MultiNet 5.6 Programmer's Reference

November 2020

This guide provides information to configure and manage MultiNet for the experienced system manager. Before using this guide, install and start MultiNet as described in the *MultiNet Installation and Administrator's Guide*.

Operating System/Version: OpenVMS VAX V5.5-2 or later

OpenVMS Alpha V6.2 or later

OpenVMS Itanium V8.2 or later

Software Version: MultiNet 5.6

Process Software
Framingham, Massachusetts
USA

Preface

Purpose of this Guide

This guide describes the programming interfaces provided with the MultiNet software: A socket library based on the UNIX 4.3BSD system calls, and an OpenVMS $QIO interface. All socket functions documented in this guide are available in the shareable image `MULTINET:MULTINET_SOCKET_LIBRARY.EXE`, included in the standard MultiNet distribution. The include files and example programs are part of the optional *MultiNet Programmers' Kit*, and should be installed as described in the *MultiNet Installation and Administrator's Guide* before using the programming interface.

> **Note:** If you are writing socket programs in C, Process Software recommends that you use the HP C include files for the socket definitions. Your program will then use the TCP/IP Services for VMS-emulation interface in TCPware and MultiNet. The MultiNet header files have been updated to work with more current versions of HP C. The MultiNet files should be used only if you are planning to use the MultiNet `INETDRIVER` API explicitly.

Document Structure

Read this guide to perform the following tasks:

- Chapter 1, *IP Programming Tutorial*, to write clients and servers that access the network.
- Chapter 2, *Socket Library Functions*, to view detailed information about socket library functions.
- Chapter 3, *$QIO Interface*, to view detailed information about SYS$QIO calls that you can use to access the network.
- Chapter 4, *SNMP Extensible Agent API Routines*.
- Chapter 5, *RPC Fundamentals*, explains RPC.

- Chapter 6, *Building Distributed Applications with RPC*, explains what components a distributed application contains, how to use RPC to develop a distributed application, step-by-step, and how to get RPC information.
- Chapter 7, *RPCGEN Compiler*, explains the RPC compiler.
- Chapter 8, *RPC RTL Management Routines*.
- Chapter 9, *RPC RTL Client Routines*.
- Chapter 10, *RPC RTL Port Mapper Routines*.
- Chapter 11, *RPC RTL Server Routines*.
- Chapter 12, *RPC RTL XDR Routines*.

Conventions Used

Examples in this guide use the following conventions:

Convention	Meaning
host	Any computer system on the network. The local host is your computer. A remote host is any other computer.
`monospaced type`	System output or user input. User input is in **`reversed bold`** `type`. Example: `Is this configuration correct?` **`YES`** Monospaced type also indicates user input where the case of the entry should be preserved.
italic type	Variable value in commands and examples. For example, *username* indicates that you must substitute your actual username. Italic text also identifies documentation references.
[*directory*]	Directory name in an OpenVMS file specification. Include the brackets in the specification.
[*optional-text*]	(Italicized text and square brackets) Enclosed information is optional. Do not include the brackets when entering the information. Example: `START/IP` *`line address`* [*`info`*] This command indicates that the *`info`* parameter is optional.

{value | value}	Denotes that you should use only one of the given values. Do not include the braces or vertical bars when entering the value.
Note	Information that follows is particularly noteworthy.
Caution	Information that follows is critical in preventing a system interruption or security breach.
`key`	Press the specified key on your keyboard.
`Ctrl+key`	Press the control key and the other specified key simultaneously.
`Return`	Press the Return or Enter key on your keyboard.

Further Reading

The following references contain additional information about programming under TCP/IP. They may be useful in learning more about socket programming. Additional titles of recommended books can be displayed using this command:

```
$ HELP MULTINET BOOKS
```


Comer, Douglas. *Internetworking with TCP/IP: Principles, Protocols, and Architecture*, Englewood Cliffs, NJ: Prentice-Hall, 1988.

Curry, Donald A. *Using C on the UNIX System*, O'Reilly and Associates.

Harspool, R. Nigel. *C Programming in the Berkeley Unix Environment*, Toronto, Canada: Prentice-Hall, 1986.

Kochan, Stephen G. and Patrick K. Wood, editors. *UNIX Networking*, Indianapolis, IN: Hatden Books, 1989.

Leffler, Samuel J., Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*, Reading, MA: Addison-Wesley, 1989.

UNIX Programming Manuals, U. C. Berkeley.

1. MultiNet Programming Tutorial

This chapter contains short tutorials on various aspects of application programming using MultiNet.

Once you have installed the MultiNet Programmers' Kit, you will find a number of example programs in the appendices in the directory `MULTINET_ROOT:[MULTINET.EXAMPLES]`. The following tutorials, together with the example programs, are designed to get you started as an application programmer using MultiNet.

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. See the htons() and htonl() functions in Chapter 3 for further information about network byte ordering.

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 "multinet_root:[multinet.include.sys]types.h"
#include "multinet_root:[multinet.include.sys]socket.h"
#include "multinet_root:[multinet.include]netdb.h"
#include "multinet_root:[multinet.include.netinet]in.h"

main(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 */
}
```

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.

Refer to the sample program `TCPECHOCLIENT.C` in the *MultiNet Programmers' Kit* examples directory. This program sends data to a server and displays what the server sends back.

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 MultiNet 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 MultiNet_Server process, you must assign a channel to `SYS$INPUT` before calling any of the VAX C I/O routines.

Refer to the sample program `TCPECHOSERVER-STANDALONE.C` in the *MultiNet Programmers' Kit* examples directory for an example of a server program that echoes data sent to it.

Another way to write a TCP server is to let the MULTINET_SERVER process do the work for you. The MULTINET_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. Refer to the sample program `TCPECHOSERVER.C` in Appendix B and in the *MultiNet Programmers' Kit* examples directory for an example of how this is done.

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 MultiNet 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 MultiNet_Server process, you must assign a channel to `SYS$INPUT` before calling any of the VAX C I/O routines.

Another way to write a UDP server is to let the `MULTINET_SERVER` process handle the work. The `MULTINET_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.

Refer to the sample programs in the *MultiNet Programmers' Kit* examples directory for examples of UDP clients and servers.

BSD-Specific Tips

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

BSD Sockets Porting Note

When porting a program written for BSD sockets to MultiNet, observe the following guidelines:

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

BSD 4.4 TCP/IP Future Compatibility Considerations

MultiNet 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];
};
```

MultiNet 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. This provides compatibility with prior versions of MultiNet.

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 *HP 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.

TCP/IP Services (UCX) Compatibility

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

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

2. Socket Library Functions

This chapter describes the purpose and format of each MultiNet socket library function.

The socket functions described in this chapter are available in the shareable image `MULTINET:MULTINET_SOCKET_LIBRARY.EXE`, included in the standard MultiNet distribution. They include files and example programs are part of the optional MultiNet Programmers' Kit, and should be installed as described in the *MultiNet Installation and Administrator's Guide* before you use the programming interface.

In addition to supporting the MultiNet socket library, applications developed for the OpenVMS TCPIP Services (UCX) software using the VAX C socket library (`UCX$IPC.OLB`) will run over MultiNet, using an emulation of `UCX$IPC_SHR.EXE`.

> **Note:** To avoid potential conflicts between MultiNet socket library definitions and C compiler definitions, include a reference to the file `MULTINET_ROOT:[MULTINET.INCLUDE.SYS]TYPES.H` before any other header file references.

Debugging and Tracing

MultiNet 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 MultiNet 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 `MULTINET_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 1 signifies 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:MULTINET_SOCKET_`*`process_name`*`.LOG`. Control, read and write operations are logged when logging is done to a file.

AST Reentrancy

The MultiNet 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 MultiNet 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 MultiNet device driver.

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

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.

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 `MULTINET_ROOT:[MULTINET.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

FORMAT

```
short New_VMS_Channel = accept(short VMS_Channel, struct sockaddr
*Address, unsigned int AddrLen);
```

ARGUMENTS

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

`Address`
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`
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()

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 the OS vendor as part of the VAX C run-time library. It is provided here for compatibility with the 4.3BSD library.

FORMAT

```
int Status = bcmp(char *String1, char *String2, unsigned int Length);
```

ARGUMENTS

`String1, String2`
Pointers to the two buffers to be compared.

`Length`
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()

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 Hewlett-Packard as part of the VAX C run-time library. It is provided here for compatibility with the 4.3BSD library.

FORMAT

```
(void) bcopy(char *String1, char *String2, unsigned int Length);
```

ARGUMENTS

String1
The source buffer for the copy operation.

String2
The destination buffer for the copy operation.

Length
The number of bytes to be copied.

---

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 `MULTINET_ROOT:[MULTINET.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

FORMAT

```
int Status = bind(short VMS_Channel, struct sockaddr *Name, unsigned
int NameLen);
```

ARGUMENTS

`VMS_Channel`
A channel to the socket.

`Name`
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`
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()

Fills memory with zeros.

`bzero()` is part of the 4.3BSD run-time library, but is not provided as part of the VAX C run-time library. It is provided here for compatibility with the 4.3BSD library.

FORMAT

```
(void) bzero(char *String, unsigned int Length);
```

ARGUMENTS

`String`
The address of the buffer to receive the zeros.

`Length`
The number of bytes to be zeroed.

---

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 MULTINET_ROOT:[MULTINET.INCLUDE.NETINET]IN.H is used and the program is compiled with USE_BSD44_ENTRIES defined.

FORMAT

```
int Status = connect(short VMS_Channel, struct sockaddr *Name,
unsigned int NameLen);
```

ARGUMENTS

VMS_Channel
A channel to the socket.

Name
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
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.

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 MultiNet 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()

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()

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()

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 MultiNet software.

FORMAT

```
(void) endprotoent();
```

---

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 MultiNet software.

FORMAT

```
(void) endservent();
```

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()

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.

Note: The MultiNet 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 `MULTINET_ROOT:[MULTINET.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

FORMAT

```
(struct hostent *) gethostbyaddr(char *Addr, unsigned int Length,
unsigned int Family);
(struct hostent *) _gethostbyaddr(char *Addr, unsigned int Length,
unsigned int Family);
```

ARGUMENTS

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

`Family`
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 `gethostbyname()` 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 `multinet_root:[multinet.include]netdb.h`.

getaddrinfo()

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

FORMAT

```
int getaddrinfo(char *hostname, char *servname, struct addrinfo
*hints, **res)
```

ARGUMENTS

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

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

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

res
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()

Returns hostname and/or servicename information from a `sockaddr` structure. This call can handle both IPv6 and IPv4 requests.

FORMAT

```
int getnameinfo(struct sockaddr *sa, size_t salen, char *host, size_t hostlen, char *serv, size_t servlen, int flags)
```

ARGUMENTS

sa
A pointer to a `sockaddr` to obtain information on.

salen
The length of the `sockaddr` structure.

host
Storage area for a hostname to be returned.

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

serv
Storage area for a service name to be returned.

servlen
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 interpreted with `gaistrerror()`.

---

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.

Note: The MultiNet 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 `MULTINET_ROOT:[MULTINET.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

FORMAT

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

ARGUMENTS

Name
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 `multinet_root:[multinet.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()

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.

Note: The MultiNet 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 `MULTINET_ROOT:[MULTINET.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

FORMAT

```
(struct hostent *) gethostbysockaddr(struct sockaddr *Addr, unsigned
int Length);
```

ARGUMENTS

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

`Length`
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 `gethostbyname()` 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 `multinet_root:[multinet.include]netdb.h`.

gethostname()

Returns the Internet name of the host it is executed on. This name comes from the logical name `MULTINET_HOST_NAME` and can be set using the `SET HOST-NAME` command in the MultiNet Network Configuration utility (NET-CONFIG).

FORMAT

```
Int gethostname(char *String, unsigned int Length);
```

ARGUMENTS

`String`
A pointer to a buffer to receive the host name.

`Length`
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()

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(unsigned int Net, unsigned int
Protocol);
(struct netent *) _getnetbyaddr(unsigned int Net, unsigned int
Protocol);
```

ARGUMENTS

`Net`
The network number to look up.

`Protocol`
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 `getnetbyname()` 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 `multinet_root:[multinet.include]netdb.h`.

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

ARGUMENTS

Name

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 `multinet_root:[multinet.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()

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 `MULTINET_ROOT:[MULTINET.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

FORMAT

```
int getpeername(short VMS_Channel, struct sockaddr *Address, unsigned
int *AddrLen);
```

ARGUMENTS

`VMS_Channel`
A channel to the socket.

`Address`
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`
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()

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

FORMAT

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

ARGUMENTS

Name

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()

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

FORMAT

```
(struct protoent *) getprotobynumber(unsigned int Number);
```

ARGUMENTS

`Number`
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()

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()

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 *MultiNet Installation and Administrator's Guide* for more information about editing and compiling the host table files.

FORMAT

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

ARGUMENTS

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

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

RETURNS

If **getservbynam**`e()` 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()

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 *MultiNet Installation and Administrator's Guide* for more information about editing and compiling the host table files.

FORMAT

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

ARGUMENTS

`Number`
The numeric value of the service port.

`Protocol`
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 `getservbyname()` for the format of the `servent` structure.) If this function fails, a value of 0 is returned.

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 `getservbyname()` for the format of the `servent` structure.) If this function fails, a value of 0 is returned.

---

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 `MULTINET_ROOT:[MULTINET.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

FORMAT

```
int getsockname(short  VMS_Channel, struct sockaddr *Address, unsigned
int *AddrLen);
```

ARGUMENTS

`VMS_Channel`
A channel to the socket.

`Address`
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`
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()

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 `multinet_root:[multinet.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 *socket options*.

FORMAT

```
int getsockopt(short VMS_Channel, unsigned int Level, unsigned int
OptName, unsigned int OptVal, char *OptLen);
```

ARGUMENTS

VMS_Channel
A channel to the socket.

Level
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
The option to be manipulated.

OptVal
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

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()

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 Hewlett-Packard as part of the VAX C run-time library. It is provided here for compatibility with the 4.3BSD library.

FORMAT

```
int gettimeofday(timeval *TimeVal);
```

ARGUMENTS

`TimeVal`
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()

Examines the user-specific host alias table (if the user has set one by defining the `MULTINET_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(char *Name);
```

ARGUMENTS

Name

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()

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(unsigned long Val);
```

ARGUMENTS

`Val`
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()

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(unsigned short Val);
```

ARGUMENTS

`Val`
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()

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

FORMAT

```
int inet_addr(char *Address);
```

ARGUMENTS

`Address`
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()

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

```
word inet_lnaof(struct in_addr Address);
```

ARGUMENTS

Address

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()

Builds a complete Internet address from the separate host and network portions.

FORMAT

```
word inet_makeaddr(unsigned int Network, unsigned int Host);
```

ARGUMENTS

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

`Host`
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()

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

```
word inet_netof(struct in_addr Address);
```

ARGUMENTS

Address

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()

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

```
int inet_network(char *Address);
```

ARGUMENTS

Address
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()

Converts an Internet address represented in binary form into an ASCII string suitable for printing.

FORMAT

```
(char *) inet_ntoa(struct in_addr Address);
```

ARGUMENTS

`Address`
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()

Used with `klseek()` and `multinet_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

```
int klread(char *Buffer, unsigned int Size);
```

ARGUMENTS

`Buffer`
The address to which to return the kernel memory.

`Size`
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()

Used with `klread()` and `multinet_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

```
word klseek(unsigned int Position);
```

ARGUMENTS

`Position`
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()

Used with `klseek()` and `multinet_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 similar 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

```
int klwrite(char *Buffer, unsigned int Size);
```

ARGUMENTS

`Buffer`
The address of the data to write into kernel memory.

`Size`
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()

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

```
int listen(short VMS_Channel, unsigned int Backlog);
```

ARGUMENTS

`VMS_Channel`
A channel to the socket.

`Backlog`
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`.

---

multinet_kernel_nlist

A special version of the UNIX 4.3BSD `nlist()` function that reads the symbol table to the MultiNet kernel. Unlike the UNIX 4.3BSD kernel, the MultiNet 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 MultiNet 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 MultiNet software.

To access the symbol table to the MultiNet image that is currently running, read from the file indicated by the logical name `MULTINET_NETWORK_IMAGE:`.

For more information about how to use `multinet_kernel_nlist()`, see `nlist()`.

nlist()

Examines the symbol table in an executable image or symbol table file.

FORMAT

```
int nlist(char *Filename, struct nlist nl[]);
```

ARGUMENTS

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

`nl`
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()

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

```
int ntohl(unsigned long Val);
```

ARGUMENTS

`Val`
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()

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

```
unsigned short ntohs(unsigned short Val);
```

ARGUMENTS

`Val`
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()

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 `MULTINET_ROOT:[MULTINET.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

FORMAT

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

ARGUMENTS

`VMS_Channel`
A channel to the socket.

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

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

`Flags`
Control information that affects the `recv()` function. The `Flags` argument is formed by `OR`ing 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()

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 `MULTINET_ROOT:[MULTINET.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

FORMAT

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

ARGUMENTS

`VMS_Channel`
A channel to the socket.

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

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

`Flags`
Control information that affects the `recvfrom()` function. The `Flags` argument is formed by `OR`ing 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
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
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()

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 `MULTINET_ROOT:[MULTINET.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

FORMAT

```
int recvmsg(short VMS_Channel, struct msghdr *Message, unsigned int
Flags);
```

ARGUMENTS

`VMS_Channel`
A channel to the socket.

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

`Flags`
Control information that affects the `recvmsg()` function. The `Flags` argument is formed by `OR`ing 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()

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.

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 MultiNet `types.h` file, as the definitions differ from the UNIX definitions.

Caution! Process Software recommends that you 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.

Note: The MultiNet 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

```
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)
```

ARGUMENTS

Width
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
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
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
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
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()

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()

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 `MULTINET_ROOT:[MULTINET.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

FORMAT

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

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

ARGUMENTS

`VMS_Channel`
A channel to the socket.

`Buffer`
The address of a buffer containing the data to send.

`Size`
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()

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 `MULTINET_ROOT:[MULTINET.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

FORMAT

```
int sendmsg(short VMS_Channel, struct msghdr *Message, unsigned int
Flags);
```

ARGUMENTS

`VMS_Channel`
A channel to the socket.

`Message`
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`
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

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 `MULTINET_ROOT:[MULTINET.INCLUDE.NETINET]IN.H` is used and the program is compiled with `USE_BSD44_ENTRIES` defined.

FORMAT

```
int sendto(short VMS_Channel, char *Buffer, int Size, unsigned short
Flags, struct sockaddr *To, unsigned int ToLen);
```

ARGUMENTS

`VMS_Channel`
A channel to the socket.

`Buffer`
The address of a buffer containing the data to send.

`Size`
The length of the buffer specified by `Buffer`.

`Flags`
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`

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`
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()

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(unsigned int StayOpen);
```

ARGUMENTS

`StayOpen`
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()

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(unsigned int StayOpen);
```

ARGUMENTS

`StayOpen`
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()

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

FORMAT

```
(void) setprotoent(unsigned int StayOpen);
```

ARGUMENTS

`StayOpen`
Provided only for compatibility with UNIX 4.3BSD, and is ignored by the MultiNet software.

---

setservent()

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

FORMAT

```
(void) setservent(unsigned int StayOpen);
```

ARGUMENTS

`StayOpen`
Provided only for compatibility with UNIX 4.3BSD, and is ignored by the MultiNet software.

---

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 `multinet_root:[multinet.include.sys]socket.h` contains definitions for socket-level options. Options at other protocol levels vary in format and name.

FORMAT

```
int setsockopt(short VMS_Channel, unsigned int Level, unsigned int
OptName, unsigned int OptVal, char *OptLen);
```

ARGUMENTS

`VMS_Channel`
A channel to the socket.

`Level`
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`
The option that is to be manipulated.

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

`OptLen`
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()

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

```
int shutdown(short VMS_Channel, unsigned int How);
```

ARGUMENTS

`VMS_Channel`
A channel to the socket.

`How`
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()

Creates an end point for communication and returns an OpenVMS channel that describes the end point.

FORMAT

```
short socket(unsigned int Address_Family, unsigned int Type, unsigned
int Protocol);
```

ARGUMENTS

Address_Family
An address family with which addresses specified in later operations using the socket should be interpreted. The AF_INET format is currently supported; any additional supported formats will be defined in the include file multinet_root:[multinet.include.sys]socket.h:

Type
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
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()

Deassigns the OpenVMS channel from the MultiNet `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

```
int socket_close(short VMS_Channel);
```

ARGUMENTS

`VMS_Channel`
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()

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 `multinet_root:[multinet.include.sys]ioctl.h`.

FORMAT

```
int socket_ioctl(short VMS_Channel, unsigned int Request, char *ArgP);
```

ARGUMENTS

`VMS_Channel`
A channel to the socket.

`Request`
Which `socket_ioctl()` function to perform.

`ArgP`
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 MultiNet, see the following pages.

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

```
int socket_ioctl(VMS_Channel, FIONBIO, unsigned int *Enable);
```

ARGUMENTS

`Enable`

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

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

FORMAT

```
int socket_ioctl(VMS_Channel, FIONREAD, unsigned int *Count);
```

ARGUMENTS

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

---

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

```
int socket_ioctl(VMS_Channel, SIOCADDRT, struct rtentry *Route);
```

ARGUMENTS

`Route`

A pointer to the address of a `rtentry` structure that describes the route to be added. The `rtentry` structure is defined in `multinet_root:[multinet.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

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 `multinet_kernel_nlist()` to read the routing tables directly from the networking kernel.

FORMAT

```
int socket_ioctl(VMS_Channel, SIOCDELRT, struct rtentry *Route);
```

ARGUMENTS

`Route`

A pointer to the address of a `rtentry` structure that describes the route to be deleted. The `rtentry` structure is defined in `multinet_root:[multinet.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

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

FORMAT

```
int socket_ioctl(VMS_Channel, SIOCATMARK, unsigned int *AtMark);
```

ARGUMENTS

`AtMark`
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

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

FORMAT

```
int socket_ioctl(VMS_Channel, SIOCDARP, struct arpreq *ARP_Req);
```

ARGUMENTS

ARP_Req

The address of an `arpreq` structure that contains the protocol address and the hardware address. The `arpreq` structure is defined in `multinet_root:[multinet.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

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

FORMAT

```
int socket_ioctl(VMS_Channel, SIOCGARP, struct arpreq *ARP_Req);
```

ARGUMENTS

ARP_Req

The address of an `arpreq` structure that contains the protocol address and the hardware address. The `arpreq` structure is defined in `multinet_root:[multinet.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

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

FORMAT

```
int socket_ioctl(VMS_Channel, SIOCSARP, struct arpreq *ARP_Req);
```

ARGUMENTS

ARP_Req

The address of an `arpreq` structure that contains the protocol address and the hardware address. The `arpreq` structure is defined in `multinet_root:[multinet.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

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

```
int socket_ioctl(VMS_Channel, SIOCGIFADDR, struct ifreq
*Interface_Req);
```

ARGUMENTS

`Interface_Req`

The address of an `ifreq` structure that describes the interface from which to retrieve the address. The `ifreq` structure is defined in `multinet_root:[multinet.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

Sets the Internet address of a network interface. Normally, this is done using the `MULTINET 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

```
int socket_ioctl(VMS_Channel, SIOCSIFADDR, struct ifreq
*Interface_Req);
```

ARGUMENTS

`Interface_Req`

The address of an `ifreq` structure that describes the address to be set. The `ifreq` structure is defined in `multinet_root:[multinet.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

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

```
int socket_ioctl(VMS_Channel, SIOCGIFBRDADDR, struct ifreq
*Interface_Req);
```

ARGUMENTS

`Interface_Req`

The address of an `ifreq` structure that describes the interface from which to retrieve the broadcast address. The `ifreq` structure is defined in `multinet_root:[multinet.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

Sets the Internet broadcast address of a network interface. Normally, this is done using the `MULTINET 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

```
int socket_ioctl(VMS_Channel, SIOCSIFBRDADDR, struct ifreq
*Interface_Req);
```

ARGUMENTS

`Interface_Req`

The address of an `ifreq` structure that describes the interface on which to set the broadcast address. The `ifreq` structure is defined in `multinet_root:[multinet.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 modified, such as `se0`.

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

socket ioctl SIOCGIFCONF

Retrieves the list of network interfaces from the networking kernel for further examination by the other SIOCG*xxxx* 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

```
int socket_ioctl(VMS_Channel, SIOCGIFCONF, struct ifconf
*Interface_Config);
```

ARGUMENTS

Interface_Config

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

```
struct ifreq6

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

---

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

```
int socket_ioctl(VMS_Channel, SIOCGIFDSTADDR, struct ifreq
*Interface_Req);
```

ARGUMENTS

`Interface_Req`

The address of an `ifreq` structure that describes the interface from which to retrieve the destination address. The `ifreq` structure is defined in `multinet_root:[multinet.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

Sets the destination Internet address of a point-to-point network interface. Normally, this is done using the `MULTINET 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

```
int socket_ioctl(VMS_Channel, SIOCSIFDSTADDR, struct ifreq
*Interface_Req);
```

ARGUMENTS

`Interface_Req`

The address of an `ifreq` structure that describes the interface on which to set the destination address. The `ifreq` structure is defined in `multinet_root:[multinet.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

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

```
int socket_ioctl(VMS_Channel, SIOCSIFFLAGS, struct ifreq
*Interface_Req);
```

ARGUMENTS

`Interface_Req`

The address of an `ifreq` structure that describes the state of the flags. The `ifreq` structure is defined in `multinet_root:[multinet.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_flags` 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

Sets various control flags on a network interface. Normally this is done using the `MULTINET 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

```
int socket_ioctl(VMS_Channel, SIOCSIFFLAGS, struct ifreq
*Interface_Req);
```

ARGUMENTS

`Interface_Req`
The address of an `ifreq` structure that describes the new state of the flags. The `ifreq` structure is defined in `multinet_root:[multinet.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

Retrieves the network interface metric, or cost. The interface metric is ignored by the MultiNet software, and is not documented further here.

socket ioctl SIOCSIFMETRIC

Sets the network interface metric, or cost. The interface metric is ignored by the MultiNet software, and is not documented further here.

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

```
int socket_ioctl(VMS_Channel, SIOCGIFNETMASK, struct ifreq
*Interface_Req);
```

ARGUMENTS

`Interface_Req`

The address of an `ifreq` structure that describes the interface from which to retrieve the address mask. The `ifreq` structure is defined in `multinet_root:[multinet.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

Sets the Internet address mask of a network interface. Normally, this is done using the `MULTINET 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

```
int socket_ioctl(VMS_Channel, SIOCSIFNETMASK, struct ifreq
*Interface_Req);
```

ARGUMENTS

`Interface_Req`

The address of an `ifreq` structure that describes the interface on which to set the address mask. The `ifreq` structure is defined in `multinet_root:[multinet.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 mask to be set.

socket option SO_BROADCAST

Enables transmission of broadcast messages on the specified socket.

FORMAT

```
int setsockopt(VMS_Channel, SOL_SOCKET, SO_BROADCAST, struct ifreq
*On, sizeof(*On));
```

ARGUMENTS

On

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

Controls the recording of debugging information by the MultiNet networking kernel.

FORMAT

```
int setsockopt(VMS_Channel, SOL_SOCKET, SO_DEBUG, unsigned int *On,
sizeof(*On));
```

ARGUMENTS

On

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

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

```
int setsockopt(VMS_Channel, SOL_SOCKET, SO_DONTROUTE, unsigned int
*On, sizeof(*On));
```

ARGUMENTS

On

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

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

FORMAT

```
int getsockopt(VMS_Channel, SOL_SOCKET, SO_ERROR, unsigned int *Value,
unsigned int *Length);
```

ARGUMENTS

`Value`

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

`Length`

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

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

```
int setsockopt(VMS_Channel, SOL_SOCKET, SO_KEEPALIVE, unsigned int
*On, sizeof(*On));
```

ARGUMENTS

On

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

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

```
int setsockopt(VMS_Channel, SOL_SOCKET, SO_LINGER, struct linger
*Linger, sizeof(*Linger));
```

ARGUMENTS

`Linger`

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

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

```
int setsockopt(VMS_Channel, SOL_SOCKET, SO_OOBINLINE, unsigned int
*On, sizeof(*On));
```

ARGUMENTS

On

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

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

```
int setsockopt(VMS_Channel, SOL_SOCKET, SO_RCVBUF, unsigned int
*Value, sizeof(*Value));
```

ARGUMENTS

Value

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

---

socket option SO_RCVLOWAT

This option exists only for compatibility with UNIX 4.3BSD and has no effect on MultiNet sockets.

socket option SO_RCVTIMEO

This option exists only for compatibility with UNIX 4.3BSD and has no effect on MultiNet sockets.

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

```
int setsockopt(VMS_Channel, SOL_SOCKET, SO_REUSEADDR, unsigned int
*On, sizeof(*On));
```

ARGUMENTS

`On`

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

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

```
int setsockopt(VMS_Channel, SOL_SOCKET, SO_SNDBUF, unsigned int
*Value, sizeof(*Value));
```

ARGUMENTS

Value

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

socket option SO_SNDLOWAT

This option exists only for compatibility with UNIX 4.3BSD and has no effect on MultiNet sockets.

socket option SO_SNDTIMEO

This option exists only for compatibility with UNIX 4.3BSD and has no effect on MultiNet sockets.

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

```
unsigned int *getsockopt(VMS_Channel, SOL_SOCKET, SO_TYPE,
sizeof(*Value));
```

Returns

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

---

socket option TCP_KEEPALIVE

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

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.

FORMAT

```
int setsockopt(VMS_Channel, IPPROTO_TCP, TCP_KEEPALIVE, struct
tcp_keepalive *keepalive, sizeof(struct tcp_keepalive));
```

ARGUMENTS

`Keepalive`
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

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

```
int setsockopt(VMS_Channel, IPPROTO_TCP, TCP_NODELAY, unsigned int
*On, sizeof(*On));
```

ARGUMENTS

On

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()

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.

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(char *String);
```

ARGUMENTS

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

socket_read()

Reads messages from a socket. See also `recv()` and `recvfrom()`. 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`
A channel to the socket.

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

`Size`
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()

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`
A channel to the socket.

`Buffer`
The address of a buffer containing the data to send.

`Size`
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()

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.

SCTP

Support for SCTP (Stream Control Transport Protocol) has been added to the MultiNet C socket library, with the shareable image `MULTINET: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
`MULTINET_ROOT:[MULTINET.INCLUDE.NETINET]SCTP.H`
`MULTINET_ROOT:[MULTINET.INCLUDE.NETINET]SCTP_CONSTANTS.H`
`MULTINET_ROOT:[MULTINET.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:

sctp_opt_info()

Description

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

FORMAT

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

Parameter Usage

sd
The socket descriptor for which the option is requested.

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

opt
Specifies the SCTP socket option to get.

arg
An option-specific structure buffer provided by the caller.

size
Aa 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
```

---

sctp_bindx()

Description

This function 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.

An application can use the `SCTP_BINDX_ADD_ADDR` option to associate additional addresses with an endpoint after calling `bind()`. The `SCTP_BINDX_REM_ADDR` option 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`.

FORMAT

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

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.

sctp_getpaddrs()

Description

sctp_getpaddrs returns all peer addresses in an association. 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.

FORMAT

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

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_getladdrs()

Description

This function returns all locally bound addresses on a socket. 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.

FORMAT

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

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() / sctp_freepaddrs()

Description

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

```
void sctp_freeladdrs(struct sockaddr *addrs);

void sctp_freepaddrs(struct sockaddr *addrs);
```

---

sctp_connectx()

Description

This function 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.

FORMAT

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

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()` or `poll()` for completion by selecting the socket for writing. After select indicates writability, use `getsockopt()` 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.
EAFNOSUPPORT	The passed address didn't 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_getassocid()

Description

This function attempts to look up the specified socket address `addr` and find the respective association identification.

FORMAT

```
sctp_assoc_t sctp_getassocid(int sd, struct sockaddr *addr);
```

Return Values

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

Errors

This 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.

sctp_getaddrlen()

Description

This 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.

FORMAT

```
int sctp_getaddrlen(int family)
```

Return Values

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

Errors

This function can return the following errors:

`EINVAL`	The address family specified does NOT exist.

3. $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 MultiNet-specific $QIO functions are defined in the include file multinet_root:[multinet.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.

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

```
int SYS$QIOW(Efn, New_VMS_Channel, IO$_ACCEPT, IOSB, AstAdr, AstPrm,
Address, AddrLen, VMS_Channel, 0, 0, 0);
```

ARGUMENTS

`New_VMS_Channel`

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`

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`

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
The length of the buffer pointed to by the `Address` argument, in bytes. It must be at least 20 bytes.

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

```
int SYS$QIOW(Efn, VMS_Channel, IO$_ACCEPT_WAIT, IOSB, AstAdr, AstPrm,
0, 0, 0, 0, 0, 0);
```

ARGUMENTS

VMS_Channel
The OpenVMS channel to the INET: device on which the IO$_LISTEN call was performed.

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

```
int SYS$QIOW(Efn, VMS_Channel, IO$_BIND, IOSB, AstAdr, AstPrm, Name,
NameLen, 0, 0, 0, 0);
```

ARGUMENTS

VMS_Channel
A channel to the socket.

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

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

```
int SYS$QIOW(Efn, VMS_Channel, IO$_CONNECT, IOSB, AstAdr, AstPrm,
Name, NameLen, 0, 0, 0, 0);
```

ARGUMENTS

VMS_Channel
A channel to the socket.

Name
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
The length of the Name argument, in bytes.

IO$_GETPEERNAME

Returns the name of the peer connected to the specified socket. It is equivalent to the `getpeername()` socket library function.

FORMAT

```
int SYS$QIOW(Efn, VMS_Channel, IO$_GETPEERNAME, IOSB, AstAdr, AstPrm,
Address, AddrLen, 0, 0, 0, 0);
```

ARGUMENTS

`VMS_Channel`
A channel to the socket.

`Address`
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`
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

Returns the current name of the specified socket. Equivalent to the `getsockname()` socket library function.

FORMAT

```
int SYS$QIOW(Efn, VMS_Channel, IO$_GETSOCKNAME, IOSB, AstAdr, AstPrm,
Address, AddrLen, 0, 0, 0, 0);
```

ARGUMENTS

`VMS_Channel`
A channel to the socket.

`Address`
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`
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

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 `multinet_root:[multinet.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 the S*ocket Option* sections.

FORMAT

```
int SYS$QIOW(Efn, VMS_Channel, IO$_GETSOCKOPT, IOSB, AstAdr, AstPrm,
Level, OptName, OptVal, OptLen, 0, 0);
```

ARGUMENTS

`VMS_Channel`
A channel to the socket.

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

`OptName`
The option that is to be manipulated.

OptVal
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
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

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 multinet_root:[multinet.include.sys]ioctl.h.

FORMAT

```
int SYS$QIOW(Efn, VMS_Channel, IO$_IOCTL, IOSB, AstAdr, AstPrm,
Request, ArgP, 0, 0, 0, 0);
```

ARGUMENTS

VMS_Channel
A channel to the socket.

Request
Which IO$_IOCTL function to perform. The available IO$_IOCTL functions are documented in the socket ioctl sections.

ArgP
A pointer to a buffer whose format and function is dependent on the Request specified.

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

```
int SYS$QIOW(Efn, VMS_Channel, IO$_LISTEN, IOSB, AstAdr, AstPrm,
BackLog, 0, 0, 0, 0, 0);
```

ARGUMENTS

`VMS_Channel`
A channel to the socket.

`Backlog`
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)

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

```
int SYS$QIOW(Efn, VMS_Channel, IO$_RECEIVE, IOSB, AstAdr, AstPrm,
Buffer, Size, Flags, From, FromLen, 0);
```

ARGUMENTS

`VMS_Channel`
A channel to the socket.

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

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

`Flags`
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
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
The length of the buffer pointed to by the From argument, in bytes. It must be at least 18 bytes.

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.

> **Note:** `IO$_SELECT` is only useful for channels assigned to the `INET:` device. It cannot be used for any other VMS I/O device.

FORMAT

```
int SYS$QIOW(Efn, VMS_Channel, IO$_SELECT, IOSB, AstAdr, AstPrm,
Modes, 0, 0, 0, 0, 0);
```

ARGUMENTS

`VMS_Channel`
A channel to the socket.

`Modes`
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

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

```
int SYS$QIOW(Efn, VMS_Channel, IO$_SEND, IOSB, AstAdr, AstPrm, Buffer,
Size, Flags, To, ToLen, 0);
```

ARGUMENTS

VMS_Channel
A channel to the socket.

Buffer
The address of a buffer containing the data to send.

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

Flags
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

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`
An optional argument that contains the length of the address pointed to by the `To` argument.

IO$_SENSEMODE

Reads the active connections status and returns status information for all of the active and listening connections.

FORMAT

```
int SYS$QIO(efn, chan, IO$_SENSEMODE, iosb, astadr, astprm, buffer,
address, conn_type, 0, 0, 0);
```

ARGUMENTS

buffer

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.

address

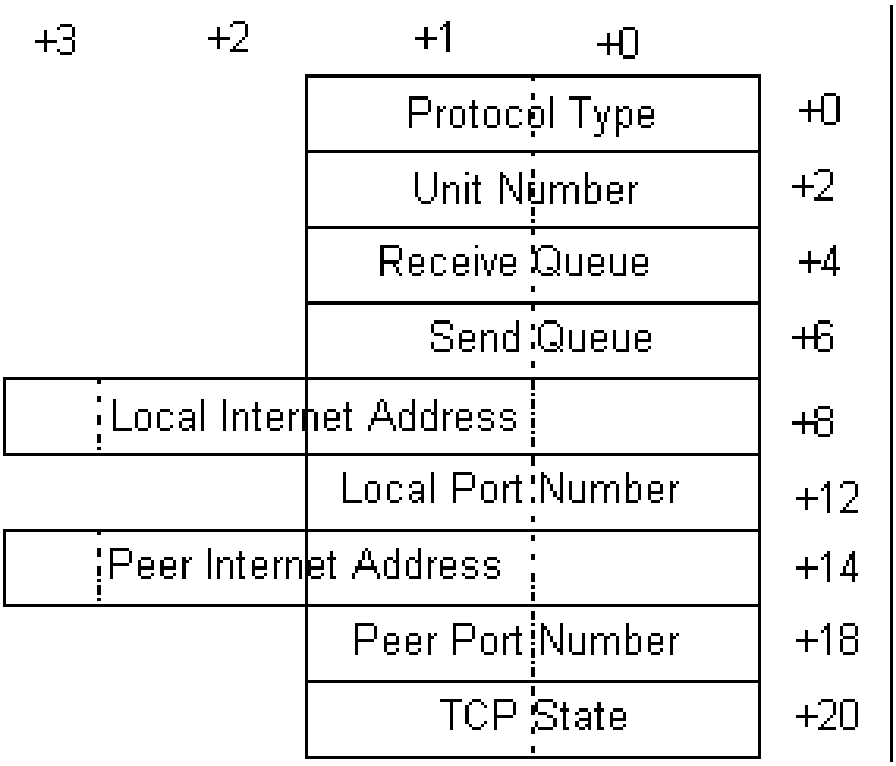
Address of the descriptor for the buffer to receive the status information on the active connections.

value

0 to get information about TCP connections, non-zero to get information about UDP connections. The below diagram 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 3-1for the mask value definitions.



Returns

SS$_BUFFEROVF	Buffer too small for all connections, truncated buffer returned.
SS$_DEVINACT	Device not active. Contact system manager why MultiNet (or INETDRIVER) not started.

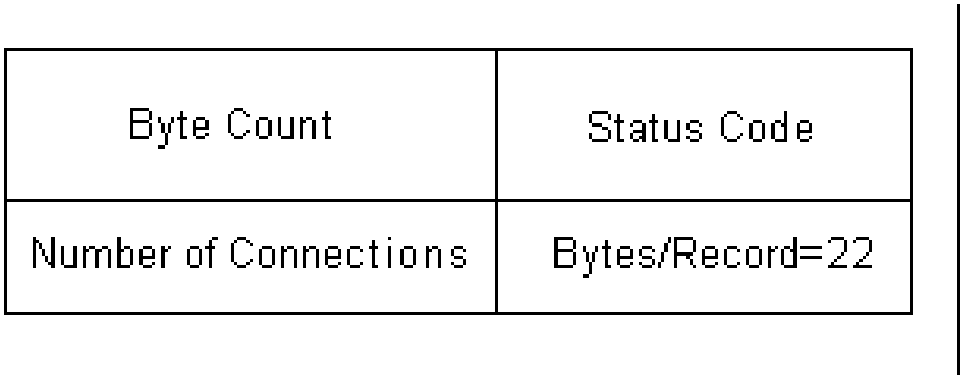
SS$_NORMAL	Success; status information returned

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 the below table 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.

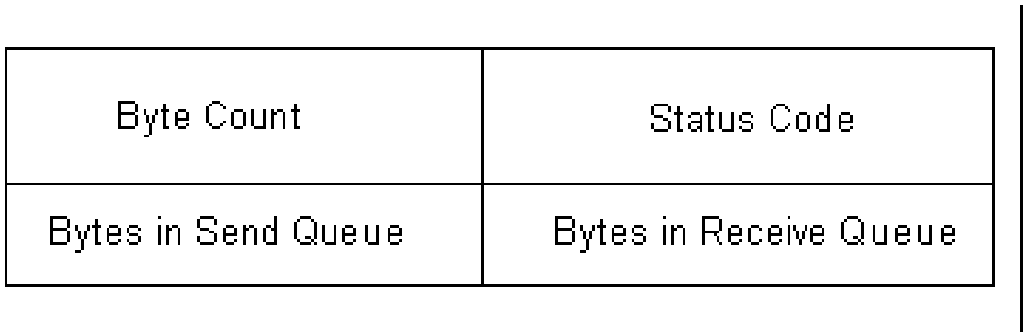
The total number of active connections is returned in the high-order word of the second longword of the I/O status block. This can be greater than the number of reported connections if the buffer is full.

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

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. The below diagram shows the 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

```
int SYS$QIO(efn, chan, IO$_SENSEMODE | IO$M_CTRL, iosb, astadr,
astprm, buffer, address, function, line-id, 0, 0);
```

ARGUMENTS

buffer
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.

address
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 MultiNet currently never returns a counted string for a parameter, this may change in the future.

function
Code that designates the function. The function codes are shown in the below table.

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

line-id
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 `function` set to 1 to read network device information. The information returned in the buffer (specified by `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 `function` set to 8 to read the routing table. The information returned in the buffer (specified by `address`) can consist of multiple records. Each record consists of five longwords, and one record is returned for each table entry.

This setting returns full routing information and is a superset of `function`=2, which was retained for backwards compatibility with existing programs. `function`=2 and `function`=8 return the same table of routing entries, in the following order, except that `function`=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 MULTINET_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. The following table 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.

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 `function`=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 `function`=10 to read network device information. The information returned in the buffer (specified by `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.

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 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 the below table:

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.
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$_DEVINACT	Device not active Contact your system manager to determine why MultiNet was not started.
SS$_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 `function`=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 `function`=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. Later versions of MultiNet may return more information for each entry, which will return a larger Entry Size. Any additional information to be returned in the future will be added to the end of the returned entry.

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

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 MULTINET_SERVER process to hand off a connection to a process that it creates to handle the connection.

FORMAT

```
int SYS$QIOW(Efn, VMS_Channel, IO$_SETCHAR, IOSB, AstAdr, AstPrm,
Flags, 0, 0, 0, 0, 0);
```

ARGUMENTS

VMS_Channel
A channel to the socket.

Flags
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

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

```
int SYS$QIOW(Efn, VMS_Channel, IO$_SETMODE|IO$M_ATTNAST, IOSB, AstAdr,
AstPrm, Routine, Parameter, 0, 0, 0, 0);
```

ARGUMENTS

`Routine`
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`
The argument with which to call the AST routine.

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 `multinet_root:[multinet.include.sys]socket.h` contains definitions for socket-level options. Options at other protocol levels vary in format and name.

FORMAT

```
int SYS$QIOW(Efn, VMS_Channel, IO$_SETSOCKOPT, IOSB, AstAdr, AstPrm,
Level, OptName, OptVal, OptLen, 0, 0);
```

ARGUMENTS

VMS_Channel
A channel to the socket.

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

OptName
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

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

`OptLen`
The length of the buffer pointed to by `OptVal`.

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

```
int SYS$QIOW(Efn, VMS_Channel, IO$_SHUTDOWN, IOSB, AstAdr, AstPrm,
How, 0, 0, 0, 0, 0);
```

ARGUMENTS

`VMS_Channel`
A channel to the socket.

`How`
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

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

```
int SYS$QIOW(Efn, VMS_Channel, IO$_SOCKET, IOSB, AstAdr, AstPrm,
Address_Family, Type, Protocol, 0, 0, 0);
```

ARGUMENTS

`Address_Family`
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 `multinet_root:[multinet.include.sys]socket.h`:

`AF_INET`	Internet (TCP/IP) addresses
`AF_PUP`	Xerox PUP addresses
`AF_CHAOS`	CHAOSnet addresses

`Type`
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`

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

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

```
int SYS$CANCEL(VMS_Channel);
```

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

```
int SYS$DASSGN(VMS_Channel);
```

---

4. 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 MultiNet SNMP agent.

To be able to use your private Management Information Base (MIB) with MultiNet'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)

The SNMP shareable images need to be configured for the SNMP agent to interact with them.

See the *Configuring MultiNet SNMP Agents* chapter of the *MultiNet Installation and Administrator's Guide* for details on configuring the SNMP agent.

SNMP subagent developers should use the include file `SNMP_COMMON.H` found in the `MULTINET_COMMON_ROOT:[MULTINET.INCLUDE]` directory. This file defines the data structures the API uses.

For details on MultiNet's SNMP agent, see *Configuring MultiNet SNMP Agents* in the *MultiNet Installation and Administrator's Guide*.

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

Linking the Extension Agent Image

To link the extension agent image, you need to create an option file. The two examples below are for VAX systems and Alpha/Itanium systems, respectively.

VAX

```
!Note: Exclude SnmpExtensionInitEx if it is not needed.
!See the definition of this routine.
!
UNIVERSAL=SnmpExtensionInit, -
SnmpExtensionQuery, -
SnmpExtensionTrap, -
SnmpExtensionInitEx
SYS$SHARE:VAXCRTL/SHARE
!
!List your object/library files here
```

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-name option-file/OPT
```

image-name is the name of the shareable image you want to build, and *option-file* is the option file mentioned above.

Installing the Extension Agent Image

You should copy the shareable image you build for your SNMP subagent to the SYS$SHARE.

> **CAUTION!** 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.

Debugging Code

SNMP subagent developers can use a debug logical, MULTINET_SNMP_DEBUG, to set certain debug masks. Define the logical as follows and use the *mask* values in the below table:

```
$ DEFINE MULTINET_SNMP_DEBUG mask
```

Mask Value	Description
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)

Subroutine Reference

The following pages include the subroutine descriptions.

SnmpExtensionInit

Initializes the SNMP subagent and registers the subagent in the SNMPD agent. The subagent calls this routine at startup.

Format

```
int SnmpExtensionInit (trap-alert-routine, time-zero-reference, trap-event, supported-view);
```

Arguments

`trap-alert-routine`
Address of the routine the subagent should call when it is ready to send a trap.

`trap-event`
Currently unused.

`time-zero-reference`
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`
Prefix of the MIB tree the subagent supports.

Return Values

- `TRUE` – Subagent initialized successfully
- `FALSE` - Subagent initialization failed

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`.

> **Note:** Only implement this routine if you have multiple MIB subtrees in your extendible agent. The MultiNet SNMP agent executes this routine if it exists and overwrites MIB information set by `SnmpExtensionInit`.

Format

```
int SnmpExtentionInitEx(supported-view);
```

Arguments

`supported-view`
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);
}
```

Return Values

- `TRUE` – Returning first or next MIB subtree
- `FALSE` - All MIB subtrees were passed back

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

```
int SnmpExtensionQuery (request-type, var-bind-list, error-status,
error-index);
```

Arguments

`request-type`
Identifies the type of request: `GET`, `SET`, or `GET NEXT`.

`var-bind-list`
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`
Status of a failed operation.

`error-index`
The index of the variable in the variable binding list for which the operation failed.

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

SnmpExtensionTrap

Sends a trap from the subagent. If the subagent wants to send a trap, it must first call the `trap-alert-routine` (see the `SnmpExtensionInit` routine). The trap-alert-routine should be called with two parameters (`objids`, `idlength`). For example:

If the DNS process wants to send trap information to all the communities that are interested then the DNS server must be running and the object IDs 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 Process Software
- 1,2,1,1,1 are the MIB object IDs for the DNS process
- 3,1 are the object IDs 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

```
int SnmpExtensionTrap (enterprise, generic-trap, specific-trap, time-stamp, var-bind-list);
```

Arguments

`enterprise`
The prefix of the MIB for the enterprise sending the trap.

`generic-trap`
The generic enterprise trap ID.

`specific-trap`
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`
The time at which the trap was generated.

`var-bind-list`
The list of name-value pairs. This list contains name and value of the MIB variable for which the trap is generated.

Return Values

- `TRUE` - More traps to be generated
- `FALSE` - No more traps to be generated

---

5. RPC Fundamentals

Introduction

MultiNet RPC Services must be used with the HP 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

What Are RPC Services?

RPC Services are a set of software development tools that allow you to build distributed applications on OpenVMS systems.

MultiNet 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)

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 an Alpha 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 hide 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.

Components of RPC Services

RPC Services comprises the following components:

- Run-time libraries (RTLs)
- RPCGEN compiler
- Port mapper
- RPC information

Run-Time Libraries (RTLs)

RPC Services provides a single shareable RTL. The library contains:

- RPC client and server routines
- XDR routines

The *RPC RTL Management Routines*, Chapter 10, and the chapters that follow it describe the RTLs in detail.

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. The *RPCGEN Compiler*, Chapter 8, describes how to use RPCGEN.

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).
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.

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 *RPC Information* in Chapter 12, *Building Distributed Applications*, for details.)

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.

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.

RPC Processing Flow

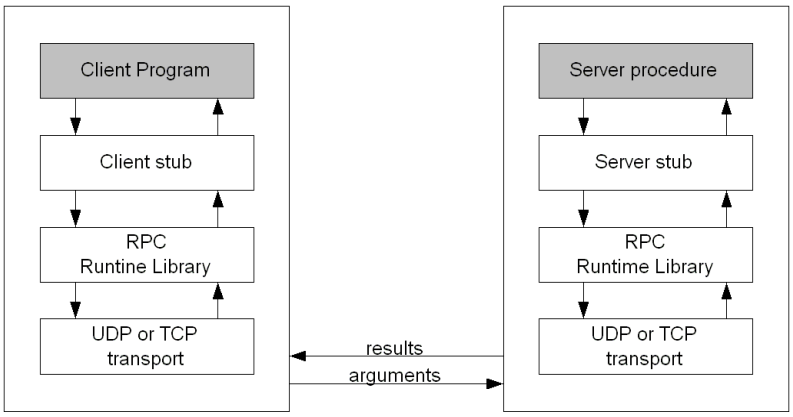
Remote and local procedure calls share some similarities. In both cases, a calling process makes arguments available to a procedure. The procedure uses the arguments to compute a result, then returns the result to the caller. The caller uses the results of the procedure and resumes execution.

The below diagram 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 7, *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
3. Using RPC RTL calls, the client stub sends the packet to the UDP or TCP layer for transmission to the server.
4. The packet travels on the network to the server, up through the layers to the server stub.
5. 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.

RPC Processing Flow:



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.

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.

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.

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.

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.

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.

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.

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.

Transport Protocols

RPC Services uses the transport protocols listed in the below table. The RPC client and server must use the same transport protocol for a given transaction.

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
	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

> **Note:** You must use the HP C Socket Library with RPC Services.

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.

The below table shows how the XID caches differ for the UDP and UDPA/TCPA transports.

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

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

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.

Execution Guarantees

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.

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.

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.

The below table shows the differences between normal RPC and 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.

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

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 the below table to determine which program numbers are available. The numbers are in groups of hexadecimal 20000000.

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.

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.

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.

Additional Terms

Before writing RPC applications, you should be familiar with the terms in the below table:

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 the OpenVMS 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 *RPC RTL Client Routines*:
	`authnone_create`
	`clnt_create`
	`clnt_perror/clnt_sperror`
	`authunix_create`
	`clnttcp_create`
	`authunix_create_default`
	`clntudp_create/clntudp_bufcreate`

Port	An abstract point through which a datagram passes from the host layer to the application layer protocols.

Server handle	Information that uniquely identifies the server. Content varies according to the transport being used. See the following routines in *RPC RTL Server Routines*:
	`svcudp_create/svctcp_create`
	`svc_destroy`
	`svc_freeargs`
	`svc_getargs`
	`svc_register`
	`svc_sendreply`

Socket	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.

6. Building Distributed Applications with RPC

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

Distributed Application Components

The below table lists the components of a distributed application.

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.

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. The *RPC Fundamentals*, Chapter 6, describes the characteristics of each RPC interface layer.
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. The *RPCGEN Compiler,* Chapter 8, explains exactly what interface definitions must contain.

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. The below table lists the texts you may use as 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.
The *RPC RTL Client Routines* chapter and those that follow	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.

```
$ 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.

```
$ 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 `MULTINET CONFIGURE/SERVER` command to start it.

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 HP's documentation for details.
2. Run the client main program.

Obtaining RPC Information

You can request a listing of all programs registered with a Port Mapper.

To request a listing of all programs that are registered with the Port Mapper, enter the `MULTINET SHOW /RPC_PORTMAP` command in the following format at the DCL prompt:

```
$ MULTINET SHOW /RPC_PORTMAP
```

If you add /REMOTE_HOST=*hostname* to this command:

```
$ MULTINET 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. Sample RPC information output:

```
$ MULTINET SHOW/RPC_PORTMAP
  MultiNet 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
```

7. RPCGEN Compiler

Introduction

This chapter is for RPC programmers.

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*, for a complete description of the application development process.

`RPCGEN` allows you to use the highest layer of the RPC programming interface. The *RPC Fundamentals*, Chapter 6, provides details on these layers.

Software Requirements

The following software must be installed on your system before you can use `RPCGEN`:

- OpenVMS 5.5 or higher
- HP C compiler V3.2 or later

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

Below is a sample 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.

Output Files

RPCGEN output files contain code in C language. The below table lists the RPCGEN output files and summarizes their purpose. You can edit RPCGEN output files during application development.

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 format, and vice versa.
Header	Contains common definitions, such as those needed for any structures being passed.

The *Invoking RPC* section explains how to request specific output files.

The below table shows the conventions you should use to name output files.

File	Output Filename
Client stub	*inputname*_CLNT.C
Server stub	*inputname*_SVC.C
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 the above table 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.

Preprocessor Directives

RPCGEN runs the input files through the C preprocessor before compiling. You can use the macros listed in the below table 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.

File	Macro
Client stub	RPC_CLNT
Server stub	RPC_SVC
Header file	RPC_HDR
XDR filter routines	RPC_XDR

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

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
```

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

Examples:

```
$ 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.

```
$ 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.

```
$ 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.

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
```

Error Handling

RPCGEN stops processing when it encounters an error. It indicates which line the error is on.

Restrictions

`RPCGEN` does not support the following:

- The syntax `int x, y;`. You must write this as `int x; int y;`

8. RPC RTL Management Routines

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.

Management Routines

The RPC RTL contains:

- RPC management routines
- RPC client and server routines for the UDP and TCP transport layers
- On VAX and 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 Hewlett-Packard 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.

Routine Name Conventions

In this chapter, all routines are documented according to their standard UNIX names.

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 below:

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.
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.

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

get_myaddress

Returns the internet address of the local host.

Format

```
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

Gets an RPC entry.

Format

```
struct rpcent *getrpcbynumber(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.

```
struct rpcent

{
  char *r_name;        /* name of server for this RPC program */
  char **r_aliases;    /* zero-terminated list of alternate names */
  long r_number;       /* RPC program number */
};
```

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.

Returns

A NULL pointer is returned on EOF or error.

getrpcport

Gets an RPC port number.

Format

```
int getrpcport(char *host, int prognum, int versnum, int 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.

9. RPC RTL Client Routines

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.

Common Arguments

Many client, Port Mapper, and server routines use the same arguments.

The below table 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

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.

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.

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.

---

authnone_create

Creates and returns a null RPC authentication handle for the client process.

Format

```
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.

---

authunix_create

Creates and returns an RPC authentication handle for the client process. Use this routine when the server requires UNIX-style authentication.

Format

```
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.

authunix_create_default

Calls the `authunix_create` routine and provides default values as arguments.

Format

```
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 the table below.

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` routine.

---

callrpc

Format

```
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 the *Common Arguments* table 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.

Returns

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` routine to translate failure status codes into messages.

---

clnt_broadcast

Broadcasts a remote procedure call to all local networks, using the broadcast address.

Format

```
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 the *Common Arguments* table 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(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.

The below table indicates how large a broadcast message can be.

Line	Maximum Size
Ethernet	1500 bytes
proNet	2044 bytes

Returns

This routine returns diagnostic values defined in the `CLNT.H` file for `enum clnt_stat`.

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 the *Common Arguments* table 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` routine to print messages for any errors that occurred.

Returns

This routine returns diagnostic values defined in the `CLNT.H` file for `enum clnt_stat`.

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 the below table.

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

* 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.

Returns

This routine returns TRUE if it succeeds, and FALSE if it fails.

---

clnt_create

Creates an RPC client handle.

Format

```
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 the Common Arguments table 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` 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.

Returns

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 routine to obtain diagnostic information.

---

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.

---

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.

clnt_pcreateerror / clnt_spcreateerror

Return a message indicating why RPC could not create a client handle.

Format

```
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` routine prints a message to `SYS$OUTPUT`.

The `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` supports.

The `clnt_pcreateerror` 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` routine fails.

clnt_perrno / clnt_sperrno

Return a message indicating why the callrpc or clnt_broadcast routine failed to create a client handle.

Format

```
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` routine prints a message to `SYS$OUTPUT`.

The `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` supports.

To save the string, copy it into your own memory space.

clnt_perror / clnt_sperror

Return a message if the `clnt_call` routine fails.

Format

```
void clnt_perror (CLIENT *clnt, char *s);
char *clnt_sperror (CLIENT *clnt, char *s);
```

Arguments

`clnt`
See the *Common* Arguments table for a description of the above argument.

`s`
String containing the message to output.

Description

Use these routines after `clnt_call`.

The `clnt_perrorc` routine prints an error message to `SYS$OUTPUT`.

The `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` supports.

The `clnt_sperror` routine overwrites the string with each call. Copy the string into your own memory space if you want to save it.

---

clntraw_create

Returns an RPC client handle. The remote procedure call uses the IP transport.

Format

```
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 the *Common* Arguments table 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` routine shows how to set up `sockp` to specify a port. See the Common Arguments table 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.

Returns

The clntraw_createroutine 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 routine to obtain additional diagnostic information.

---

clnttcp_create

Returns an RPC client handle. The remote procedure call uses the TCP transport.

Format

```
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 the *Common Arguments* table 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` routine shows how to set up `sockp` to specify a port. See the *Common Arguments* table 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_createroutine` 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.

Returns

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` routine to obtain additional diagnostic information.

---

clntudp_create / clntudp_bufcreate

Returns an RPC client handle. The remote procedure call uses the UDP transport.

Format

```
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 the *Common Arguments* table 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` routine only for procedures that handle messages shorter than 8K bytes. Use the `clntudp_bufcreate` routine for procedures that handle messages longer than 8K bytes.

The `clntudp_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.

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.

Returns

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` routine.

10. RPC RTL Port Mapper Routines

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.

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. The table below**Error! Reference source not found.** summarizes the purpose of each Port Mapper routine.

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.

Port Mapper Arguments

Port Mapper routines use many of the same arguments as client routines. See the *Common Arguments* table in the *RPC RTL Client Routines* chapter for a list of these arguments.

The following sections describe each Port Mapper routine in detail.

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 `MULTINET SHOW /RPC_PORTMAP` command uses this routine.

Returns

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.

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 the *Common Arguments* table in the *RPC RTL Client Routines* chapter for a list of these arguments.

Returns

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.

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 the *Common Arguments* table in the *RPC RTL Client Routines* chapter 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.

Returns

This routine returns diagnostic values defined in the `CLNT.H` file for `enum clnt_stat`.

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 the *Common Arguments* table in the *RPC RTL Client Routines* chapter for a list of these arguments.

port
Remote port number.

Description

The `pmap_set` routine 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.

Returns

The `pmap_set` routine returns TRUE if it succeeds, and FALSE if it fails.

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 the *Common Arguments* table in the *RPC RTL Client Routines* chapter 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.

Returns

The `pmap_unset` routine returns TRUE if it succeeds, FALSE if it fails.

11. RPC RTL Server Routines

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.

Server Routines

The server routines are called by the server program or the server stub procedures. The below table lists each server routine and summarizes its purpose.

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.

`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_bufcreate / svcudp_create`	Returns address of server handle that uses UDP transport. For procedures that pass messages longer 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

Performs creation and registration tasks for the server.

Format

```
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 the *Common Arguments* table in the *RPC RTL Client Routines* chapter 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(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.

Returns

The `registerrpc` routine returns zero if it succeeds, and -1 if it fails.

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.

---

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 the *Common Arguments* table in the *RPC RTL Client Routines* chapter for a list of these arguments.

Description

The `svc_freeargs` routine calls the `xdr_free` routine.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

---

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 the *Common Arguments* table in the *RPC RTL Client Routines* chapter for a list of these arguments.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

svc_getreqset

Reads data for each server connection.

Format

```
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);
}
```

svc_register

Adds the specified server to a list of active servers, and registers the service program with the Port Mapper.

Format

```
bool_t svc_register (SVCXPRT *xprt, u_long prognum, u_long versnum,
void (*dispatch) (), u_long protocol);
```

Arguments

xprt, prognum, versnum
See the *Common Arguments* table in the *RPC RTL Client Routines* chapter 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(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.

Returns

The `svc_register` routine returns TRUE if it succeeds and FALSE if it fails.

svc_run

Waits for RPC requests and calls the `svc_getreqset` routine to dispatch to the appropriate RPC service program.

Format

```
void svc_run();
```

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`.

---

svc_sendreply

Sends the results of a remote procedure call to the client.

Format

```
bool_t svc_sendreply (SVCXPRT *xprt, xdrproc_t outproc, caddr_t *out);
```

Arguments

xprt, outproc, out
See the *Common Arguments* table in the *RPC RTL Client Routines* chapter for a list of these arguments.

Description

The routine sends the results of a remote procedure call to the client.

Returns

These routines return TRUE if they succeed and FALSE if they fail.

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

```
void svc_unregister (u_long prognum, u_long versnum);
```

Arguments

`prognum, versnum`
See the *Common Arguments* table in the *RPC RTL Client Routines* chapter for a list of these arguments.

svcerr_auth / svcerr_decode / svcerr_noproc / svcerr_noprog / svcerr_progvers / svcerr_systemerr / svcerr_weakauth

Sends various error codes to the client process.

Format

```
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` routine. The value of `why` is `AUTH_TOOWEAK`, which means "access permission denied."

---

svcfd_create

Returns the address of a structure containing a server handle for the specified TCP socket.

Format

```
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.

Returns

This routine returns zero if it fails.

svcraw_create

Creates a server handle for memory-based Sun RPC for simple testing and timing.

Format

```
SVCXPRT svcraw_create();
```

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.

Returns

This routine returns NULL if it fails.

svctcp_create

Returns the address of a server handle that uses the TCP transport.

Format

```
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.

Returns

The `svctcp_create` routine returns either the address of the server handle, or zero (if it could not create the server handle).

---

svcudp_create / svcudp_bufcreate

Returns the address of a server handle that uses the UDP transport.

Format

```
SVCXPRT *svcudp_create (int sock);

SVCXPRT *svcudp_bufcreate (int sock, u_long sendsize, u_long
recvsize);
```

Arguments

sock
Socket for this service. These routines create a new socket if you enter `RPC_ANYSOCK`. If the socket is not bound to a UDP port, the routines bind 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_bufcreate` routine for procedures that pass messages longer than 8Kbytes.

Returns

These routines return either a server handle, or zero (if they could not create the server handle).

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 *RPC Fundamentals*, Chapter 6, 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");
```

Returns

This routine returns TRUE if it enables the XID cache, and FALSE if the cache was previously enabled or an error occurs.

---

xprt_register

Adds a TCP or UDP server socket to a list of sockets.

Format

```
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.

xprt_unregister

Removes a TCP or UDP server socket from a list of sockets.

Format

```
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.

12. RPC RTL XDR Routines

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.

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.

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.

The below table shows the order in which XDR routines perform encoding and decoding.

Client	**Server**
1. Encodes arguments 2. Decodes results 3. Frees results from memory	1. Decodes arguments 2. Encodes results 3. Frees arguments from memory

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.

Quick Reference

The below table lists the XDR routines that encode and decode data.

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`	VAX 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	VAX 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

The below table lists the XDR routines that perform various support functions.

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

The below table lists the upper layer XDR routines that support RPC.

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

Converts an RPC reply message from local format to XDR format, or the other way around.

Format

```
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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_array

Converts a variable-length array from local format to XDR format, or the other way around.

Format

```
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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_authunix_parms

Converts UNIX-style authentication information from local format to XDR format, or the other way around.

Format

```
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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_bool

Converts a boolean value from local format to XDR format, or the other way around.

Format

```
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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_bytes

Converts bytes from local format to XDR format, or the other way around.

Format

```
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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_callhdr

Encodes the static part of an RPC request message header.

Format

```
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.

Returns

This routine always returns TRUE.

xdr_callmsg

Converts an RPC request message from local format to XDR format, or the other way around.

Format

```
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`.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_char

Converts a character from local format to XDR format, or the other way around.

Format

```
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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_double

Converts a double-precision floating point number between local and XDR format.

Format

```
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 VAX D format floating point numbers
`xdr_double_G`	Converts VAX 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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_enum

Converts an enumerated type from local format to XDR format, or the other way around.

Format

```
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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_float

Converts a floating point value from local format to XDR format, or the other way around.

Format

```
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 VAX 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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_free

Deallocates a data structure from memory.

Format

```
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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_hyper

Converts a VAX 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 VAX quad word format, with the low-order longword first in memory.

Description

This routine provided the same functionality as the `xdr_u_hyper` routine.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_int

Converts one four-byte integer from local format to XDR format, or the other way around.

Format

```
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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_long

Converts one longword from local format to XDR format, or the other way around.

Format

```
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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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

```
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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_opaque_auth

Converts authentication information from the local data format to XDR format, or the other way around.

Format

```
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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_pmap

Converts port mapper parameters from the local data format to XDR format, or the other way around.

Format

```
#include "MULTINET_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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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(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));
}
```

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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`.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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`.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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`.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_u_hyper

Converts a VAX 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 VAX format, with the low-order longword first in memory.

Description

This routine provides the same functionality as the `xdr_hyper` routine.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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`.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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`.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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`.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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 `dfault` routine.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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.

Returns

This routine always returns TRUE.

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 2^{32} - 1.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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

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(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(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 MultiNet. 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.

---

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`.

Returns

This routine returns TRUE if it succeeds and FALSE if it fails.

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`.

Returns

This routine returns TRUE if it reaches the end of the data stream, and FALSE if it finds more data to read.

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`.

Returns

This routine returns TRUE if it has skipped to the start of a record. Otherwise, it returns FALSE.

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 stdio stream for the specified file.