /* name of server for this RPC program */
 char **r_aliases;   /* alias list */
 long r_number;      /* RPC program number */
 };
```

The members of this structure are:

r_name	Name of the server for this RPC program
r_aliases	Zero-terminated list of alternate names for the RPC program
r_number	RPC program number for this service

The getrpcbynumber routine sequentially searches from the beginning of the file until a matching RPC program name or program number is found, or until an EOF is encountered.

Diagnostics

A NULL pointer is returned on EOF or error.

getrpcport

getrpcport — Gets an RPC port number.

Format

```
int getrpcport(host, prognum, versnum, proto)
char *host;
int prognum, versnum, proto;
```

Arguments

host

Host running the RPC program.

prognum

Program number.

proto

Protocol name. Must be `IPPROTO_TCP` or `IPPROTO_UDP`.

Description

The getrpcport routine returns the port number for version *versnum* of the RPC program *prognum* running on *host* and using protocol *proto*.

It returns 0 if it cannot contact the portmapper, or if *prognum* is not registered. If *prognum* is registered but not with *versnum*, it still returns a port number (for some version of the program), indicating that the program is indeed registered. The version mismatch is detected on the first call to the service.

Chapter 10. RPC RTL Client Routines

10.1. Introduction

This chapter is for RPC programmers. It documents the client routines in the RPC Run-Time Library (RTL). These routines are the programming interface to RPC.

10.2. Common Arguments

Many client, Port Mapper, and server routines use the same arguments.

Table 10.1, “Common Arguments” lists these arguments and defines their purpose. Arguments that are unique to each routine are documented together with their respective routines in this and the following chapters

Table 10.1. Common Arguments

Argument	Purpose
args_ptr	Address of the buffer to contain the decoded RPC arguments.
auth	RPC authentication client handle created by the **authnone_create**, **authunix_create**, or **authunix_create_default** routine.
clnt	Client handle returned by any of the client create routines.
in	Input arguments for the service procedure.
inproc	XDR routine that encodes input arguments.
out	Results of the remote procedure call.
outproc	XDR routine that decodes output arguments.
procnum	Number of the service procedure.
prognum	Program number of the service program.
protocol	Transport protocol for the service. Must be `IPPROTO_UDP` or `IPPROTO_TCP`.
s	String containing the message of your choice. The routines append an error message to this string.
sockp	Socket to be used for this remote procedure call. If *`sockp`* is `RPC_ANYSOCK`, the routine creates a new socket and defines *`sockp`*. The **clnt_destroy** routine closes the socket. If *`sockp`* is a value other than `RPC_ANYSOCK`, the routine uses this socket and ignores the internet address of the server.
versnum	Version number of the service program.
xdr_args	XDR procedure that describes the RPC arguments.
xdrs	Structure containing XDR encoding and decoding information.
xprt	RPC server handle.

10.3. Client Routines

The client routines are called by the client main program or the client stub procedures.

The following sections describe each client routine in detail. The client routines are

auth_destroy	**clnt_destroy**
authnone_create	**clnt_geterr**
authunix_create	**clnt_pcreateerror / clnt_spcreateerror**
authunix_create_default	**clnt_perrno / clnt_sperrno**
callrpc	**clnt_perror / clnt_sperror**
clnt_broadcast	**clntraw_create**
clnt_call	**clnttcp_create**
clnt_control	**clntudp_create / clntudp_bufcreate**
clnt_create	

auth_destroy

auth_destroy — A macro that destroys authentication information associated with an authentication handle.

Format

```
void auth_destroy (AUTH *auth)
```

Argument

auth

RPC authentication client handle created by the **authnone_create**, **authunix_create**, or **authunix_create_default** routine.

Description

Use **auth_destroy** to free memory that was allocated for authentication handles. This routine undefines the value of *auth* by deallocating private data structures.

Do not use this memory space after **auth_destroy** has completed. You no longer own it.

See Also

authnone_create, **authunix_create**, **authunix_create_default**

authnone_create

authnone_create — Creates and returns a null RPC authentication handle for the client process.

Format

```
#include

AUTH *authnone_create();
```

Arguments

None.

Description

This routine is for client processes that require no authentication. RPC uses it as a default when it creates a client handle.

See Also

authunix_create_default, **clnt_create**, **clntraw_create**, **clnttcp_create**, **clntudp_create / clntudp_bufcreate**

authunix_create

authunix_create — Creates and returns an RPC authentication handle for the client process. Use this routine when the server requires UNIX-style authentication.

Format

```
#include

AUTH *authunix_create (char *host, int uid,int gid,int len,int
gids);
```

Arguments

host

Address of the name of the host that created the authentication information. This is usually the local host running the client process.

uid

User ID of the person who is executing this process.

gid

User's group ID.

len

Number of elements in the **gids* array.

gids

Address of the array of groups to which the user belongs.

Description

Since the client does not validate the *uid* and *gid*, it is easy to impersonate an unauthorized user. Choose values the server expects to receive. The application must provide OpenVMS-to-UNIX authorization mapping.

You can use a Socket Library lookup routine to get the host name.

See Also

authnone_create, **authunix_create_default**

authunix_create_default

authunix_create_default — Calls the **authunix_create** routine and provides default values as arguments.

Format

```
#include

AUTH *authunix_create_default()
```

Arguments

See below.

Description

Like the **authunix_create** routine, **authunix_create_default** provides UNIX-style authentication for the client process. However, **authunix_create_default** does not require you to enter any arguments. Instead, this routine provides default values for the arguments used by **authunix_create**, listed in Table 10.2, “Default Arguments”.

Table 10.2. Default Arguments

Argument	Default Value
host	local host domain name
uid	getuid ( )
gid	getgid ( )
len	0
gids	0

You can replace this call with **authunix_create** and provide appropriate values.

Example

```
auth_destroy(client->cl_auth);
client->cl_auth = authunix_create_default();
```

This example overrides the **authnone_create** routine, where *client* is the value returned by the **clnt_create**, **clntraw_create**, **clnttcp_create**, or **clntudp_create / clntudp_bufcreate** routine.

See Also

callrpc

callrpc

callrpc

Format

```
#include

int callrpc (char *host,u_long prognum,u_long versnum,u_long
procnum,xdrproc_t inproc,u_char *in, xdrproc_t outproc,u_char *out);
```

Arguments

host

Host where the procedure resides.

prognum, *versnum*, *procnum*, *inproc*, *in*, *outproc*, *out*

See Table 10.1, “Common Arguments” for a description of the above arguments.

Description

The **callrpc** routine performs the same functions as the **clnt_create**, and **clnt_destroy** routines.

Since the **callrpc** routine uses the UDP transport protocol, messages can be no larger than 8Kbytes. This routine does not allow you to control timeouts or authentication.

If you want to use the TCP transport, use the **clnt_create** or **clnttcp_create** routine.

Diagnostics

The **callrpc** routine returns zero if it succeeds, and the value of *enum clnt_stat* cast to an integer if it fails.

You can use the **clnt_perrno / clnt_sperrno** routine to translate failure status codes into messages.

See Also

clnt_broadcast, **clnt_call**, **clnt_create**, **clnt_destroy**, **clnt_perrno / clnt_sperrno**, **clnttcp_create**

clnt_broadcast

clnt_broadcast — Broadcasts a remote procedure call to all local networks, using the broadcast address.

Format

```
#include

enum clnt_stat clnt_broadcast (u_long prognum, u_long versnum,
u_long procnum, xdrproc_t inproc, u_char *in, xdrproc_t outproc,
u_char *out, resultproc_t eachresult);
```

Arguments

progNum, *versnum*, *procnum*, *inproc*, *in*, *outproc*, *out*

See Table 10.1, “Common Arguments” for a description of the above arguments.

eachresult

Each time **clnt_broadcast** receives a response, it calls the *eachresult* routine. If *eachresult* returns zero, **clnt_broadcast** waits for more replies. If *eachresult* returns a nonzero value, **clnt_broadcast** stops waiting for replies. The *eachresult* routine uses this form:

```
int eachresult(out, addr)
u_char *out;
struct sockaddr_in *addr;
```

out	Contains the results of the remote procedure call, in the local data format.
*addr	Is the address of the host that sent the results.

Description

The **clnt_broadcast** routine performs the same functions as the **callrpc** routine. However, **clnt_broadcast** sends a message to all local networks, using the broadcast address. The **clnt_broadcast** routine uses the UDP protocol.

Table 10.3, “Maximum Message Size” indicates how large a broadcast message can be.

Table 10.3. Maximum Message Size

Line	Maximum Size
Ethernet	1500 bytes
proNet	2044 bytes

Diagnostics

This routine returns diagnostic values defined in the CLNT.H file for enum clnt_stat.

See Also

callrpc, **clnt_perrno / clnt_sperrno**

clnt_call

clnt_call — A macro that calls a remote procedure.

Format

```
enum clnt_stat clnt_call (CLIENT *clnt, u_long procnum, xdrproc_t
inproc, u_char *in, xdrproc_t outproc, u_char *out, struct timeval
tout);
```

Arguments

`clnt, procnum, inproc, in, outproc, out`

See Table 10.1, “Common Arguments” for a description of the above arguments.

`tout`

Time allowed for the results to return to the client, in seconds and microseconds. If you use the **clnt_control** routine to change the `CLSET_TIMEOUT` code, this argument is ignored.

Description

Use the **clnt_call** routine after using **clnt_create**. After you have finished with the client handle, use the **clnt_destroy** routine. You can use the **clnt_perror / clnt_sperror** routine to print messages for any errors that occurred.

Diagnostics

This routine returns diagnostic values defined in the `CLNT.H` file for enum clnt_stat.

See Also

clnt_control, **clnt_create**, **clnt_destroy**, **clnt_perrno / clnt_sperrno**

clnt_control

clnt_control — A macro that changes or retrieves information about an RPC client process.

Format

```
bool_t clnt_control (CLIENT *clnt, u_long code, void *info);
```

Arguments

`clnt`

Client handle returned by any of the client create routines.

`code`

Code listed in Table 10.4, “Valid Codes”.

Table 10.4. Valid Codes

Code	Type	Purpose
`CLSET_TIMEOUT`	struct timeval	Set total timeout
`CLGET_TIMEOUT`	struct timeval	Get total timeout
`CLSET_RETRY_TIMEOUT*`	struct timeval	Set retry timeout
`CLGET_RETRY_TIMEOUT*`	struct timeval	Get retry timeout
`CLGET_SERVER_ADDR`	struct sockaddr_in	Get server address

Code	Type	Purpose
* Valid only for the UDP transport protocol.		

The timeval is specified in seconds and microseconds. The total timeout is the length of time that the client waits for a reply. The default total timeout is 25 seconds.

The retry time is the length of time that UDP waits for the server to reply before transmitting the request. The default retry timeout is 5 seconds. You might want to increase the retry time if your network is slow.

For example, suppose the total timeout is 10 seconds and the retry time is five seconds. The client sends the request and waits five seconds. If the client does not receive a reply, it sends the request again. If the client does not receive a reply within five seconds, it does not send the request again.

If you use `CLSET_TIMEOUT` to set the timeout, the **clnt_call** routine ignores the timeout parameter it receives for all future calls.

info

Address of the information being changed or retrieved.

Diagnostics

This routine returns TRUE if it succeeds, and FALSE if it fails.

See Also

clnt_call, **clnt_create**, **clnt_destroy**, **clntraw_create**, **clnttcp_create**, **clntudp_create** / **clntudp_bufcreate**

clnt_create

clnt_create — Creates an RPC client handle.

Format

```
#include

CLIENT *clnt_create (char *host, u_long prognum, u_long versnum,
char *proto);
```

Arguments

host

Address of the string containing the name of the remote host where the server is located.

prognum, versnum

See Table 10.1, “Common Arguments” for a description of the above arguments.

proto

Address of a string containing the name of the transport protocol. Valid values are **UDP** and **TCP**.

Description

The **clnt_create** routine creates an RPC client handle for *prognum*. An RPC client handle is a structure containing information about the RPC client. The client can use the UDP or TCP transport protocol.

This routine uses the Port Mapper. You cannot control the local port.

The default sizes of the send and receive buffers are 8800 bytes for the UDP transport, and 4000 bytes for the TCP transport.

The retry time for the UDP transport is five seconds.

Use the **clnt_create** routine instead of the **callrpc** or **clnt_broadcast** routines if you want to use one of the following:

- The TCP transport
- An authentication other than null
- More than one active client at the same time

You can also use **clntraw_create** to use the IP protocol, **clnttcp_create** to use the TCP protocol, or **clntudp_create / clntudp_bufcreate** to use the UDP protocol.

The **clnt_create** routine uses the global variable `rpc_createerr`. `rpc_createerr` is a structure that contains the most recent service creation error. Use `rpc_createerr` if you want the client program to handle the error. The value of `rpc_createerr` is set by any RPC client creation routine that does not succeed.

The `rpc_createerr` variable is defined in the `CLNT.H` file.

Diagnostics

The **clnt_create** routine returns the address of the client handle, or zero (if it could not create the client handle).

If the **clnt_create** routine fails, you can use the**clnt_pcreateerror / clnt_spcreateerror** routine to obtain diagnostic information.

See Also

clnt_call, **clnt_control**, **clnt_destroy**, **clntraw_create**, **clnt_pcreateerror / clnt_spcreateerror**, **clnttcp_create**, **clntudp_create / clntudp_bufcreate**

clnt_destroy

clnt_destroy — A macro that destroys an RPC client handle.

Format

```
void clnt_destroy (CLIENT *clnt);
```

Argument

clnt

Client handle returned by any of the client create routines.

Description

The **clnt_destroy** routine destroys the client's RPC handle by deallocating all memory related to the handle. The client is undefined after the **clnt_destroy** call.

If the **clnt_create** routine had previously opened a socket, this routine closes the socket. Otherwise, the socket remains open.

See Also

clnt_create, **clntraw_create**, **clnttcp_create**, **clntudp_create / clntudp_bufcreate**

clnt_geterr

clnt_geterr — A macro that returns an error code indicating why an RPC call failed.

Format

```
void clnt_geterr (CLIENT *clnt, struct rpc_err *errp);
```

Arguments

clnt

Client handle returned by any of the client create routines.

errp

Address of the structure containing information that indicates why an RPC call failed. This information is the same as *clnt_stat* contains, plus one of the following: the C error number, the range of server versions supported, or authentication errors.

Description

This routine is primarily for internal diagnostic use.

Example

```
#define PROGRAM   1
#define VERSION   1

 CLIENT  *clnt;
 struct rpc_err  err;

 clnt = clnt_create( "server name", PROGRAM, VERSION, "udp");

 /* calls to RPC library */
```

```
clnt_geterr( clnt, &err);
```

This example creates a UDP client handle and performs some additional RPC processing. If an RPC call fails, **clnt_geterr** returns the error code.

See Also

clnt_perror / clnt_sperror

clnt_pcreateerror / clnt_spcreateerror

clnt_pcreateerror / clnt_spcreateerror — Return a message indicating why RPC could not create a client handle.

Format

```
#include

void clnt_pcreateerror (char *s); char *clnt_spcreateerror (char
*s);
```

Argument

s

String containing the message of your choice. The routines append an error message to this string.

Description

The **clnt_pcreateerror / clnt_spcreateerror** routine prints a message to SYS$OUTPUT.

The **clnt_pcreateerror / clnt_spcreateerror** routine returns the address of a string. Use this routine if:

- You want to save the string.
- You do not want to use *printf* to print the message.
- The message format is different from the one that **clnt_perrno / clnt_sperrno** supports.

The **clnt_pcreateerror / clnt_spcreateerror** routine overwrites the string it returns, unless you save the results.

Use these routines when the **clnt_create**, **clntraw_create**, **clnttcp_create**, or **clntudp_create / clntudp_bufcreate** routine fails.

See Also

clnt_create, **clntraw_create**, **clnttcp_create**, **clntudp_create / clntudp_bufcreate**

clnt_perrno / clnt_sperrno

clnt_perrno / clnt_sperrno — Return a message indicating why the **callrpc** or **clnt_broadcast** routine failed to create a client handle.

Format

```
#include

void clnt_perrno (enum clnt_stat stat); char *clnt_sperrno (enum
clnt_stat stat);
```

Argument

stat

Appropriate error condition. Values for *stat* are defined in the CLNT.H file.

Description

The **clnt_perrno / clnt_sperrno** routine prints a message to SYS$OUTPUT.

The **clnt_perrno / clnt_sperrno** routine returns the address of a string. Use this routine instead if:

- You want to save the string.
- You do not want to use printf to print the message.
- The message format is different from the one that **clnt_perrno / clnt_sperrno** supports.

To save the string, copy it into your own memory space.

See Also

callrpc, **clnt_broadcast**

clnt_perror / clnt_sperror

clnt_perror / clnt_sperror — Return a message if the **clnt_call** routine fails.

Format

```
#include

void clnt_perror (CLIENT *clnt, char *s); char *clnt_sperror (CLIENT
*clnt, char *s);
```

Arguments

clnt

See Table 10.1, “Common Arguments” for a description of the above argument. String containing the message to output.

Description

Use these routines after **clnt_call**.

The **clnt_perror / clnt_sperror** routine prints an error message to SYS$OUTPUT.

The **clnt_perror / clnt_sperror** routine returns a string. Use this routine if:

- You want to save the string.
- You do not want to use *printf* to print the message.
- The message format is different from the one that **clnt_perror / clnt_sperror** supports.

The **clnt_perror / clnt_sperror** routine overwrites the string with each call. Copy the string into your own memory space if you want to save it.

See Also

clnt_call, **clnt_create**, **clntraw_create**, **clnttcp_create**, **clntudp_create / clntudp_bufcreate**

clntraw_create

clntraw_create — Returns an RPC client handle. The remote procedure call uses the IP transport.

Format

```
#include

CLIENT *clntraw_create (struct sockaddr_in *addr, u_long prognum,
u_long versnum, int *sockp, u_long sendsize, u_long recvsize);
```

Arguments

addr, prognum, versnum

See Table 10.1, “Common Arguments”for a description of the above arguments.

sockp

Socket to be used for this remote procedure call. *sockp* can specify the local address and port number. If *sockp* is RPC_ANYSOCK, then a port number is assigned. The example shown for the **clntudp_create / clntudp_bufcreate** routine shows how to set up *sockp* to specify a port. See Table 10.1, “Common Arguments” for a description of sockp and RPC_ANYSOCK.

addr

Internet address of the host on which the server resides.

sendsize

Size of the send buffer. If you enter a value less than 100, then 4000 is used as the default.

recvsize

Size of the receive buffer. If you enter a value less than 100, then 4000 is used as the default.

Description

The **clntraw_create** routine creates an RPC client handle for *addr*, *prognum*, and *versnum*. The client uses the IP transport. The routine is similar to the **clnt_create** routine, except clnttcp_create

allows you to specify a socket and buffer sizes. If you specify the port number as zero by using `addr->sin_port`, the Port Mapper provides the number of the port on which the remote program is listening.

The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding RPC server should live in the same address space (see also **svcraw_create**). This allows simulation of RPC and getting RPC overheads, such as round trip times, without kernel interference.

The **clnttcp_create** routine uses the global variable `rpc_createerr`, which is a structure that contains the most recent service creation error. Use rpc_createerr if you want the client program to handle the error. The value of rpc_createerr is set by any RPC client creation routine that does not succeed. The rpc_createerr variable is defined in the `CLNT.H` file.

Diagnostics

The **clntraw_create** routine returns the address of the client handle, or zero (if it could not create the client handle). If the routine fails, use the **clnt_pcreateerror / clnt_spcreateerror** routine to obtain additional diagnostic information.

See Also

clnt_call, **clnt_control**, **clnt_create**, **clnt_destroy**, **clnt_pcreateerror / clnt_spcreateerror**, **clnttcp_create**, **clntudp_create / clntudp_bufcreate**

clnttcp_create

clnttcp_create — Returns an RPC client handle. The remote procedure call uses the TCP transport.

Format

```
#include

CLIENT *clnttcp_create (struct sockaddr_in *addr, u_long prognum,
u_long versnum, int *sockp, u_long sendsize, u_long recvsize);
```

Arguments

addr, prognum, versnum

See Table 10.1, “Common Arguments” for a description of the above arguments.

sockp

Socket to be used for this remote procedure call. *sockp* can specify the local address and port number. If *sockp* is `RPC_ANYSOCK`, then a port number is assigned. The example shown for the **clntudp_create / clntudp_bufcreate** routine shows how to set up *sockp* to specify a port. See Table 10.1, “Common Arguments” for a description of *sockp* and `RPC_ANYSOCK`.

addr

Internet address of the host on which the server resides.

sendsize

Size of the send buffer. If you enter a value less than 100, then 4000 is used as the default.

recvsize

Size of the receive buffer. If you enter a value less than 100, then 4000 is used as the default.

Description

The **clnttcp_create** routine creates an RPC client handle for *addr*, *prognum*, and *versnum*. The client uses the TCP transport. The routine is similar to the **clnt_create** routine, except **clnttcp_create** allows you to specify a socket and buffer sizes. If you specify the port number as zero by using `addr->sin_port`, the Port Mapper provides the number of the port on which the remote program is listening.

The **clnttcp_create** routine uses the global variable `rpc_createerr`. `rpc_createerr` is a structure that contains the most recent service creation error. Use `rpc_createerr` if you want the client program to handle the error. The value of `rpc_createerr` is set by any RPC client creation routine that does not succeed. The `rpc_createerr` variable is defined in the `CLNT.H` file.

Diagnostics

The **clnttcp_create** routine returns the address of the client handle, or zero (if it could not create the client handle). If the routine fails, use the **clnt_pcreateerror** / **clnt_spcreateerror** routine to obtain additional diagnostic information.

See Also

clnt_call, **clnt_control**, **clnt_create**, **clnt_destroy**, **clnt_pcreateerror** / **clnt_spcreateerror**, **clntudp_create** / **clntudp_bufcreate**

clntudp_create / clntudp_bufcreate

clntudp_create / **clntudp_bufcreate** — Returns an RPC client handle. The remote procedure call uses the UDP transport.

Format

```
#include

CLIENT *clntudp_create (struct sockaddr_in *addr, u_long prognum,
u_long versnum, struct timeval wait, int *sockp);

CLIENT *clntudp_bufcreate (struct sockaddr_in *addr, u_long prognum,
u_long versnum, struct timeval wait, int *sockp, u_long sendsize,
u_long recvsize);
```

Arguments

addr

Internet address of the host on which the server resides.

prognum, *versnum*, *sockp*

See Table 10.1, “Common Arguments” for a description of the above arguments.

wait

Time interval the client waits before resending the call message. This value changes the CLSET_RETRY_TIMEOUT code. The **clnt_call** routine uses this value.

sendsize

Size of the send buffer. If you enter a value less than 100, then 4000 is used as the default.

recvsize

Size of the receive buffer. If you enter a value less than 100, then 4000 is used as the default.

Description

These routines create an RPC client handle for *addr*, *prognum*, and *versnum*. The client uses the UDP transport protocol.

If you specify the port number as zero by using addr->sin_port, the Port Mapper provides the number of the port on which the remote program is listening.

Note

Use the **clntudp_create / clntudp_bufcreate** routine only for procedures that handle messages shorter than 8K bytes. Use the **clntudp_create / clntudp_bufcreate** routine for procedures that handle messages longer than 8K bytes.

The **clntudp_create / clntudp_bufcreate** routine uses the global variable rpc_createerr. rpc_createerr is a structure that contains the most recent service creation error. Use rpc_createerr if you want the client program to handle the error. The value of rpc_createerr is set by any RPC client creation routine that does not succeed.

The rpc_createerr variable is defined in the CLNT.H file.

Example

```
main()
{
  int  sock;
  u_long  prog = PROGRAM, vers = VERSION;
  CLIENT  *clnt;
  struct sockaddr_in  local_addr, remote_addr;
  struct timeval  timeout = { 35, 0},
    retry = { 5, 0};
  remote_addr.sin_family = AF_INET;
  remote_addr.sin_port = 0;  /* consult the remote port mapper */
  remote_addr.sin_addr.s_addr = 0x04030201; /* internet
  addr 1.2.3.4 */
  local_addr.sin_family = AF_INET;
  local_addr.sin_port = 12345;   /* use port 12345 */
  local_addr.sin_addr.s_addr = 0x05030201;  /* internet addr
         1.2.3.5 */
  sock = socket( AF_INET, SOCK_DGRAM, 0);
```

```
  /* bind the socket to the local addr */
  bind( sock, &local_addr, sizeof( local_addr));
  /* create a client that uses the local IA and port given above */
  clnt = clntudp_create( &remote_addr, prog, vers, retry, &sock);
  /* use a connection timeout of 35 seconds, not the default */
  clnt_control( clnt, CLSET_TIMEOUT, &timeout);
  /*call the server here*/
}
```

This example defines a socket structure, binds the socket, and creates a UDP client handle.

Diagnostics

These routines return the address of the client handle, or zero (if they cannot create the client handle).

If these routines fail, you can obtain additional diagnostic information by using the **clnt_pcreateerror / clnt_spcreateerror** routine.

See Also

clnt_call, **clnt_control**, **clnt_create**, **clnt_destroy**, **clnt_pcreateerror / clnt_spcreateerror**, **clnttcp_create**

Chapter 11. RPC RTL Port Mapper Routines

11.1. Introduction

This chapter is for RPC programmers. It documents the port mapper routines in the RPC Run-Time Library (RTL). These routines are the programming interface to RPC.

11.2. Port Mapper Routines

Port Mapper routines provide a simple callable interface to the Port Mapper. They allow you to request Port Mapper services and information about port mappings. Table 11.1, “Port Mapper Routines” summarizes the purpose of each Port Mapper routine.

Table 11.1. Port Mapper Routines

Routine	Purpose
pmap_getmaps	Returns a list of Port Mappings for the specified host.
pmap_getport	Returns the port number on which a specified service is waiting.
pmap_rmtcall	Requests the Port Mapper on a remote host to call a procedure on that host.
pmap_set	Registers a remote service with a remote port.
pmap_unset	Unregisters a service so it is no longer mapped to a port.

11.3. Port Mapper Arguments

Port Mapper routines use many of the same arguments as client routines. See Table 10.1, “Common Arguments” for a list of these arguments.

The following sections describe each Port Mapper routine in detail.

pmap_getmaps

pmap_getmaps — Returns a list of Port Mappings for the specified host.

Format

```
struct pmaplist *pmap_getmaps (struct sockaddr_in *addr);
```

Argument

addr

Address of a structure containing the internet address of the host whose Port Mapper is being called.

Description

The **pmap_getmaps** routine returns a list of current RPC server-to-Port Mappings on the host at *addr*. The list structure is defined in the PMAP_PROT.H file.

The **IP SHOW /RPC_PORTMAP** command uses this routine.

Diagnostics

If an error occurs (for example, **pmap_getmaps** cannot get a list of Port Mappings, the internet address is invalid, or the remote Port Mapper does not exist), the routine returns either NULL or the address of the list.

See Also

pmap_getport, **pmap_set**, **pmap_unset**

pmap_getport

pmap_getport — Returns the port number on which a specified service is waiting.

Format

```
u_short pmap_getport (struct sockaddr_in *addr, u_long prognum,
u_long versnum, u_long protocol);
```

Arguments

addr

Address of a structure containing the internet address of the remote host on which the server resides.

prognum, versnum, protocol

See Table 10.1, “Common Arguments” for a list of these arguments

Diagnostics

If the requested mapping does not exist or the routine fails to contact the remote Port Mapper, the routine returns either the port number or zero.

The **pmap_getport** routine uses the global variable `rpc_createerr`. `rpc_createerr` is a structure that contains the most recent service creation error. Use `rpc_createerr` if you want the service program to handle the error. The value of `rpc_createerr` is set by any RPC server creation routine that does not succeed.

The `rpc_createerr` variable is defined in the `CLNT.H` file.

See Also

pmap_getmaps, **pmap_set**, **pmap_unset**

pmap_rmtcall

pmap_rmtcall — Requests the Port Mapper on a remote host to call a procedure on that host.

Format

```
enum clnt_stat pmap_rmtcall (struct sockaddr_in *addr, u_long
prognum, u_long versnum, u_long procnum, xdrproc_t inproc, u_char
```

```
*in, xdrproc_t outproc, u_char *out, struct timeval tout, u_long
*portp);
```

Arguments

addr

Address of a structure containing the internet address of the remote host on which the server resides.

prognum, versnum, procnum, inproc, in, outproc, out

See Table 10.1, “Common Arguments” for a list of these arguments

tout

Time allowed for the results to return to the client, in seconds and microseconds.

portp

Address where **pmap_rmtcall** will write the port number of the remote service.

Description

The **pmap_rmtcall** routine allows you to get a port number and call a remote procedure in one call. The routine requests a remote Port Mapper to call a *prognum*, *versnum*, and *procnum* on the Port Mapper's host. The remote procedure call uses the UDP transport.

If **pmap_rmtcall** succeeds, it changes *portp* to contain the port number of the remote service.

After calling the **pmap_rmtcall** routine, you may call the **clnt_perrno / clnt_sperrno** routine.

Diagnostics

This routine returns diagnostic values defined in the CLNT.H file for *enum clnt_stat*.

See Also

clnt_broadcast, **clnt_perrno / clnt_sperrno**

pmap_set

pmap_set — Registers a remote service with a remote port.

Format

```
bool_t pmap_set (u_long prognum, u_long versnum, u_long protocol,
u_short port);
```

Arguments

prognum, versnum, protocol

See Table 10.1, “Common Arguments” for a list of these arguments

port

Remote port number.

Description

The **pmap_set** calls the local Port Mapper to tell it which *port* and *protocol* the*prognum*, *versnum* is using.

You are not likely to use **pmap_set**, because svc_register calls it.

Diagnostics

The **pmap_set** routine returns TRUE if it succeeds, and FALSE if it fails.

See Also

pmap_getport, **pmap_getmaps**, **pmap_unset**, **svc_register**

pmap_unset

pmap_unset — Unregisters a service so it is no longer mapped it to a port.

Format

```
bool_t pmap_unset (u_long prognum, u_long versnum);
```

Arguments

prognum, versnum

See Table 10.1, “Common Arguments” for a list of these arguments

Description

The **pmap_unset** routine calls the local Port Mapper and, for all protocols, removes the *prognum* and *versnum* from the list that maps servers to ports.

You are not likely to use **pmap_unset**, because svc_unregister calls it.

Diagnostics

The **pmap_unset** routine returns TRUE if it succeeds, FALSE if it fails.

See Also

pmap_getport, **pmap_getmaps**, **pmap_set**, **svc_unregister**

Chapter 12. RPC RTL Server Routines

12.1. Introduction

This chapter is for RPC programmers. It documents the server routines in the RPC Run-Time Library (RTL). These routines are the programming interface to RPC.

12.2. Server Routines

The server routines are called by the server program or the server stub procedures. Table 12.1, “Server Routines” lists each server routine and summarizes its purpose.

Table 12.1. Server Routines

Routine	Purpose
registerrpc	Performs creation and registration tasks for server.
svc_destroy	Macro that destroys RPC server handle.
svc_freeargs	Macro that frees memory allocated when RPC arguments were decoded.
svc_getargs	Macro that decodes RPC arguments.
svc_getreqset	Reads data for each server connection.
svc_register	Adds specified server to list of active servers, and registers service program with Port Mapper.
svc_run	Waits for RPC requests and calls **svc_getreqset** routine to dispatch to appropriate RPC service program.
svc_sendreply	Sends results of remote procedure call to client.
svc_unregister	Calls Port Mapper to unregister specified program and version for all protocols.
svcerr_auth / svcerr_decode / svcerr_noproc / svcerr_noprog / svcerr_progvers / svcerr_systemerr / svcerr_weakauth	Sends error code when server cannot authenticate client. Sends error code to client if server cannot decode arguments. Sends error code to client if server cannot implement requested procedure. Sends error code to client when requested program is not registered with Port Mapper. Sends error code to client when requested program is registered with Port Mapper, but requested version is not registered. Sends error code to client when server encounters error not handled by particular protocol. Sends error code to client when server cannot perform remote procedure call because it received insufficient (but correct) authentication parameters.

Routine	Purpose
svcfd_create	Returns address of structure containing server handle for specified TCP socket.
svctcp_create	Returns address of server handle that uses TCP transport.
svcudp_create / **svcudp_bufcreate**	Returns address of server handle that uses UDP transport. For procedures that pass messages longer than 8Kbytes. Returns address of server handle that uses UDP transport. For procedures that pass messages shorter than 8Kbytes.
svcudp_enablecache	Enables XID cache for specified UDP transport server.
xprt_register	Adds UDP or TCP server socket to list of sockets.
xprt_unregister	Removes UDP or TCP server socket from list of sockets.

The following sections describe each server routine in detail.

registerrpc

registerrpc — Performs creation and registration tasks for the server.

Format

```
#include int registerrpc (u_long prognum, u_long versnum, u_long
procnum, u_char *(*procname) (), xdrproc_t inproc, xdrproc_t
outproc);
```

Arguments

prognum, *versnum*, *procnum*, *inproc*, *outproc*

See Table 10.1, “Common Arguments” for a list of these arguments

procname

Address of the routine that implements the service procedure. The routine uses the following format:

```
u_char *procname(out);
u_char *out;
```

out is the address of the data decoded by *outproc*.

Description

The **registerrpc** routine performs the following tasks for a server:

- Creates a UDP server handle.
- Calls the **svc_register** routine to register the program with the Port Mapper.
- Adds *prognum*, *versnum*, and *procnum* to an internal list of registered procedures. When the server receives a request, it uses this list to determine which routine to call.

A server should call **registerrpc** for every procedure it implements, except for the NULL procedure.

Diagnostics

The **registerrpc** routine returns zero if it succeeds, and -1 if it fails.

See Also

svc_register

svc_destroy

svc_destroy — Macro that destroys the RPC server handle.

Format

```
void svc_destroy (SVCXPRT *xprt);
```

Argument

xprt

RPC server handle.

Description

The **svc_destroy** routine destroys *xprt* by deallocating private data structures. After this call, *xprt* is undefined.

If the server creation routine received RPC_ANYSOCK as the socket, **svc_destroy** closes the socket. Otherwise, you must close the socket.

See Also

svcfd_create, **svctcp_create**, **svcudp_create / svcudp_bufcreate**

svc_freeargs

svc_freeargs — Macro that frees the memory that was allocated when the RPC arguments were decoded.

Format

```
bool_t svc_freeargs (SVCXPRT *xprt, xdrproc_t xdr_args, char
*args_ptr);
```

Arguments

xprt, *xdr_args*, *args_ptr*

See Table 10.1, “Common Arguments” for a list of these arguments

Description

The **svc_freeargs** routine calls the **xdr_free** routine.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

svc_getargs, **xdr_free**

svc_getargs

svc_getargs — Macro that decodes the RPC arguments.

Format

```
bool_t svc_getargs (SVCXPRT *xprt, xdrproc_t xdr_args, u_char
*args_ptr);
```

Arguments

xprt, *xdr_args*, *args_ptr*

See Table 10.1, “Common Arguments” for a list of these arguments

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

svc_freeargs

svc_getreqset

svc_getreqset — Reads data for each server connection.

Format

```
#include

void svc_getreqset (int rdfds);
```

Argument

rdfds

Address of the read socket descriptor array. This array is returned by the select routine.

Description

The server calls **svc_getreqset** when it receives an RPC request. The **svc_getreqset** routine reads in data for each server connection, then calls the server program to handle the data.

The **svc_getreqset** routine does not return a value. It finishes executing after all *rdfds* sockets have been serviced.

You are unlikely to call this routine directly, because the **svc_run** routine calls it. However, there are times when you cannot call **svc_run**. For example, suppose a program services RPC requests and reads or writes to another socket at the same time. The program cannot call **svc_run**. It must call select and **svc_getreqset**.

The **svc_getreqset** routine is for servers that implement custom asynchronous event processing, do not use the **svc_run** routine.

You may use the global variable `svc_fdset` with **svc_getreqset**. The `svc_fdset` variable lists all sockets the server is using. It contains an array of structures, where each element is a socket pointer and a service handle. It uses the following format:

```
struct sockarr svc_fdset [MAXSOCK +1];
```

This is how to use `svc_fdset`: first, copy the socket handles from `svc_fdset` into a temporary array that ends with a zero. Pass the array to the **select()** routine. The **select()** routine overwrites the array and returns it. Pass this array to the **svc_getreqset** routine.

You may use `svc_fdset` when the server does not use **svc_run**.

The `svc_fdset` variable is not compatible with UNIX.

Example

```
#define MAXSOCK  10

  int readfds[ MAXSOCK+1],  /* sockets to select from */
   i, j;

  for( i = 0, j = 0; i < MAXSOCK; i++)
  if( (svc_fdset[i].sockname != 0) && (svc_fdset[i].sockname !=
1))
  readfds[j++] = svc_fdset[i].sockname;
  readfds[j] = 0;   /* list of sockets ends w/ a zero */
  switch( select( 0, readfds, 0, 0, 0))
  {
  case -1:          /* an error happened */
  case 0:           /* time out */
   break;
  default: /* 1 or more sockets ready for reading */
   errno = 0;
   ONCRPC_SVC_GET_REQSET( readfds);
   if( errno == ENETDOWN || errno == ENOTCONN)
    sys$exit( SS$_THIRDPARTY);
  }
```

See Also

svc_run

svc_register

svc_register — Adds the specified server to a list of active servers, and registers the service program with the Port Mapper.

Format

```
#include

bool_t svc_register (SVCXPRT *xprt, u_long prognum, u_long versnum,
void (*dispatch) (), u_long protocol);
```

Arguments

xprt, *prognum*, *versnum*

See Table 10.1, “Common Arguments” for a list of these arguments

dispatch

Routine that **svc_register** calls when the server receives a request for *prognum*, *versnum*. This routine determines which routine to call for each server procedure. This routine uses the following form:

```
void dispatch(request, xprt)

struct svc_req *request;

SVCXPRT *xprt;
```

The **svc_getreqset** and **svc_run** routines call *dispatch*.

protocol

Must be *IPPROTO_UDP*, *IPPROTO_TCP*, or zero. Zero indicates that you do not want to register the server with the Port Mapper.

Diagnostics

The **svc_register** routine returns TRUE if it succeeds and FALSE if it fails.

See Also

pmap_set, **svc_getreqset**, **svc_unregister**

svc_run

svc_run — Waits for RPC requests and calls the **svc_getreqset** routine to dispatch to the appropriate RPC service program.

Format

```
#include

void svc_run()
```

Arguments

None.

Description

The **svc_run** routine calls the **select()** routine to wait for RPC requests. When a request arrives, **svc_run** calls the **svc_getreqset** routine. Then **svc_run** calls **select()** again.

The **svc_run** routine never returns.

You may use the global variable `svc_fdset` with **svc_run**. See the **svc_getreqset** routine for more information on `svc_fdset`.

See Also

svc_getreqset

svc_sendreply

svc_sendreply — Sends the results of a remote procedure call to the client.

Format

```
#include

bool_t svc_sendreply (SVCXPRT *xprt, xdrproc_t outproc, caddr_t
*out);
```

Arguments

xprt, *outproc*, *out*

See Table 10.1, “Common Arguments” for a list of these arguments

Description

The routine sends the results of a remote procedure call to the client.

Diagnostics

These routines returns TRUE if they succeed and FALSE if they fail.

svc_unregister

svc_unregister — Calls the Port Mapper to unregister the specified program and version for all protocols. The program and version are removed from the list of active servers.

Format

```
#include

void svc_unregister (u_long prognum, u_long versnum);
```

Arguments

prognum, *versnum*

See Table 10.1, “Common Arguments” for a list of these arguments

See Also

pmap_unset, **svc_register**

svcerr_auth / svcerr_decode / svcerr_noproc / svcerr_noprog / svcerr_progvers / svcerr_systemerr / svcerr_weakauth

svcerr_auth / svcerr_decode / svcerr_noproc / svcerr_noprog / svcerr_progvers / svcerr_systemerr / svcerr_weakauth — Sends various error codes to the client process.

Format

```
#include

void svcerr_auth (SVCXPRT *xprt, enum auth_stat why);

void svcerr_decode (SVCXPRT *xprt);

void svcerr_noproc (SVCXPRT *xprt);

void svcerr_noprog (SVCXPRT *xprt);

void svcerr_progvers (SVCXPRT *xprt, u_long low-vers, u_long high-
vers);

void svcerr_systemerr (SVCXPRT *xprt);

void svcerr_weakauth (SVCXPRT *xprt);
```

Arguments

xprt

RPC server handle.

why

Error code defined in the AUTH.H file.

low-vers

Lowest version number in the range of versions that the server supports.

high-vers

Highest version in the range of versions that the server supports.

Description

svcerr_auth

See **svc_getreqset**. Calls **svcerr_auth** when it cannot authenticate a client. The **svcerr_auth** routine returns an error code (*why*) to the caller.

`svcerr_decode`

Sends an error code to the client if the server cannot decode the arguments.

`svcerr_noproc`

Sends an error code to the client if the server does not implement the requested procedure.

`svcerr_noprog`

Sends an error code to the client when the requested program is not registered with the Port Mapper. Generally, the Port Mapper informs the client when a server is not registered. Therefore, the server is not expected to use this routine.

`svcerr_progvers`

Sends an error code to the client when the requested program is registered with the Port Mapper, but the requested version is not registered.

`svcerr_systemerr`

Sends an error code to the client when the server encounters an error that is not handled by a particular protocol.

`svcerr_weakauth`

Sends an error code to the client when the server cannot perform a remote procedure call because it received insufficient (but correct) authentication parameters. This routine calls the **svcerr_auth / svcerr_decode / svcerr_noproc / svcerr_noprog / svcerr_progvers / svcerr_systemerr / svcerr_weakauth** routine. The value of *why* is *AUTH_TOOWEAK*, which means "access permission denied."

svcfd_create

svcfd_create — Returns the address of a structure containing a server handle for the specified TCP socket.

Format

```
#include

SVCXPRT *svcfd_create (int sock, u_long sendsize, u_long recvsize);
```

Arguments

sock

Socket number. Do not specify a file descriptor.

sendsize

Size of the send buffer. If you enter a value less than 100, then 4000 is used as the default.

recvsize

Size of the receive buffer. If you enter a value less than 100, then 4000 is used as the default.

Description

The **svcfd_create** routine returns the address of a server handle for the specified TCP socket. This handle cannot use a file. The server calls the **svcfd_create** routine after it accepts a TCP connection.

Diagnostics

This routine returns zero if it fails.

See Also

svctcp_create

svcraw_create

svcraw_create — Creates a server handle for memory-based Sun RPC for simple testing and timing.

Format

```
#include

SVCXPRT svcraw_create ();
```

Argument

None.

Description

The **svcraw_create** routine creates a toy Sun RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding client should live in the same address space.

This routine allows simulation of and acquisition of Sun RPC overheads (such as round trip times) without any kernel interference.

Diagnostics

This routine returns NULL if it fails.

See Also

clntraw_create

svctcp_create

svctcp_create — Returns the address of a server handle that uses the TCP transport.

Format

```
#include

SVCXPRT *svctcp_create (int sock, u_long sendsize, u_long recvsize);
```

Arguments

sock

Socket for this service. The **svctcp_create** routine creates a new socket if you enter *RPC_ANYSOCK*. If the socket is not bound to a TCP port, **svctcp_create** binds it to an arbitrary port.

sendsize

Size of the send buffer. If you enter a value less than 100, then 4000 bytes is used as the default.

recvsize

Size of the receive buffer. If you enter a value less than 100, then 4000 bytes is used as the default.

Diagnostics

The **svctcp_create** routine returns either the address of the server handle or zero (if it could not create the server handle).

See Also

svcfd_create, **svc_destroy**

svcudp_create / svcudp_bufcreate

svcudp_create / svcudp_bufcreate — Returns the address of a server handle that uses the UDP transport.

Format

```
#include

SVCXPRT *svcudp_create (int sock);

SVCXPRT *svcudp_bufcreate (int sock, u_long sendsize, u_long
recvsize);
```

Arguments

sock

Socket for this service. The **svcudp_create /** routine creates a new socket if you enter *RPC_ANYSOCK*. If the socket is not bound to a UDP port, the **svcudp_create /** routine binds it to an arbitrary port.

sendsize

Size of the send buffer. The minimum size is 100 bytes. The maximum size is 65468, the maximum UDP packet size. If you enter a value less than 100, then 4000 is used as the default.

recvsize

Size of the receive buffer. The minimum size is 100 bytes. The maximum size is 65000, the maximum UDP packet size. If you enter a value less than 100, then 4000 is used as the default.

Description

Use the **svc_create** routine only for procedures that pass messages shorter than 8Kbytes long. Use the **svcudp_create / svcudp_bufcreate** routine for procedures that pass messages longer than 8Kbytes.

Diagnostics

These routines return either a server handle, or zero (if they could not create the server handle).

See Also

svc_destroy, **svcudp_enablecache**

svcudp_enablecache

svcudp_enablecache — Enables the XID cache for the specified UDP transport server.

Format

```
bool_t svcudp_enablecache (SVCXPRT *xprt, u_long size);
```

Arguments

xprt

RPC server handle.

size

Number of entries permitted in the XID cache. You may estimate this number based on how active the server is, and on how long you want to retain old replies.

Description

Use the **svcudp_enablecache** routine after a UDP server handle is created. The server places all outgoing responses in the XID cache. The cache can be used to improve the performance of the server, for example, by preventing the server from recalculating the results or sending incorrect results.

You cannot disable the XID cache for UDP servers.

The Chapter 6, *RPC Fundamentals* provides more information on the XID cache.

Example

```
#define FALSE   0
```

```
#define UDP_CACHE_SIZE  10

  SVCXPRT *udp_xprt;

  udp_xprt = svcudp_create( RPC_ANYSOCK);
  if( svcudp_enablecache( udp_xprts, UDP_CACHE_SIZE) == FALSE)
  printf( "XID cache was not enabled");
  else
  printf( "XID cache was enabled");
```

Diagnostics

This routine returns TRUE if it enables the XID cache, and FALSE if the cache was previously enabled or an error occurs.

xprt_register

xprt_register — Adds a TCP or UDP server socket to a list of sockets.

Format

```
#include

void xprt_register (SVCXPRT *xprt);
```

Argument

xprt

RPC server handle.

Description

The **xprt_register** and **xprt_unregister** routines maintain a list of sockets. This list ensures that the correct server is called to process the request. The **xprt_register** routine adds the server socket to the `svc_fdset` variable, which also stores the server handle that is associated with the socket. The **svc_run** routine passes the list of sockets to the **select()** routine. The **select()** routine returns to **svc_run** a list of sockets that have outstanding requests.

You are unlikely to call this routine directly because **svc_register** calls it.

See Also

svc_register, **xprt_unregister**

xprt_unregister

xprt_unregister — Removes a TCP or UDP server socket from a list of sockets.

Format

```
#include

void xprt_unregister (SVCXPRT *xprt);
```

Argument

xprt

RPC server handle.

Description

This list of sockets ensures that the correct server is called to process the request. See the **xprt_unregister** routine for a description of how this list is maintained.

You are unlikely to call this routine directly because **svc_unregister** calls it.

See Also

svc_unregister, **xprt_register**

Chapter 13. RPC RTL XDR Routines

13.1. Introduction

This chapter is for RPC programmers. It documents the XDR routines in the RPC Run-Time Library (RTL). These routines are the programming interface to RPC.

13.2. XDR Routines

This section explains what XDR routines do and when you would call them. It also provides quick reference and detailed reference sections describing each XDR routine.

13.2.1. What XDR Routines Do

Most XDR routines share these characteristics:

- They convert data in two directions: from the host's local data format to XDR format (called encoding or marshalling), or the other way around (called decoding or unmarshalling).
- They use xdrs, a structure containing instructions for encoding, decoding, and deallocating memory.
- They return a boolean value to indicate success or failure.

Some XDR routines allocate memory while decoding an argument. To free this memory, call the **xdr_free** routine after the program is done with the decoded value.

Table 13.1, “XDR Actions” shows the order in which XDR routines perform encoding and decoding.

Table 13.1. XDR Actions

Client	Server
1. Encodes arguments 2. Decodes results 3. Frees results from memory	1. Decodes arguments 2. Encodes results 3. Frees arguments from memory

13.2.2. When to Call XDR Routines

Under most circumstances, you are not likely to call any XDR routines directly. The **clnt_call** and **svc_sendreply** routines call the XDR routines.

You would call the XDR routines directly only when you write your own routines to convert data to or from XDR format.

13.3. Quick Reference

Table 13.2, “XDR Encoding and Decoding Routines” lists the XDR routines that encode and decode data.

Table 13.2. XDR Encoding and Decoding Routines

This routine...	Encodes and decodes...
xdr_array	Variable-length array
xdr_bool	Boolean value
xdr_bytes	Bytes
xdr_char	Character
xdr_double	Double-precision floating point number
xdr_enum	Enumerated type
xdr_float	Floating point value
xdr_hyper	Quad word to an XDR hyper-integer, or the other way
xdr_int	Four-byte integer
xdr_long	Longword
xdr_opaque	Contents of a buffer (treats the data as a fixed length of bytes and does not attempt to interpret them)
xdr_pointer	Pointer to a data structure
xdr_reference	Pointer to a data structure (the address must be non-zero)
xdr_short	Two-byte unsigned integer
xdr_string	Null-terminated string
xdr_u_char	Unsigned character
xdr_u_hyper	Quad word to an XDR unsigned hyper-integer
xdr_u_int	Four-byte unsigned integer
xdr_u_long	Unsigned longword
xdr_u_short	Two-byte unsigned integer
xdr_union	Union
xdr_vector	Vector (fixed length array)
xdr_void	Nothing
xdr_wrapstring	Null-terminated string

Table 13.3, “XDR Support Routines” lists the XDR routines that perform various support functions.

Table 13.3. XDR Support Routines

This routine...	Does this...
xdr_free	Deallocates a data structure from memory
xdrmem_create	Creates a memory buffer XDR stream
xdrrec_create	Creates a record-oriented XDR stream
xdrrec_endofrecord	Marks the end of a record
xdrrec_eof	Goes to the end of the current record, then verifies whether any more data can be read
xdrrec_skiprecord	Goes to the end of the current record
xdrstdio_create	Initializes an stdio stream

Table 13.4, “Upper Layer XDR Routines” lists the upper layer XDR routines that support RPC.

Table 13.4. Upper Layer XDR Routines

This routine...	Encodes and decodes...
xdr_accepted_reply	Part of an RPC reply message after the reply is accepted
xdr_authunix_parms	UNIX-style authentication information
xdr_callhdr	Static part of an RPC request message header (encoding only)
xdr_callmsg	RPC request message
xdr_netobj	Data in the netobj structure
xdr_opaque_auth	Authentication information
xdr_pmap	Port Mapper parameters
xdr_pmaplist	List of Port Mapping data
xdr_rejected_reply	Part of an RPC reply message after the reply is rejected
xdr_replymsg	RPC reply header; it then calls the appropriate routine to convert the rest of the message

The following sections describe each XDR routine in detail.

xdr_accepted_reply

xdr_accepted_reply — Converts an RPC reply message from local format to XDR format, or the other way around.

Format

```
#include

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

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

ar

Address of the structure containing the RPC reply message.

Description

The **xdr_replymsg** routine calls the **xdr_accepted_reply** routine.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdr_replymsg

xdr_array

xdr_array — Converts a variable-length array from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_array (XDR *xdrs, u_char **addrp, u_long *sizep, u_long
maxsize, u_long elsize, xdrproc_t elproc);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

addrp

Address of the address containing the array being converted. If *addrp* is zero, then xdr_array allocates ((*sizep) *elsize) number of bytes when it decodes.

sizep

Address of the number of elements in the array.

maxsize

Maximum number of elements the array can hold.

elsize

Size of each element, in bytes.

elproc

XDR routine that handles each array element.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_authunix_parms

xdr_authunix_parms — Converts UNIX-style authentication information from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_authunix_parms (XDR *xdrs, struct authunix_parms *aupp);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

aupp

UNIX-style authentication information being converted.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_bool

xdr_bool — Converts a boolean value from local format to XDR format, or the other way around.

Format

```
#include

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

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

bp

Address of the boolean value.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_bytes

xdr_bytes — Converts bytes from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_bytes (XDR *xdrs, u_char **cpp, u_long *sizep, u_long
maxsize);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

cpp

Address of the address of the buffer containing the bytes being converted. If **cpp* is zero, **xdr_bytes** allocates *maxsize* bytes when it decodes.

sizep

Address of the actual number of bytes being converted.

maxsize

Maximum number of bytes that can be used. The server protocol determines this number.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_callhdr

xdr_callhdr — Encodes the static part of an RPC request message header.

Format

```
#include

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

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

chdr

Address of the data being converted.

Description

The **xdr_callhdr** routine converts the following fields: transaction ID, direction, RPC version, server program number, and server version. It converts the last four fields once, when the client handle is created.

The **clnttcp_create** and **clntudp_create / clntudp_bufcreate** routines call the **xdr_callhdr** routine.

Diagnostics

This routine always returns TRUE.

See Also

clnt_call, **clnttcp_create**, **clntudp_create / clntudp_bufcreate**, **xdr_callmsg**

xdr_callmsg

xdr_callmsg — Converts an RPC request message from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_callmsg (XDR *xdrs, struct rpc_msg *cmsg);
```

Arguments

`xdrs`

Address of a structure containing XDR encoding and decoding information.

`cmsg`

Address of the message being converted.

Description

The **xdr_callmsg** routine converts the following fields: transaction ID, RPC direction, RPC version, program number, version number, procedure number, client authentication.

The **pmap_rmtcall**, **svc_sendreply**, and svc_sendreply_dq routines call **xdr_callmsg**.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdr_callhdr

xdr_char

xdr_char — Converts a character from local format to XDR format, or the other way around.

Format

```
#include

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

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

cp

Address of the character being converted.

Description

This routine provides the same functionality as the **xdr_u_char** routine.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdr_u_char

xdr_double

xdr_double — Converts a double-precision floating point number between local and XDR format.

Format

```
#include

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

Arguments

xdrs

Pointer to an XDR stream handle created by one of the XDR stream handle creation routines.

dp

Pointer to the double-precision floating point number.

Description

This routine provides a filter primitive that translates between double-precision numbers and their external representations. It is actually implemented by four XDR routines:

xdr_double_D	Converts D format floating point numbers
xdr_double_G	Converts G format floating point numbers
xdr_double_T	Converts IEEE T format floating point numbers
xdr_double_X	Converts IEEE X format floating point numbers

You can reference these routines explicitly or you can use compiler settings to control which routine is used when you reference the **xdr_double** routine.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_enum

xdr_enum — Converts an enumerated type from local format to XDR format, or the other way around.

Format

```
#include

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

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

ep

Address containing the enumerated type.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_float

xdr_float — Converts a floating point value from local format to XDR format, or the other way around.

Format

```
#include

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

Arguments

xdrs

Pointer to an XDR stream handle created by one of the XDR stream handle creation routines.

fp

Pointer to a single-precision floating point number.

Description

This routine provides a filter primitive that translates between double-precision numbers and their external representations. It is actually implemented by four XDR routines:

xdr_float_F	Converts F format floating point numbers
xdr_float_S	Converts IEEE T format floating point numbers

You can reference these routines explicitly or you can use compiler settings to control which routine is used when you reference the **xdr_float** routine.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_free

xdr_free — Deallocates a data structure from memory.

Format

```
#include

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

Arguments

proc

XDR routine that describes the data structure.

objp

Address of the data structure.

Description

Call this routine after decoded data is no longer needed. Do not call it for encoded data.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_hyper

xdr_hyper — Converts a quad word to an XDR hyper-integer, or the other way around.

Format

```
bool_t xdr_hyper (XDR *xdrs, quad *ptr);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

ptr

Address of the structure containing the quad word. The quad word is stored in standard quad word format, with the low-order longword first in memory.

Description

This routine provided the same functionality as the **xdr_u_hyper** routine.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdr_u_hyper

xdr_int

xdr_int — Converts one four-byte integer from local format to XDR format, or the other way around.

Format

```
#include

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

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

ip

Address containing the integer.

Description

This routine provides the same functionality as the **xdr_u_int**, **xdr_long**, and **xdr_u_long** routines.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdr_u_int, **xdr_long**, **xdr_u_long**

xdr_long

xdr_long — Converts one longword from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_long (XDR *xdrs, u_long *lp);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

lp

Address containing the longword.

Description

This routine provides the same functionality as the **xdr_u_long**, **xdr_int**, and **xdr_u_int** routines.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdr_u_long, **xdr_int**, **xdr_u_int**

xdr_netobj

xdr_netobj — Converts data in the netobj structure from the local data format to XDR format, or the other way around.

Format

```
bool_t xdr_netobj (XDR *xdrs, netobj *ptr);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

ptr

Address of the following structure:

```
typedef struct
{
   u_long n_len;
   byte *n_bytes;
} netobj
```

This structure defines the data being converted.

Description

The netobj structure is an aggregate data structure that is opaque and contains a counted array of 1024 bytes.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_opaque

xdr_opaque — Converts the contents of a buffer from the local data format to XDR format, or the other way around. This routine treats the data as a fixed length of bytes and does not attempt to interpret them.

Format

```
#include

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

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

cp

Address of the buffer containing opaque data.

cnt

Byte length.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_opaque_auth

xdr_opaque_auth — Converts authentication information from the local data format to XDR format, or the other way around.

Format

```
#include

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

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

ap

Address of the authentication information. This data was created by the **authnone_create**, **authunix_create**, or **authunix_create_default** routine.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_pmap

xdr_pmap — Converts Port Mapper parameters from the local data format to XDR format, or the other way around.

Format

```
#include "IP$INCLUDE:PMAP_PROT.H"

bool_t xdr_pmap (XDR *xdrs, struct pmap *regs);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

regs

Address of a structure containing the program number, version number, protocol number, and port number. This is the data being converted.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_pmaplist

xdr_pmaplist — Converts a list of Port Mapping data from the local data format to XDR format, or the other way around.

Format

```
#include "TCPIP$RPC:PMAP_PROT.H"

bool_t xdr_pmaplist (XDR *xdrs, struct pmaplist **rpp);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

rpp

Address of the address of the structure containing Port Mapper data. If this routine is used to decode a Port Mapper listing, *rpp* is set to the address of the newly allocated linked list of structures.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_pointer

xdr_pointer — Converts a recursive data structure from the local data format to XDR format, or the other way around.

Format

```
#include tcpip$rpc:xdr.h

bool_t xdr_pointer (XDR *xdrs, u_char **objpp, u_long obj_size,
xdrproc_t xdr_obj);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

objpp

Address of the address containing the data being converted. May be zero.

obj_size

Size of the data structure in bytes.

xdr_obj

XDR routine that describes the object being pointed to. This routine can describe complex data structures, and these structures may contain pointers.

Description

An XDR routine for a data structure that contains pointers to other structures, such as a linked list, would call the **xdr_pointer** routine. The **xdr_pointer** routine encodes a pointer from an address into a boolean. If the boolean is TRUE, the data follows the boolean.

Example

```
bool_t xdr_pointer( xdrs, objpp, obj_size, xdr_obj)
  XDR           *xdrs;
  char          **objpp;
  longw         obj_size;
  xdrproc_t     xdr_obj;
{
        bool_t  more_data;

/*
** determine if the pointer is a valid address (0 is invalid)
*/
        if( *objpp != NULL)
```

```
          more_data = TRUE;
        else
          more_data = FALSE;
/*
** XDR the flag
** If we are decoding, then more_data is overwritten.
*/
        if( !xdr_bool( xdrs, &more_data))
          return( FALSE);
/*
** If there is no more data, set the pointer to 0 (No effect if we
** were encoding) and return TRUE
*/
        if( !more_data)
        {
          *objpp = NULL;
          return( TRUE);
        }
/*
** Otherwise, call xdr_reference.  The result is that xdr_pointer is
** the same as xdr_reference, except that xdr_pointer adds a Boolean
** to the encoded data and will properly handle NULL pointers.
*/
        return( xdr_reference( xdrs, objpp, obj_size, xdr_obj));
} /* end function xdr_pointer() */
```

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_reference

xdr_reference — This routine recursively converts a structure that is referenced by a pointer inside the structure.

Format

```
#include tcpip$rpc:xdr.h

bool_t xdr_reference (XDR *xdrs, u_char **objpp, u_long obj_size,
xdrproc_t xdr_obj);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

objpp

Address of the address of a structure containing the data being converted. If *objpp* is zero, the **xdr_reference** routine allocates the necessary storage when decoding. This argument must be non-zero when encoding.

When **xdr_reference** encodes data, it passes **objpp* to *xdr_obj*. When decoding, **xdr_reference** allocates memory if **objpp* equals zero.

obj_size

Size of the referenced structure.

xdr_obj

XDR routine that describes the object being pointed to. This routine can describe complex data structures, and these structures may contain pointers.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_rejected_reply

xdr_rejected_reply — Converts the remainder of an RPC reply message after the header indicates that the reply is rejected.

Format

```
#include tcpip$rpc:xdr.h

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

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

rr

Address of the structure containing the reply message.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_replymsg

xdr_replymsg — Converts the RPC reply header, then calls the appropriate routine to convert the rest of the message.

Format

```
#include tcpip$rpc:xdr.h

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

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

rmsg

Address of the structure containing the reply message.

Description

The **xdr_replymsg** routine calls the **xdr_rejected_reply** or **xdr_accepted_reply** routine to convert the body of the RPC reply message from the local data format to XDR format, or the other way around.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdr_accepted_reply, **xdr_rejected_reply**

xdr_short

xdr_short — Converts a two-byte integer from the local data format to XDR format, or the other way around.

Format

```
#include tcpip$rpc:xdr.h

bool_t xdr_short (XDR *xdrs, short *sp);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

sp

Address of the integer being converted.

Description

This routine provides the same functionality as **xdr_u_short**.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdr_u_short

xdr_string

xdr_string — Converts a null-terminated string from the local data format to XDR format, or the other way around.

Format

```
#include tcpip$rpc:xdr.h

bool_t xdr_string (XDR *xdrs, char **cpp, u_long maxsize);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

cpp

Address of the address of the first byte in the string.

maxsize

Maximum length of the string. The service protocol determines this value.

Description

The **xdr_string** routine is the same as the **xdr_wrapstring** routine, except **xdr_string** allows you to specify the *maxsize*.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdr_wrapstring

xdr_u_char

xdr_u_char — Converts an unsigned character from local format to XDR format, or the other way around.

Format

```
#include tcpip$rpc:xdr.h

bool_t xdr_u_char (XDR *xdrs, u_char bp);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

bp

Address of the character being converted.

Description

This routine provides the same functionality as **xdr_char**.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdr_char

xdr_u_hyper

xdr_u_hyper — Converts a quad word to an XDR unsigned hyper-integer, or the other way around.

Format

```
bool_t xdr_u_hyper (XDR *xdrs, quad *ptr);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

ptr

Address of the structure containing the quad word. The quad word is stored in standard format, with the low-order longword first in memory.

Description

This routine provides the same functionality as the **xdr_hyper** routine.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdr_hyper

xdr_u_int

xdr_u_int — Converts a four-byte unsigned integer from local format to XDR format, or the other way around.

Format

```
#include tcpip$rpc:xdr.h

bool_t xdr_u_int (XDR *xdrs, int *ip);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

ip

Address of the integer.

Description

This routine provides the same functionality as **xdr_int**, **xdr_long**, and **xdr_u_long**.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdr_int

xdr_u_long

xdr_u_long — Converts an unsigned longword from local format to XDR format, or the other way around.

Format

```
#include tcpip$rpc:xdr.h

bool_t xdr_u_long (XDR *xdrs, u_long *lp);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

lp

Address of the longword.

Description

This routine provides the same functionality as **xdr_long**, **xdr_int**, and **xdr_u_int**.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdr_long, **xdr_int**, **xdr_u_int**

xdr_u_short

xdr_u_short — Converts a two-byte unsigned integer from the local data format to XDR format, or the other way around.

Format

```
#include tcpip$rpc:xdr.h

bool_t xdr_u_short (XDR *xdrs, u_short *sp);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

sp

Address of the integer being converted.

Description

This routine provides the same functionality as **xdr_short**.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdr_short

xdr_union

xdr_union — Converts a union from the local data format to XDR format, or the other way around.

Format

```
#include tcpip$rpc:xdr.h

bool_t xdr_union (XDR *xdrs, enum_t *dscmp, u_char *unp, xdr_discrim
*choices, xdrproc_t dfault);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

dscmp

Integer from the *choices* array.

unp

Address of the union.

choices

Address of an array. This array maps integers to XDR routines.

dfault

XDR routine that is called if the *dscmp* integer is not in the *choices* array.

Description

The **xdr_union** routine searches the array choices for the value of *dscmp*. If it finds the value, it calls the corresponding XDR routine to process the remaining data. If **xdr_union** does not find the value, it calls the default routine.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_vector

xdr_vector — Converts a vector (fixed length array) from the local data format to XDR format, or the other way around.

Format

```
#include tcpip$rpc:xdr.h

bool_t xdr_vector (XDR *xdrs, u_char *basep, u_long nelem, u_long
elmsize, xdrproc_t xdr_elem);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

basep

Address of the array.

nelem

Number of elements in the array.

elmsize

Size of each element.

xdr_elem

Converts each element from the local data format to XDR format, or the other way around.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_void

xdr_void — Converts nothing.

Format

```
#include tcpip$rpc:xdr.h

bool_t xdr_void (XDR *xdrs, u_char *ptr);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

ptr

Ignored.

Description

Use this routine as a place-holder for a program that passes no data. The server and client expect an XDR routine to be called, even when there is no data to pass.

Diagnostics

This routine always returns TRUE.

xdr_wrapstring

xdr_wrapstring — Converts a null-terminated string from the local data format to XDR format, or the other way around.

Format

```
#include tcpip$rpc:xdr.h

bool_t xdr_wrapstring (XDR *xdrs, char **cpp);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

cpp

Address of the address of the first byte in the string.

Description

The **xdr_wrapstring** routine calls the **xdr_string** routine. The **xdr_wrapstring** routine hides the *maxsize* argument from the programmer. Instead, the maximum size of the string is assumed to be 232 - 1.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdr_string

xdrmem_create

xdrmem_create — Creates a memory buffer XDR stream.

Format

```
#include tcpip$rpc:xdr.h

void xdrmem_create (XDR *xdrs, u_char *addr, u_long size, enum
xdr_op op);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

addr

Address of the buffer containing the encoded data.

size

Size of the *addr* buffer.

op

Operations you will perform on the buffer. Valid values are *XDR_ENCODE*, *XDR_DECODE*, and **xdr_free**. You may change this value.

Description

The **xdrmem_create** routine initializes a structure so that other XDR routines can write to a buffer.

xdrrec_create

xdrrec_create — Creates a record-oriented XDR stream.

Format

```
#include tcpip$rpc:xdr.h

void xdrrec_create (XDR *xdrs, u_long sendsize, u_long recvsize,
u_char *tcp_handle, int (*readit)(), int (*writeit)();
```

Arguments

xdrs

Address of the structure being created. The **xdrrec_create** routine will write XDR encoding and decoding information to this structure.

sendsize

Size of the send buffer in bytes. The minimum size is 100 bytes. If you specify fewer than 100 bytes, 4000 bytes is used as the default.

recvsize

Size of the receive buffer in bytes. The minimum size is 100 bytes. If you specify fewer than 100 bytes, 4000 bytes is used as the default.

tcp_handle

Address of the client or server handle.

readit

Address of a user-written routine that reads data from the stream transport. This routine must use the following format:

```
int readit(tcp_handle, buffer, len)
u_char *tcp_handle;
u_char *buffer;
u_long len;
```

**tcp_handle* is the client or server handle

**buffer* is the buffer to fill

len is the number of bytes to read

The *readit* routine returns either the number of bytes read, or -1 if an error occurs.

writeit

Address of a user-written routine that writes data to the stream transport. This routine must use the following format:

```
int writeit(tcp_handle, buffer, len)
u_char *tcp_handle;
u_char *buffer;
u_long len;
```

`tcp_handle` is the client or server handle.

`buffer` is the address of the buffer being written.

`len` is the number of bytes to write.

The **writeit** routine returns either the number of bytes written, or -1 if an error occurs.

Description

The **xdrrec_create** routine requires one of the following:

- The TCP transport
- A stream-oriented interface (such as file I/O) not supported by VSI TCP/IP. The stream consists of data organized into records. Each record is either an RPC request or reply.

The **clnttcp_create** and **svcfd_create** routines call the **xdrrec_create** routine.

See Also

clnttcp_create, **svcfd_create**, **xdrrec_endofrecord**, **xdrrec_eof**, **xdrrec_skiprecord**

xdrrec_endofrecord

xdrrec_endofrecord — Marks the end of a record.

Format

```
#include tcpip$rpc:xdr.h

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

Arguments

`xdrs`

Address of the structure containing XDR encoding and decoding information.

`sendnow`

Indicates when the calling program will send the record to the writeit routine (see **xdrrec_create**).

If *`sendnow`* is TRUE, **xdrrec_endofrecord** sends the record now. If *`sendnow`* is FALSE, **xdrrec_endofrecord** writes the record to a buffer and sends the buffer when it runs out of buffer space.

Description

A client or server program calls the **xdrrec_endofrecord** routine when it reaches the end of a record it is writing. The program must call the **xdrrec_create** routine before calling **xdrrec_endofrecord**.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdrrec_create, **xdrrec_eof**, **xdrrec_skiprecord**

xdrrec_eof

xdrrec_eof — Goes to the end of the current record, then verifies whether any more data can be read.

Format

```
#include tcpip$rpc:xdr.h

bool_t xdrrec_eof (XDR *xdrs);
```

Argument

xdrs

Address of the structure containing XDR encoding and decoding information.

Description

The client or server program must call the **xdrrec_create** routine before calling **xdrrec_eof**.

Diagnostics

This routine returns TRUE if it reaches the end of the data stream, and FALSE if it finds more data to read.

See Also

xdrrec_create, **xdrrec_endofrecord**, **xdrrec_skiprecord**

xdrrec_skiprecord

xdrrec_skiprecord — Goes to the end of the current record.

Format

```
#include tcpip$rpc:xdr.h

bool_t xdrrec_skiprecord (XDR *xdrs);
```

Argument

xdrs

Address of the structure containing XDR encoding and decoding information.

Description

A client or server program calls the **xdrrec_skiprecord** routine before it reads data from a stream. This routine ensures that the program starts reading a record from the beginning.

The **xdrrec_skiprecord** routine is similar to the **xdrrec_eof** routine, except that **xdrrec_skiprecord** does not verify whether any more data can be read.

The client or server program must call the **xdrrec_create** routine before calling **xdrrec_skiprecord**.

Diagnostics

This routine returns TRUE if it has skipped to the start of a record. Otherwise, it returns FALSE.

See Also

xdrrec_create, **xdrrec_endofrecord**, **xdrrec_eof**

xdrstdio_create

xdrstdio_create — Initializes a stdio XDR stream.

Format

```
#include tcpip$rpc:xdr.h

void xdrstdio_create (XDR *xdrs, FILE *file, enum xdr_op op);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

file

File pointer FILE *, which is to be associated with the stream.

op

An XDR operation, one of: *XDR_ENCODE*, *XDR_DECODE*, or **xdr_free**.

Description

The **xdrstdio_create** routine initializes a studio stream for the specified file.

Appendix A. Socket Options

This appendix describes the socket options that you can set with the Sockets API setsockopt() function and the $QIO system service IO$_SETMODE and IO$_SETCHAR I/O function codes. You can query the value of these socket options using the Sockets API getstockopt() function or the $QIO system service IO$_SENSEMODE or IO$_SENSECHAR I/O function code.

The following tables list:

- Socket Options
- TCP Protocol Options
- IP Protocol Options
- IPv6 Socket Options

The following table lists the socket options that are set at the SOL_SOCKET level and their Sockets API and system service symbol names.

Table A.1. Socket Options

Sockets API Symbol	System Service Symbol	Description
SO_BROADCAST	TCPIP$C_BROADCAST	Permits the sending of broadcast messages. Takes an integer parameter and requires a system user identification code (UIC) or SYSPRV, BYPASS, or OPER privilege. Optional for a connectionless datagram.
SO_DONTROUTE	TCPIP$C_DONTROUTE	Indicates that outgoing messages should bypass the standard routing facilities. Instead, the messages are directed to the appropriate network interface according to the network portion of the destination address.
SO_ERROR	TCPIP$C_ERROR	Obtains the socket error status and clears the error on the socket.
SO_FULL_DUPLEX_CLOSE	TCPIP $C_FULL_DUPLEX_CLOSE	When set before a close operation, the receive and transmit sides of the communications are closed.
SO_KEEPALIVE	TCPIP$C_KEEPALIVE	Keeps connections active. Enables the periodic transmission of keepalive probes to the remote system. If the remote system fails to respond to the keepalive probes, the connection is broken.If the SO_KEEPALIVE

		option is enabled, the values of TCP_KEEPCNT, TCP_KEEPINTVL and TCP_KEEPIDLE affect TCP behavior on the socket.
SO_LINGER	TCPIP$C_LINGER	Lingers on a close() function if data is present. Controls the action taken when unsent messages queue on a socket and a close() function is performed. Uses a lingerstructure parameter defined in SOCKET.H to specify the state of the option and the linger interval. If SO_LINGER is specified, the system blocks the process during the close() function until it can transmit the data or until the time expires. If the option is not specified and aclose() function is issued, the system allows the process to resume as soon as possible.
SO_OOBINLINE	TCPIP$C_OOBINLINE	When this option is set, out-of-band data is placed in the normal input queue. WhenSO_OOBINLINE is set, the MSG_OOB flag to the receive functions cannot be used to read the out-of-band data. A value of 0 disables the option, and a nonzero value enables the option.
SO_RCVBUF	TCPIP$C_RCVBUF	Sets the receive buffer size, in bytes. Takes an integer parameter and requires a system UIC or SYSPRV, BYPASS, or OPER privilege.
SO_RCVTIMEO	TCPIP$C_RCVTIMEO	For Compaq use only. Sets the timeout value for a recv() operation. The argument to the two sockopt functions is a pointer to a timeval structure containing an integer value specified in seconds.
SO_REUSEADDR	TCPIP$C_REUSEADDR	Specifies that the rules used in validating addresses supplied by a bind() function should allow reuse of local addresses. A value of 0 disables the option, and

		a non-zero value enables the option. The SO_REUSEPORT option is automatically set when an application sets SO_REUSEADDR
SO_REUSEPORT	TCPIP$C_REUSEPORT	Allows more than one process to receive UDP datagrams destined for the same port. Thebind() call that binds a process to the port must be preceded by a setsockopt() call specifying this option. SO_REUSEPORT is automatically set when an application sets the SO_REUSEADDR option.
SO_SHARE	TCPIP$C_SHARE	Allows multiple processes to share the socket.
SO_SNDBUF	TCPIP$C_SNDBUF	Sets the send buffer size in bytes. Takes an integer parameter and requires a system UIC or SYSPRV, BYPASS, or OPER privilege. Optional for a connectionless datagram.
SO_SNDLOWAT	TCPIP$C_SNDLOWAT	Sets the low-water mark for a send() operation. The send low-water mark is the amount of space that must exist in the socket send buffer for select() to return writeable. Takes an integer value specified in bytes.
SO_SNDTIMEO	TCPIP$C_SNDTIMEO	For Compaq use only. Sets the timeout value for a send() operation. The argument to the two sockopt() functions is a pointer to a timeval structure containing an integer value specified in seconds.
SO_TYPE	TCPIP$C_TYPE	Obtains the socket type.
SO_USELOOPBACK	TCPIP$C_USELOOPBACK	For Compaq use only. This option applies only to sockets in the routing domain (AF_ROUTE), When you enable this option, the socket receives a copy of everything sent on the socket.

The following table lists the TCP protocol options that are set at the IPPROTO_TCP level and their Sockets API and system service symbol names.

Table A.2. TCP Protocol Options

ts API Symbol	System Service Symbol	Description
TCP_KEEPCNT	TCPIP$C_TCP_KEEPCNT	When the SO_KEEPALIVE option is enabled, TCP sends a keepalive probe to the remote system of a connection that has been idle for a period of time. If the remote system does not respond to the keepalive probe, TCP retransmits a keepalive probe for a certain number of times before a connection is considered to be broken. The TCP_KEEPCNT option specifies the maximum number of keepalive probes to be sent. The value of TCP_KEEPCNT is an integer value between 1 and *n*, where *n* is the value of the systemwide tcp_keepcnt parameter. The default value for for the systemwide parameter,tcp_keepcnt , is 8 To display the values of the systemwide parameters, enter the following command at the system prompt: `$ sysconfig -q inet` The default value for TCP_KEEPCNT is 8.
TCP_KEEPIDLE	TCPIP$C_TCP_KEEPIDLE	When the SO_KEEPALIVE option is enabled, TCP sends a keepalive probe to the remote system of a connection that has been idle for a period of time. If the remote system does not respond to the keepalive probe, TCP retransmits a keepalive probe for a certain number of times before a connection is considered to be broken. TCP_KEEPIDLE specifies the number of seconds before TCP will send the initial keepalive probe. The default value for TCP_KEEPIDLE is an integer value between 1 and *n*, where *n* is the value for the systemwide parameter tcp_keepidle . The

		default value for tcp_keepidle , specified in half-second units, is 150 (75 seconds). To display the values of the systemwide parameters, enter the following command at the system prompt: $ sysconfig -q inet The default value for TCP_KEEPIDLE is 75 seconds.
TCP_KEEPINIT	TCPIP$C_TCP_KEEPINIT	If a TCP connection cannot be established within a period of time, TCP will time out the connection attempt. The default timeout value for this initial connection establishment is 75 seconds. The TCP_KEEPINIT option specifies the number of seconds to wait before the connection attempt times out. For passive connections, the TCP_KEEPINIT option value is inherited from the listening socket. The value of TCP_KEEPINIT is an integer between 1 and *n*, where *n* is the value for the systemwide parameter tcp_keepinit . The default value of the systemwide parameter tcp_keepinit , specified in half-second units, is 150 (75 seconds). To display the values of the systemwide parameters, enter the following command at the system prompt: $ sysconfig -q inet The TCP_KEEPINIT option does not require the SO_KEEPALIVE option to be enabled.
TCP_KEEPINTVL	TCPIP$C_TCP_KEEPINTVL	When the SO_KEEPALIVE option is enabled, TCP sends a keepalive probe to the remote system on a connection that has been idle for a period of time. If the remote system does not

		respond to a keepalive probe, TCP retransmits the keepalive probe after a period of time. The default value for this retransmit interval is 75 seconds. The TCP_KEEPINTVL option specifies the number of seconds to wait before retransmitting a keepalive probe. The value of the TCP_KEEPINTVL option is an integer between 1 and*n*, where *n* is the value of the systemwide parameter tcp_keepintvl which is specified in half-second units. The default value for the systemwide parameter tcp_keepintvl is 150 (75 seconds). To display the values of the systemwide parameters, enter the following command at the system prompt: $ sysconfig -q inet
TCP_NODELAY	TCPIP$C_TCP_NODELAY	Specifies that the send() operation not be delayed to merge packets. Under most circumstances, TCP sends data when it is presented. When outstanding data has not yet been acknowledged, TCP gathers small amounts of the data into a single packet and sends it when an acknowledgment is received. This functionality can cause significant delays for some clients that do not expect replies (such as windowing systems that send a stream of events from the mouse). The TCP_NODELAY disables the Nagle algorithm, which reduces the number of small packets on a wide area network.
TCP_MAXSEG	TCPIP$C_TCP_MAXSEG	Sets the maximum transmission unit (MTU) of a TCP segment to a specified integer value from 1 to 65535. The default is 576 bytes. Can only be set

		before a listen() or connect() operation on the socket. For passive connections, the value is obtained from the listening socket. Note that TCP does not use an MTU value that is less than 32 or greater than the local network's MTU. Setting the option to zero results in the default behavior.
TCP_NODELACK	TCPIP$C_TCP_NODELACK	When specified, disables the algorithm that gathers outstanding data that has not been acknowledged and sends it in a single packet when acknowledgment is received. Takes an integer value.
The following TCP protocol options are obsolete but provided for backward compatibility:		
TCP_DROP_IDLE	TCPIP$C_TCP_DROP_IDLE	When the TCP_KEEPALIVE option is enabled, the TCP_DROP_IDLE option specifies the time interval after which a connection is dropped. The value of TCP_DROP_IDLE is an integer specified in seconds. The default value is 600 seconds. When the TCP_DROP_IDLE option is set, the value of the TCP_KEEPCNT option is calculated as the value of TCP_DROP_IDLE divided by the value of TCP_KEEPINTVL. A call to getsockopt() function specifying the TCP_DROP_IDLE option returns the result of multiplying the values of TCP_KEEPCNT and TCP_KEEPINTVL.
TCP_PROBE_IDLE	TCPIP$C_TCP_PROBE_IDLE	When the TCP_KEEPALIVE option is enabled, the TCP_PROBE_IDLE option specifies the time interval between the keepalive probes and for the connections establishing the timeout. The default value for TCP_PROBE_IDLE is

		75 seconds. The value of TCP_PROBE_IDLE is an integer specified in seconds. When this option is set, TCP_KEEPINTVL, TCP_KEEPIDLE and TCP_KEEPINIT are set to the value specified for TCP_PROBE_IDLE. A call to the getsockopt() function specifying the TCP_PROBE_IDLE option returns the value of TCP_KEEPINTVL.

The following table lists options that are set at the IPPROTO_IP level and their Sockets API and system service symbol names.

Table A.3. Protocol Options

Sockets API Symbol	System Service Symbol	Description
IP_ADD_MEMBERSHIP	TCPIP $C_IP_ADD_MEMBERSHIP	Adds the host to the membership of a multicast group. A host must become a member of a multicast group before it can receive datagrams sent to the group. Membership is associated with a single interface; programs running on multihomed hosts may need to join the same group on more than one interface. Up to IP_MAX_MEMBERSHIPS (currently 20) memberships may be added on a single socket.
IP_DROP_MEMBERSHIP	TCPIP $C_IP_DROP_MEMBERSHIP	Removes the host from the membership of a multicast group.
IP_HDRINCL	TCPIP$C_IP_HDRINCL	If specified for a raw IP socket, you must build the IP header for all datagrams sent on the raw socket.
IP_MULTICAST_IF	TCPIP$C_IP_MULTICAST_IF	Specifies the interface for outgoing multicast datagrams sent on this socket. The interface is specified as an in_addr structure.

<table>
<tr><td>IP_MULTICAST_LOOP</td><td>TCPIP $C_IP_MULTICAST_LOOP</td><td>Disables loopback of local delivery.If a multicast datagram is sent to a group which the sending host is a member, a copy of the datagram is looped back by the IP layer for local delivery (the default). To disable the loopback delivery, specify a value of 0.</td></tr>
<tr><td>IP_MULTICAST_TTL</td><td>TCPIP $C_IP_MULTICAST_TTL</td><td>Specifies the time-to-live (TTL) value for outgoing multicast datagrams.Takes an integer value between 0 and 255:
<table>
<tr><td>Value</td><td>Action</td></tr>
<tr><td>0</td><td>Restricts distribution to applications running on the local host.</td></tr>
<tr><td>1</td><td>Forwards the multicast datagram to hosts on the local subnet.</td></tr>
<tr><td>2 - 255</td><td>With a multicast router attached to the sending host's network, forwards multicast datagrams beyond the local subnet.Multicast routers forward the datagram to known networks that have hosts belonging to the specified multicast group. The TTL value is decremented by each multicast router in the path. When the</td></tr>
</table></td></tr>
</table>

		TTL value is decremented to zero, the datagram is no longer forwarded.
IP_OPTIONS	TCPIP$C_IP_OPTIONS	Provides IP options to be transmitted in the IP header of each outgoing packet.
IP_RECVDSTADDR	TCPIP$C_IP_RECVDSTADDR	Enables a SOCK_DGRAM socket to receive the destination IP address for a UDP datagram.
IP_RECVOPTS	TCPIP$C_IP_RECVOPTS	Enables a SOCK_DGRAM socket to receive IP options.
IP_TTL	TCPIP$C_IP_TTL	Time to live (TTL) for a datagram.
IP_TOS	TCPIP$C_IP_TOS	Type of service (1-byte value).

The following table describes the socket options supporting IPv6. The IPv6 socket options do not have system service symbols.

Table A.4. IPv6 Socket Options

IPV6_RECVPKTINFO	Source and destination IPv6 address, and sending and receiving interface.
IPV6_RECVHOPLIMIT	Hop limit.
IPV6_RECVRTHDR	Routing header.
IPV6_RECVHOPOPTS	Hop-by-hop options.
IPV6_RECVDSTOPTS	Destination options.
IPV6_CHECKSUM	For raw IPv6 sockets other than ICMPv6 raw sockets, causes the kernel to compute and store checksum for output and to verify the received checksum on input. Discards the packet if the checksum is in error.
IPV6_ICMP6_FILTER	Fetches and stores the filter associated with the ICMPv6 raw socket using the getsockopt( ) function and setsockopt( ) functions.
IPV6_UNICAST_HOPS	Sets the hop limit for all subsequent unicast packets sent on a socket. You can also use this option with the getsockopt( ) function to determine the current hop limit for a socket.
IPV6_MULTICAST_	IF Sets the interface to use for outgoing multicast packets.
IPV6_MULTICAST_HOPS	Sets the hop limit for outgoing multicast packets.
IPV6_MULTICAST_LOOP	Controls whether to deliver outgoing multicast packets back to the local application.

IPV6_JOIN_GROUP	Joins a multicast group on the specified interface
IPV6_LEAVE_GROUP	Leaves a multicast group on the specified interface.

Appendix B. Trademark and Copyright Notifications

This appendix contains a complete listing of trademarks and copyright notification contained in this manual.

The material in this document is for informational purposes only and is subject to change without notice. It should not be construed as a commitment by VMS Software, inc. VMS Software, inc. assumes no responsibility for any errors that may appear in this document.

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

The following third-party software may be included with your product and will be subject to the software license agreement.

Network Time Protocol (NTP). Copyright © 1992-2004 by David L. Mills. The University of Delaware makes no representations about the suitability of this software for any purpose.

Point-to-Point Protocol. Copyright © 1989 by Carnegie-Mellon University. All rights reserved. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission. Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed by Carnegie Mellon University. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED ``AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.

RES_RANDOM.C. Copyright © 1997 by Niels Provos <provos@physnet.uni-hamburg.de> All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgement: This product includes software developed by Niels Provos.

4. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

Copyright © 1990 by John Robert LoVerso. All rights reserved. Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed by John Robert LoVerso.

Kerberos. Copyright © 1989, DES.C and PCBC_ENCRYPT.C Copyright © 1985, 1986, 1987, 1988 by Massachusetts Institute of Technology. Export of this software from the United States of America is assumed to require a specific license from the United States Government. It is the responsibility of any person or organization contemplating export to obtain such a license before exporting. WITHIN THAT CONSTRAINT, permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T. makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

DNSSIGNER (from BIND distribution) Portions Copyright (c) 1995-1998 by Trusted Information Systems, Inc.

Appendix E. Trademark and Copyright Notifications

E-160

Portions Copyright (c) 1998-1999 Network Associates, Inc.

Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies. THE SOFTWARE IS PROVIDED "AS IS" AND TRUSTED INFORMATION SYSTEMS DISCLAIMS

ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES

OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL TRUSTED INFORMATION SYSTEMS BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

ERRWARN.C. Copyright © 1995 by RadioMail Corporation. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of RadioMail Corporation, the Internet Software Consortium nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY RADIOMAIL CORPORATION, THE INTERNET SOFTWARE CONSORTIUM AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL RADIOMAIL CORPORATION OR CONTRIBUTORS

BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This software was written for RadioMail Corporation by Ted Lemon under a contract with Vixie Enterprises. Further modifications have been made for the Internet Software Consortium under a contract with Vixie Laboratories.

IMAP4R1.C, MISC.C, RFC822.C, SMTP.C Original version Copyright © 1988 by The Leland Stanford Junior University

ACCPORNAM technology Copyright (c) 1999 by Brian Schenkenberger - TMESIS SOFTWARE

NS_PARSER.C Copyright © 1984, 1989, 1990 by Bob Corbett and Richard Stallman

This program is free software. You can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 1, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139 USA

IF_ACP.C Copyright © 1985 and IF_DDA.C Copyright © 1986 by Advanced Computer Communications

IF_PPP.C Copyright © 1993 by Drew D. Perkins

ASCII_ADDR.C Copyright © 1994 Bell Communications Research, Inc. (Bellcore)

DEBUG.C Copyright © 1998 by Lou Bergandi. All Rights Reserved.

NTP_FILEGEN.C Copyright © 1992 by Rainer Pruy Friedrich-Alexander Universitaet Erlangen-Nuernberg

RANNY.C Copyright © 1988 by Rayan S. Zachariassen. All Rights Reserved.

MD5.C Copyright © 1990 by RSA Data Security, Inc. All Rights Reserved.

Portions Copyright © 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989 by SRI International

Portions Copyright © 1984, 1989 by Free Software Foundation

Portions Copyright © 1993, 1994, 1995, 1996, 1997, 1998 by the University of Washington. Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notices appear in all copies and that both the above copyright notices and this permission notice appear in supporting documentation, and that the name of the University of Washington or The Leland Stanford Junior University not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. This software is made available "as is", and THE UNIVERSITY OF WASHINGTON AND THE LELAND STANFORD JUNIOR UNIVERSITY DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, WITH REGARD TO THIS

Appendix E. Trademark and Copyright Notifications

E-161

SOFTWARE, INCLUDING WITHOUT LIMITATION ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, AND IN NO EVENT SHALL THE UNIVERSITY OF WASHINGTON OR THE LELAND STANFORD JUNIOR UNIVERSITY BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, TORT (INCLUDING NEGLIGENCE) OR STRICT LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Portions Copyright © 1980, 1982, 1985, 1986, 1988, 1989, 1990, 1993 by The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

This product includes software developed by the University of California, Berkeley and its contributors.

4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Portions Copyright © 1993 by Hewlett-Packard Corporation.

Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies, and that the name of Hewlett-Packard Corporation not be used in advertising or publicity pertaining to distribution of the document or software without specific, written prior permission. THE SOFTWARE IS PROVIDED "AS IS" AND HEWLETT-PACKARD CORP. DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL

IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL HEWLETT-PACKARD CORPORATION BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Portions Copyright © 1995 by International Business Machines, Inc.

International Business Machines, Inc. (hereinafter called IBM) grants permission under its copyrights to use, copy, modify, and distribute this Software with or without fee, provided that the above copyright notice and all paragraphs of this notice appear in all copies, and that the name of IBM not be used in connection with the marketing of any product incorporating the Software or modifications thereof, without specific, written prior

permission. To the extent it has a right to do so, IBM grants an immunity from suit under its patents, if any, for the use, sale or manufacture of products to the extent that such products are used for performing Domain Name System dynamic updates in TCP/IP networks by means of the Software. No immunity is granted for any product per se or for any other function of any product. THE SOFTWARE IS PROVIDED "AS IS", AND IBM DISCLAIMS ALL WARRANTIES, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE, EVEN IF IBM IS APPRISED OF THE POSSIBILITY OF SUCH DAMAGES.

Portions Copyright © 1995, 1996, 1997, 1998, 1999, 2000 by Internet Software Consortium. All Rights Reserved. Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies. THE SOFTWARE IS PROVIDED "AS IS" AND INTERNET SOFTWARE CONSORTIUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INTERNET SOFTWARE CONSORTIUM BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Copyright (c) 1996-2000 Internet Software Consortium.

Use is subject to license terms which appear in the file named ISC-LICENSE that should have accompanied this file when you received it. If a file named ISC-LICENSE did not accompany this file, or you are not sure the one you have is correct, you may obtain an applicable copy of the license at: http://www.isc.org/isc-license-1.0.html.

This file is part of the ISC DHCP distribution. The documentation associated with this file is listed in the file

Appendix E. Trademark and Copyright Notifications

E-162

DOCUMENTATION, included in the top-level directory of this release. Support and other services are available for ISC products - see http://www.isc.org for more information.

ISC LICENSE, Version 1.0

1. This license covers any file containing a statement following its copyright message indicating that it is covered by this license. It also covers any text or binary file, executable, electronic or printed image that is derived from a file that is covered by this license, or is a modified version of a file covered by this license, whether such works exist now or in the future. Hereafter, such works will be referred to as "works covered by this license," or "covered works."

2. Each source file covered by this license contains a sequence of text starting with the copyright message and ending with "Support and other services are available for ISC products - see http://www.isc.org for more information." This will hereafter be referred to as the file's Bootstrap License.

3. If you take significant portions of any source file covered by this license and include those portions in some other file, then you must also copy the Bootstrap License into that other file, and that file becomes a covered file. You may make a good-faith judgement as to where in this file the bootstrap license should appear.

4. The acronym "ISC", when used in this license or generally in the context of works covered by this license, is an abbreviation for the words "Internet Software Consortium."

5. A distribution, as referred to hereafter, is any file, collection of printed text, CD ROM, boxed set, or other collection, physical or electronic, which can be distributed as a single object and which contains one or more works covered by this license.

6. You may make distributions containing covered files and provide copies of such distributions to whomever you choose, with or without charge, as long as you obey the other terms of this license. Except as stated in (9), you may include as many or as few covered files as you choose in such distributions.

7. When making copies of covered works to distribute to others, you must not remove or alter the Bootstrap License. You may not place your own copyright message, license, or similar statements in the file prior to the original copyright message or anywhere within the Bootstrap License. Object files and executable files are exempt from the restrictions specified in this clause.

8. If the version of a covered source file as you received it, when compiled, would normally produce executable code that would print a copyright message followed by a message referring to an ISC web page or other ISC documentation, you may not modify the file in such a way that, when compiled, it no longer produces executable code to print such a message.

9. Any source file covered by this license will specify within the Bootstrap License the name of the ISC distribution from which it came, as well as a list of associated documentation files. The associated documentation for a binary file is the same as the associated documentation for the source file or files from which it was derived. Associated documentation files contain human-readable documentation which the ISC intends to accompany any distribution.

If you produce a distribution, then for every covered file in that distribution, you must include all of the associated documentation files for that file. You need only include one copy of each such documentation file in such distributions.

Absence of required documentation files from a distribution you receive or absence of the list of documentation files from a source file covered by this license does not excuse you from this requirement. If the distribution you receive does not contain these files, you must obtain them from the ISC and include them in any redistribution of any work covered by this license. For information on how to obtain required documentation not included with your distribution, see: http://www.isc.org/getting-documentation.html.

If the list of documentation files was removed from your copy of a covered work, you must obtain such a list from the ISC. The web page at http://www.isc.org/getting-documentation.html contains pointers to lists of files for each ISC distribution covered by this license.

It is permissible in a source or binary distribution containing covered works to include reformatted versions of the documentation files. It is also permissible to add to or modify the documentation files, as long as the formatting is similar in legibility, readability, font, and font size to other documentation in the derived product, as long as any sections labeled CONTRIBUTIONS in these files are unchanged except with respect to formatting, as long as the order in which the CONTRIBUTIONS section appears in these files is not changed, and as long as the manual page which describes how to contribute to the Internet Software Consortium (hereafter referred to as the Contributions Manual Page) is unchanged except with respect to formatting.

Documentation that has been translated into another natural language may be included in place of or in addition to the required documentation, so long as the CONTRIBUTIONS section and the Contributions Manual Page are either left in their original language or translated into the new language with such care and diligence as is required to preserve the original meaning.

10. You must include this license with any distribution that you make, in such a way that it is clearly associated with such covered works as are present in that distribution. In any electronic distribution, the license must be in a file called "ISC-LICENSE".

If you make a distribution that contains works from more than one ISC distribution, you may either include a copy of the ISC-LICENSE file that accompanied each such ISC distribution in such a way that works covered by each license are all clearly grouped with that license, or you may include the single copy of the ISC-LICENSE that has the highest version number of all the ISC-LICENSE files included with such distributions, in which case all covered works will be covered by that single license file. The version number of a license appears at the top of the file containing the text of that license, or if in printed form, at the top of the first page of that license.

Appendix E. Trademark and Copyright Notifications

E-163

11. If the list of associated documentation is in a separated file, you must include that file with any distribution you make, in such a way that the relationship between that file and the files that refer to it is clear. It is not permissible to merge such files in the event that you make a distribution including files from more than one ISC distribution, unless all the Bootstrap Licenses refer to files for their lists of associated documentation, and those references all list the same filename.

12. If a distribution that includes covered works includes a mechanism for automatically installing covered works, following that installation process must not cause the person following that process to violate this license, knowingly or unknowingly. In the event that the producer of a distribution containing covered files accidentally or wilfully violates this clause, persons other than the producer of such a distribution shall not be held liable for such violations, but are not otherwise excused from any requirement of this license.

13. COVERED WORKS ARE PROVIDED "AS IS". ISC DISCLAIMS ALL WARRANTIES WITH REGARD TO COVERED WORKS INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

14. IN NO EVENT SHALL ISC BE LIABLE FOR ANY SPECIAL, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,

NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OF COVERED WORKS.

Use of covered works under different terms is prohibited unless you have first obtained a license from ISC granting use pursuant to different terms. Such terms may be negotiated by contacting ISC as follows:

Internet Software Consortium

950 Charter Street

Redwood City, CA 94063

Tel: 1-888-868-1001 (toll free in U.S.)

Tel: 1-650-779-7091

Fax: 1-650-779-7055

Email: info@isc.org

Email: licensing@isc.org

DNSSAFE LICENSE TERMS

This BIND software includes the DNSsafe software from RSA Data Security, Inc., which is copyrighted software that can only be distributed under the terms of this license agreement.

The DNSsafe software cannot be used or distributed separately from the BIND software. You only have the right to use it or distribute it as a bundled, integrated product.

The DNSsafe software can ONLY be used to provide authentication for resource records in the Domain Name System, as specified in RFC 2065 and successors. You cannot modify the BIND software to use the

DNSsafe software for other purposes, or to make its cryptographic functions available to end-users for other uses.

If you modify the DNSsafe software itself, you cannot modify its documented API, and you must grant RSA Data Security the right to use, modify, and distribute your modifications, including the right to use

any patents or other intellectual property that your modifications depend upon.

You must not remove, alter, or destroy any of RSA's copyright notices or license information. When distributing the software to the Federal Government, it must be licensed to them as "commercial computer software" protected under 48 CFR 12.212 of the FAR, or 48 CFR 227.7202.1 of the DFARS.

You must not violate United States export control laws by distributing the DNSsafe software or information about it, when such distribution is prohibited by law.

THE DNSSAFE SOFTWARE IS PROVIDED "AS IS" WITHOUT ANY WARRANTY WHATSOEVER. RSA HAS NO OBLIGATION TO SUPPORT, CORRECT, UPDATE OR MAINTAIN THE RSA SOFTWARE. RSA DISCLAIMS ALL WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO ANY MATTER WHATSOEVER, INCLUDING ALL

IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF THIRD PARTY RIGHTS.

If you desire to use DNSsafe in ways that these terms do not permit, please contact:

RSA Data Security, Inc.

100 Marine Parkway

Redwood City, California 94065, USA