Compaq TCP/IP Services for OpenVMS

Sockets API and System Services Programming

Order Number: AA–LU51L–TE

January 2001

This manual describes how to use Compaq TCP/IP Services for OpenVMS to develop network applications using Berkeley Sockets or OpenVMS system services.

Revision/Update Information:	This manual supersedes the *Compaq TCP/IP Services for OpenVMS System Services and C Socket Programming*, Version 5.0.
Software Version:	Compaq TCP/IP Services for OpenVMS Version 5.1
Operating System:	OpenVMS Alpha Versions 7.1 and 7.2-1 OpenVMS VAX Versions 7.1 and 7.2

Compaq Computer Corporation
Houston, Texas

ZK6529

This document is available on CD-ROM.

This document was prepared using DECdocument, Version 3.3-1b.

Contents

3 Using the Sockets API

4 Sockets API Reference

5 Using the $QIO System Service

6 OpenVMS System Services Reference

Figures

Tables

Preface

The Compaq TCP/IP Services for OpenVMS product is the Compaq implementation of the TCP/IP networking protocol suite and internet services for OpenVMS Alpha and OpenVMS VAX systems.

A layered software product, TCP/IP Services provides a comprehensive suite of functions and applications that support industry-standard protocols for heterogeneous network communications and resource sharing.

See the *Compaq TCP/IP Services for OpenVMS Installation and Configuration* manual for information about installing, configuring, and starting this product.

This manual describes how to use TCP/IP Services to develop network applications using Berkeley Sockets or OpenVMS system services.

Intended Audience

This manual is intended for experienced programmers who want to write network application programs that run in the TCP/IP Services environment. Readers should be familiar with the C programming language, TCP/IP protocols, and networking concepts.

Document Structure

This manual contains the following chapters and appendixes:

- Chapter 1 describes the application programming interfaces that TCP/IP Services supports.
- Chapter 2 describes the typical function calls for developing network applications using the TCP and UDP protocols and either the Sockets API or OpenVMS system services programming interface. To help network programmers write internet applications, this chapter provides examples of network applications using the Sockets API and OpenVMS system services.
- Chapter 3 discusses information to consider when writing portable network applications using the Sockets API.
- Chapter 4 contains Sockets API reference information.
- Chapter 5 describes how to use $QIO system services and data structures to write network applications using OpenVMS system services.
- Chapter 6 contains OpenVMS system services and I/O function reference information pertinent to TCP/IP Services. This information supplements the OpenVMS system services programming information contained in *OpenVMS System Services Reference*.
- Appendix A lists socket options supported by both programming interfaces.
- Appendix B lists IOCTL requests.

- Appendix C describes TCP/IP Services data types.
- Appendix D lists Sockets API error codes and equivalent OpenVMS system services status codes.
- Appendix E contains client/server application examples using the TCP and UDP protocols.

Related Documentation

Table 1 lists the documents available with this version of TCP/IP Services.

Table 1 TCP/IP Services Documentation

Manual	Contents
DIGITAL TCP/IP Services for OpenVMS Concepts and Planning	This manual provides conceptual information about networking and the TCP/IP protocol including a description of the Compaq implementation of the Berkeley Internet Name Domain (BIND) service and the Network File System (NFS). It outlines general planning issues to consider before configuring your system to use the TCP/IP Services software. This manual also describes the manuals in the documentation set, provides a glossary of terms and acronyms for the TCP/IP Services software product, and documents how to contact the InterNIC Registration Service to register domains and access Request for Comments (RFCs).
Compaq TCP/IP Services for OpenVMS Release Notes	This text file describes new features and changes to the software including installation, upgrade, configuration, and compatibility information. These notes also describe new and existing software problems and restrictions, and software and documentation corrections. Print this text file at the beginning of the installation procedure and read it before you install TCP/IP Services.
Compaq TCP/IP Services for OpenVMS Installation and Configuration	This manual explains how to install and configure the TCP/IP Services layered application product.
DIGITAL TCP/IP Services for OpenVMS User's Guide	This manual describes how to use the applications available with TCP/IP Services such as remote file operations, email, TELNET, TN3270, and network printing. This manual explains how to use these services to communicate with systems on private internets or on the worldwide Internet.
Compaq TCP/IP Services for OpenVMS Management	This manual describes how to configure and manage the TCP/IP Services product. Use this manual with the *Compaq TCP/IP Services for OpenVMS Management Command Reference* manual.
Compaq TCP/IP Services for OpenVMS Management Command Reference	This manual describes the TCP/IP Services management commands. Use this manual with the *Compaq TCP/IP Services for OpenVMS Management* manual.
Compaq TCP/IP Services for OpenVMS Management Command Quick Reference Card	This reference card lists the TCP/IP management commands by component and describes the purpose of each command.

(continued on next page)

Table 1 (Cont.) TCP/IP Services Documentation

Manual	Contents
Compaq TCP/IP Services for OpenVMS UNIX Command Reference Card	This reference card contains information about commonly performed network management tasks and their corresponding TCP/IP management and Compaq *Tru64* UNIX command formats.
DIGITAL TCP/IP Services for OpenVMS ONC RPC Programming	This manual presents an overview of high-level programming using open network computing remote procedure calls (ONC RPC). This manual also describes the RPC programming interface and how to use the RPCGEN protocol compiler to create applications.
Compaq TCP/IP Services for OpenVMS Sockets API and System Services Programming	This manual describes how to use the Sockets API and OpenVMS system services to develop network applications.
Compaq TCP/IP Services for OpenVMS SNMP Programming and Reference	This manual describes the Simple Network Management Protocol (SNMP) and the SNMP application programming interface (eSNMP). It describes the subagents provided with TCP/IP Services, utilities provided for managing subagents, and how to build your own subagents.
Compaq TCP/IP Services for OpenVMS Tuning and Troubleshooting	This manual provides information about how to isolate the causes of network problems and how to tune the TCP/IP Services software for the best performance.
Compaq TCP/IP Services for OpenVMS Guide to IPv6	This manual describes the IPv6 environment, the roles of systems in this environment, the types and function of the different IPv6 addresses, and how to configure TCP/IP Services to access the 6bone network.

For additional information about Compaq *OpenVMS* products and services, access the Compaq website at the following location:

`http://www.openvms.compaq.com/`

For a comprehensive overview of the TCP/IP protocol suite, you might find the following books useful.

- *Internetworking with TCP/IP Vol 1: Principles, Protocols,and Architecture* by Douglas Comer, Prentice Hall Englewood Cliffs, New Jersey 07632, ISBN 0-13-468505-9.
- *UNIX Network Programming Volume 1 Networking APIs: Sockets and XTI* by W. Richard Stevens, Prentice Hall PTR, Upper Saddle River, NJ 07458, ISBN 0-13-490012-X.

Reader's Comments

Compaq welcomes your comments on this manual. Please send comments to either of the following addresses:

Internet	**openvmsdoc@compaq.com**
Mail	Compaq Computer Corporation OSSG Documentation Group, ZKO3-4/U08 110 Spit Brook Rd. Nashua, NH 03062-2698

How To Order Additional Documentation

Use the following World Wide Web address to order additional documentation:

`http://www.openvms.compaq.com/`

If you need help deciding which documentation best meets your needs, call 800-282-6672.

Conventions

TCP/IP Services is used to mean both:

- Compaq TCP/IP Services for OpenVMS Alpha
- Compaq TCP/IP Services for OpenVMS VAX

The following conventions are used in this manual. In addition, please note that all IP addresses are fictitious.

Ctrl/*x*	A sequence such as Ctrl/*x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.) In the HTML version of this document, this convention appears as brackets, rather than a box.
. . .	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose choices in parentheses if you specify more than one.
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.
|	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are optional; within braces, at least one choice is required. Do not type the vertical bars on the command line.

{ }	In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line.
bold text	This typeface represents the introduction of a new term. It also represents the name of an argument, an attribute, or a reason.
italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER=*name*), and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
Monospace text	Monospace type indicates code examples and interactive screen displays. This typeface indicates UNIX system output or user input, commands, options, files, directories, utilities, hosts, and users. In the C programming language, this typeface identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1
Application Programming Interfaces

The application programming interfaces available with Compaq TCP/IP Services for OpenVMS allow programmers to write network applications that are independent of the underlying communication facilities. This means that the system can support communications networks that use different sets of protocols, naming conventions, and hardware platforms.

The TCP/IP Services product supports two network communication application programming interfaces (APIs):

- Berkeley Software Distribution (BSD) Sockets
- OpenVMS system services

1.1 BSD Sockets

The Sockets application programming interface (API) supports only the C programming language. The benefits of using this API include:

- Ease of use.
- Portability — you can create common code for use on UNIX, OpenVMS, and many other platforms.
- 64-bit addressing capability on OpenVMS Alpha systems.

See Chapter 4 for a detailed description of Sockets API functions.

1.2 OpenVMS System Services

Each step in the Sockets communications process has a corresponding OpenVMS system service routine. The benefits of using OpenVMS system services include:

- Improved application performance
- 64-bit addressing capability on OpenVMS Alpha systems
- Finer granularity of control
- Easier asynchronous programming
- Support for the following products
  - MACRO-32
  - BLISS-32
  - Compaq Ada
  - Compaq BASIC
  - Compaq C
  - Compaq C++

- Compaq COBOL
- Compaq Fortran
- Compaq Pascal

See Chapter 6 for a detailed description of OpenVMS system service calls.

1.3 Application Development Files

TCP/IP Services provides definition files, a shared library file for use in developing network applications and programming example files to assist in learning how to develop network applications.

1.3.1 Definition Files

Table 1–1 lists the definition files that are included with TCP/IP Services in the SYS$LIBRARY directory. Specific languages may also supply additional files that define structures related to network programming. Check the documentation for the language you are using.

Table 1–1 Network Definition Files

File	Description
TCPIP$INETDEF.ADA	Ada definition file
TCPIP$INETDEF.BAS	BASIC definition file
TCPIP$INETDEF.FOR	Fortran definition file
TCPIP$INETDEF.H	C and C++ definition file
TCPIP$INETDEF.MAR	MACRO-32 definition file
TCPIP$INETDEF.PAS	Pascal definition file
TCPIP$INETDEF.PLI	PL/I definition file
TCPIP$INETDEF.R32	BLISS-32 definition file

Compaq provides header files, data types, and support functions to facilitate OpenVMS system services programming. The header files provide definitions for constants. Table 1–2 lists the available files.

Table 1–2 C Language Definition Files

Header File	Description
Common Industry Standard	
IN.H	Internet system. Constants, functions, and structures
INET.H	Network address information
NETDB.H	Network database library information
SIGNAL.H	UNIX style signal value definitions
SOCKET.H	BSD Sockets API

(continued on next page)

Table 1–2 (Cont.) C Language Definition Files

Header File	Description
OpenVMS Related	
DESCRIP.H	OpenVMS descriptor
IOCTL.H	I/O control
IODEF.H	I/O function codes
LIB$FUNCTIONS.H	Run-time library function signatures
SSDEF.H	System services status code
STARLET.H	System services calls
TCPIP$INETDEF.H	TCP/IP network constants, functions, and structures
Standard UNIX	
STDIO.H	Standard UNIX I/O functions
STDLIB.H	Standard UNIX library functions
STRING.H	String-handling functions

1.3.2 Libraries

Table 1–3 lists the routine libraries included with TCP/IP Services.

Table 1–3 Sockets API Libraries

File	Location	Description
TCPIP$IPC_SHR.EXE	SYS$LIBRARY	Sockets API Run-Time Library
TCPIP$LIB.OLB	TCPIP$LIBRARY	BSD Version 4.4 Sockets object library

1.3.3 Programming Examples

Table 1–4 and Table 1–5 summarize the programming examples included with TCP/IP Services in the TCPIP$EXAMPLES directory. Most of these examples consist of a client and a corresponding server. Appendix E contains printed examples of the programs described in Table 1–4 and Table 1–5.

Table 1–4 TCP Programming Examples

File	Description
TCPIP$TCP_SERVER_SOCK.C TCPIP$TCP_CLIENT_SOCK.C	Example TCP/IP client and server using the Sockets API.
TCPIP$TCP_SERVER_SOCK_AUXS.C	Example TCP/IP server using the Sockets API that accepts connections from the auxiliary server.
TCPIP$TCP_SERVER_QIO.C TCPIP$TCP_CLIENT_QIO.C	Example TCP/IP client and server using QIO system services.

(continued on next page)

Table 1–4 (Cont.) TCP Programming Examples

File	Description
TCPIP$TCP_SERVER_QIO_AUXS.C	Example TCP/IP server using QIO system services that accepts connections from the auxiliary server.
TCPIP$TCP_CLIENT_QIO.MAR TCPIP$TCP_SERVER_QIO.MAR	Example TCP/IP client and server using QIO system services and the MACRO-32 programming language.

Table 1–5 UDP Programming Examples

File	Description
TCPIP$UDP_SERVER_SOCK.C TCPIP$UDP_CLIENT_SOCK.C	Example UDP/IP client and server using the Sockets API.
TCPIP$UDP_SERVER_QIO.C TCPIP$UDP_CLIENT_QIO.C	Example UDP/IP client and server using QIO system services.
TCPIP$UDP_CLIENT_QIO.MAR TCPIP$UDP_SERVER_QIO.MAR	Example UDP/IP client and server using QIO system services and the MACRO-32 programming language.

1.4 Compiling and Linking Compaq's C Language Programs

The examples in this manual were written using the Compaq C compiler. To compile and link your program, enter the following commands:

```
$ CC MAIN.C/PREFIX=ALL
$ LINK MAIN.OBJ
```

1.4.1 Compiling and Linking Programs Using BSD Version 4.4

To compile and link your Compaq C program using BSD Version 4.4, enter the following commands, where *filename* is the name of your program:

```
$ CC/PREFIX=ALL/DEFINE=(_SOCKADDR_LEN) filename.C
```

```
$ LINK/MAP filename
```

Instead of using the /DEFINE=(_SOCKADDR_LEN) option to the compile command, you can change your code to include the following #DEFINE preprocessor directive:

```
#DEFINE   _SOCKADDR_LEN    1
```

This statement must appear before you include any of the following header files:

```
#include <in.h>
#include <netdb.h>
#include <inet.h>
```

1.4.2 Compaq C Compilation Warnings

Certain parameters to the TCP/IP Services Sockets API functions require typecasting to avoid Compaq C compilation warnings. Typecasting is required because of parameter prototyping, which the Compaq C header (*filename*.H) files have in order to comply with ANSI standards. The Compaq *Tru64* UNIX header files have different requirements because their Sockets API functions are not parameter prototyped.

1.5 Using 64-Bit Addresses (Alpha Only)

For applications that run on OpenVMS Alpha systems, input and output (I/O) operations can be performed directly to and from the P2 or S2 addressable space by means of the 64-bit friendly $QIO and $QIOW system services.

To write data to a remote host, use the $QIO(IO$_WRITEVBLK) function with either the **p1** (input buffer) or **p5** (input buffer list) parameter. The address you specify for the parameter can be a 64-bit value.

To read data from a remote host, use the $QIO(IO$_READVBLK) function with either the **p1** (output buffer) or **p6** (output buffer list) parameter. The address you specify for the parameter can be a 64-bit value.

MACRO-32 does not provide 64-bit macros for system services. For more information about MACRO-32 programming support and for 64-bit addressing in general, see the *OpenVMS Alpha Guide to 64-Bit Addressing and VLM Features*.

For more information about using the $QIO and $QIOW system services for 64-bit addressing, see Chapter 5 and Chapter 6.

2
Writing Network Applications

You can use either the Sockets API or OpenVMS system services to write TCP/IP applications that run on your corporate network. These applications consist of a series of system calls that perform tasks, such as creating a socket, performing host and IP address lookups, accepting and closing connections, and setting socket options. These system calls are direct entry points that client and server processes use to obtain services from the TCP/IP kernel software. System calls look and behave exactly like other procedural calls in that they take arguments and return one or more results, including a status value. These arguments can contain values or pointers to objects in the application program.

This chapter describes the communication process followed by client and server applications. This process reflects the sequence of system calls within the client and server programs (see Tables 2–1 through 2–4). The chapter also includes Sockets API and OpenVMS system services examples for each step in the communication process.

2.1 The Client/Server Communication Process

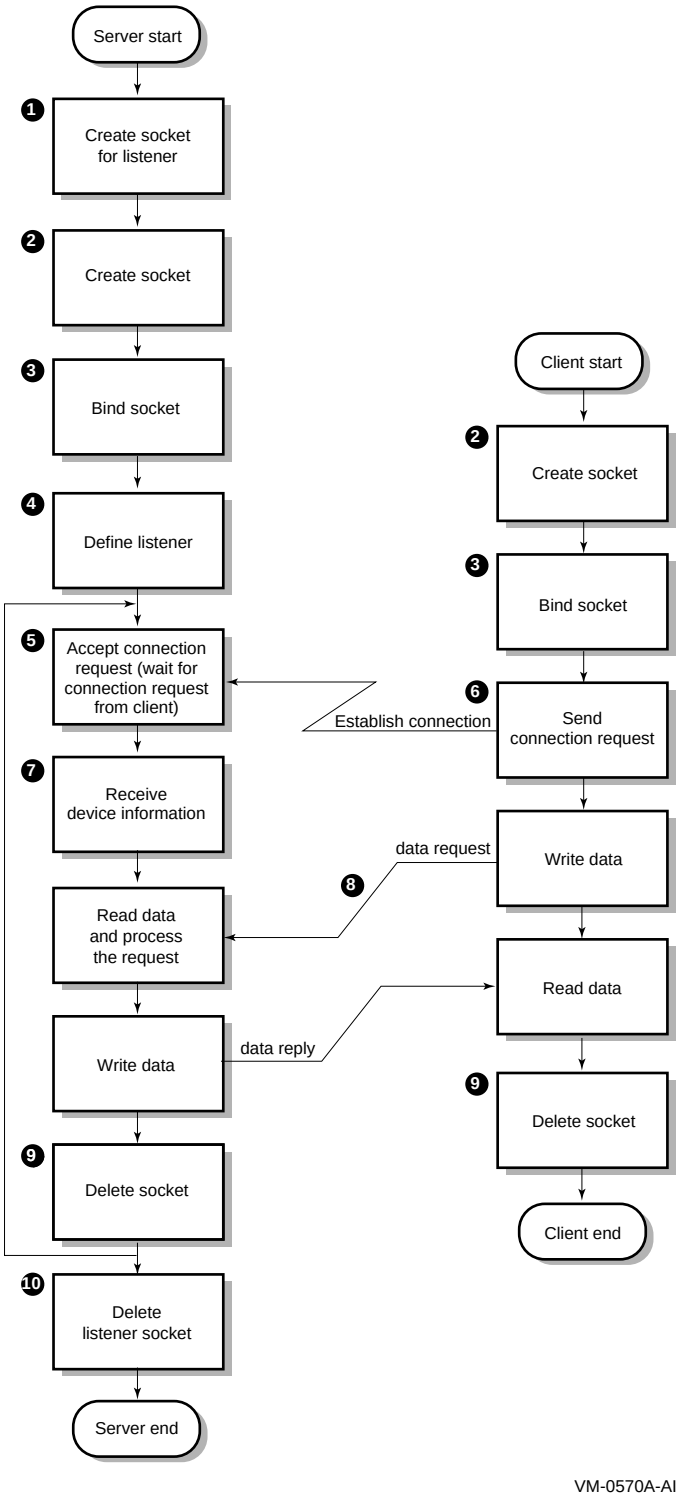
The most commonly used paradigm in constructing distributed applications is the client/server model. The requester, known as the client, sends a request to a server and waits for a response. The server is an application-level program that offers a service that can be reached over the network. Servers accept requests that arrive over the network, perform their service, and return the result to the client.

In addition to having a client and a server, the connection also has a mode of communication. This variable can be either connection-oriented or connectionless. When writing network applications, the developer uses the mode of communication required by the application-level service. Therefore, if the application-level service uses the connection-oriented mode of communication, the developer uses the virtual circuit or the Transmission Control Protocol (TCP) approach. If the application-level service uses the connectionless mode of communication, then the developer uses the datagram or the User Datagram Protocol (UDP) approach.

2.1.1 Using the TCP Protocol

Figure 2–1 shows the communication process for a TCP client/server application.

Figure 2–1 Client/Server Communication Process Using TCP



In this figure:

1 Server issues a call to create a listening socket.

2 Server and client create a socket.

3 Server and client bind socket. (This step is optional for a client.)

4 Server converts an unconnected socket into a passive socket (LISTEN state).

5 Server issues an `accept( )` and process blocks waiting for a connection request.

6 Client sends a connection request.

7 Server accepts the connection; a new socket is created for communication with this client.

8 Server receives device information from the local host.

9 Data exchange takes place.

10 Client and server delete the socket.

11 Server deletes the listener socket when the service to the client is terminating.

For server applications that use the TCP protocol, Table 2–1 identifies the typical communication tasks, the applicable Sockets API function, and the equivalent OpenVMS system service.

Table 2–1 TCP Server Tasks and Related Functions

Task	Sockets API Function	OpenVMS System Service Function
Create a socket	`socket( )`	$ASSIGN $QIO(IO$_SETMODE)[1]
Bind socket name	`bind( )`	$QIO(IO$_SETMODE)[1]
Define listener socket	`listen( )`	$QIO(IO$_SETMODE)[1]
Accept connection request	`accept( )`	$QIO(IO$_ACCESS | IO$M_ ACCEPT)

Exchange data	`read( )`
	`recv( )`
	`recvmsg( )`
		$QIO(IO$_READVBLK)

	`write( )`
	`send( )`
	`sendmsg( )`
		$QIO(IO$_WRITEVBLK)

Shut down the socket (optional)	`shutdown( )`	$QIO(IO$_DEACCESS | IO$M_ SHUTDOWN)
Close and delete the socket	`close( )`	$QIO(IO$_DEACCESS) $DASSGN

[1]The $QIO system service calls for creating a socket, binding a socket name, and defining a network pseudodevice as a listener are listed as three separate calls in this table. You can perform all three steps with one $QIO(IO$_SETMODE) call.

For a client application using the TCP protocol, Table 2–2 shows the tasks in the communication process, the applicable Sockets API functions, and the equivalent OpenVMS system service.

Table 2–2 TCP Client Calling Sequence and Related Functions

Task	Sockets API Function	OpenVMS System Service Function
Create a socket	`socket( )`	$ASSIGN $QIO(IO$_SETMODE)[1]
Bind socket name	`bind( )`	$QIO(IO$_SETMODE)[1]
Connect to server	`connect( )`	$QIO(IO$_ACCESS)

Exchange data	`read( )`
	`recv( )`
	`recvmsg( )`
		$QIO(IO$_READVBLK)

	`write( )`
	`send( )`
	`sendmsg( )`
		$QIO(IO$_WRITEVBLK)

Shut down the socket (optional)	`shutdown( )`	$QIO(IO$_DEACCESS | IO$M_SHUTDOWN)
Close and delete the socket	`close( )`	$QIO(IO$_DEACCESS) $DASSGN

[1]The $QIO system service calls for creating a socket and binding a socket name are listed as two separate calls in this table. You can perform both steps with one $QIO(IO$_SETMODE) call.

2.1.2 Using the UDP Protocol

Figure 2–2 shows the steps in the communication process for a client/server application using the UDP protocol.

In this figure:

1. Server and client create a socket.
2. Server and client bind the socket name. (This step is optional for a client.)
3. Data exchange takes place.
4. Server and client delete the socket.

For server applications using the UDP protocol, Table 2–3 identifies the tasks in the communication process, the Sockets API functions, and the equivalent OpenVMS system service function.

Figure 2–2 UDP Socket Communication Process

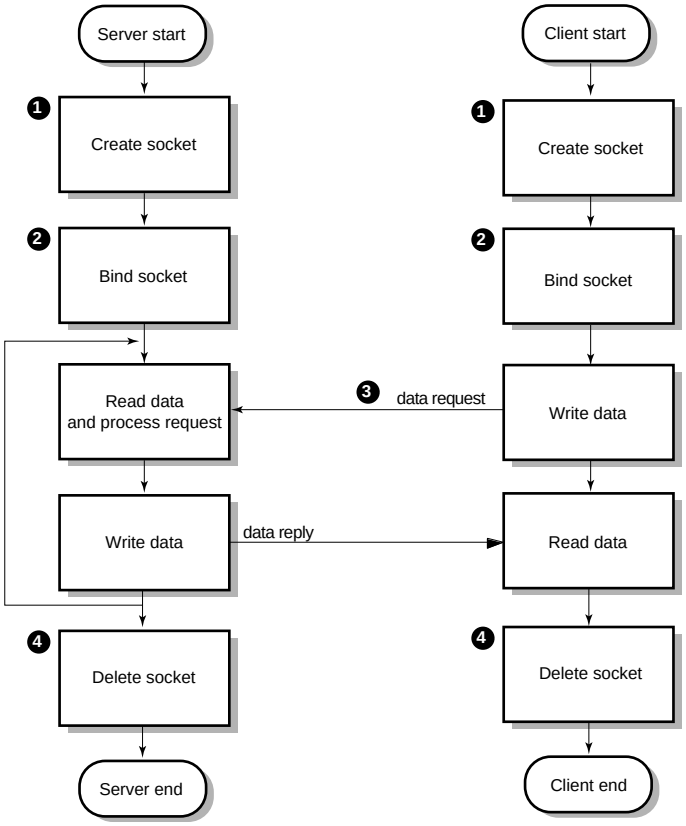


Table 2–3 UDP Server Tasks and Related Functions

Task	Sockets API Function	OpenVMS System Service Function
Create a socket	socket( )	$ASSIGN $QIO(IO$_SETMODE)[1]
Bind socket name	bind( )	$QIO(IO$_SETMODE)[1]
Exchange data	read( ) recv( ) recvfrom( ) recvmsg( )	$QIO(IO$_READVBLK)
	write( ) send( ) sendto( ) sendmsg( )	$QIO(IO$_WRITEVBLK)
Shut down the socket (optional)	shutdown( )	$QIO(IO$_DEACCESS | IO$M_ SHUTDOWN)
Close and delete the socket	close( )	$QIO(IO$_DEACCESS) $DASSGN

[1]The $QIO system service calls for creating a socket and binding a socket name are listed as two separate calls in this table. You can perform both steps with one $QIO(IO$_SETMODE) call.

For client applications using the UDP protocol, Table 2–4 describes the tasks in the communication process, the Sockets API function, and the equivalent OpenVMS system service.

Table 2–4 UDP Client Tasks and Related Functions

Task	Sockets API Function	OpenVMS System Service Function
Create a socket	socket( )	$ASSIGN $QIO(IO$_SETMODE)[1]
Bind socket name (optional)	bind( )	$QIO(IO$_SETMODE)[1]
Specify a destination address for outgoing datagrams	connect( )	$QIO(IO$_ACCESS)
Exchange data	read( ) recv( ) recvfrom( ) recvmsg( )	$QIO(IO$_READVBLK)
	write( ) send( ) sendto( ) sendmsg( )	$QIO(IO$_WRITEVBLK)
Shut down the socket (optional)	shutdown( )	$QIO(IO$_DEACCESS | IO$M_ SHUTDOWN)

[1]The $QIO system service calls for creating a socket and binding a socket name are listed as two separate calls in this table. You can perform both of these steps with one $QIO(IO$_SETMODE) call.

(continued on next page)

Table 2–4 (Cont.) UDP Client Tasks and Related Functions

Task	Sockets API Function	OpenVMS System Service Function
Close and delete the socket	`close( )`	$QIO(IO$_DEACCESS) $DASSGN

2.2 Creating a Socket

For network communication to take place between two processes, each process requires an end point to establish a communication link between the two processes. This end point, called a **socket**, sends messages to and receives messages from the socket associated with the process at the other end of the communication link.

Sockets are created by issuing a call to the `socket( )` function (Sockets API) or by the $ASSIGN and $QIO(IO$_SETMODE) routines (system service) specifying an address family, a protocol family, and a socket type.

If the socket creation is successful, the operation returns a small nonnegative integer value called a **socket descriptor**, or **sockfd**. From this point on, the application program uses the socket descriptor to reference the newly created socket.

In the TCP/IP Services implementation, this socket is also referred to as a **device socket**. A device socket is the pairing of an OpenVMS network device and a BSD-style socket. A device socket can be created implicitly when using the Sockets API or explicitly when using OpenVMS system services.

To displaying information about a device socket, use the TCP/IP management command SHOW DEVICE_SOCKET.

Perform the following steps to create a socket:

1. Assign a channel to the network device.
2. Create a socket.

The functions of the TCP/IP protocols are performed as I/O functions of the network device. When using the TCP/IP Services software, TCPIP$DEVICE: is the logical name for network devices.

When a channel is assigned to the TCPIP$DEVICE template network device, TCP/IP Services creates a new pseudodevice with a unique unit number and returns a channel number to use in subsequent operation requests with that device.

When the auxiliary server creates your application server process in response to incoming network traffic for a service with the `LISTEN` flag, it creates a device socket for your application server process. For your application to receive the device socket, assign a channel to SYS$NET, which is the logical name of a network pseudodevice, and perform an appropriate $QIO(IO$_SETMODE) operation. For examples of how to do this, see Appendix E. For a discussion of the auxiliary server, see the *Compaq TCP/IP Services for OpenVMS Management* manual.

2.2.1 Creating Sockets (Sockets API)

When using the Sockets API, create the socket with a call to the socket( ) function. Example 2–1 shows how to use the socket( ) function to create a TCP socket.

Example 2–1 Creating a Socket (Sockets API)

```
#include <socket.h>                /* define BSD socket api               */
#include <stdio.h>                 /* define standard i/o functions       */
#include <stdlib.h>                /* define standard library functions   */

int main( void )
{
    int sockfd;

    /*
     * create a socket
     */
1
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }

    exit( EXIT_SUCCESS );
}
```

1 This line creates the socket with the following arguments:

- AF_INET is the address family that references the socket in IPv4 protocol format.
- SOCK_STREAM specifies that the socket type is stream (TCP).
- 0 specifies that the protocol type is IPPROTO_TCP (default).

2.2.2 Creating Sockets (System Services)

When you use OpenVMS system services, you make two calls to create the socket:

- $ASSIGN to assign a channel to the network device
- $QIO or $QIOW to create the socket

  The Queue I/O Request ($QIO) service completes asynchronously. It returns to the caller immediately after queuing the I/O request, without waiting for the I/O operation to complete.

  For synchronous completion, use the Queue I/O Request and Wait ($QIOW) service. The $QIOW service is identical to the $QIO service, except the $QIOW returns to the caller after the I/O operation completes.

When you make the $QIO or $QIOW call, use either the IO$_SETMODE or the IO$_SETCHAR I/O function. You generally create, bind, and set up sockets to listen with one $QIO call. The IO$_SETMODE and IO$_SETCHAR functions perform in an identical manner, where network software is concerned. However, you must have LOG_IO privilege to successfully use the IO$_SETMODE I/O function.

Example 2–2 shows how to use the $ASSIGN and $QIOW system services to create a TCP socket.

Example 2–2 Creating a Socket (System Services)

```
#include <descrip.h>                /* define OpenVMS descriptors          */
#include <efndef.h>                 /* define 'EFN$C_ENF' event flag       */
#include <iodef.h>                  /* define i/o function codes           */
#include <ssdef.h>                  /* define system service status codes  */
#include <starlet.h>                /* define system service calls         */
#include <stdio.h>                  /* define standard i/o functions       */
#include <stdlib.h>                 /* define standard library functions   */
#include <stsdef.h>                 /* define condition value fields       */
#include <tcpip$inetdef.h>          /* define tcp/ip network constants,    */
                                    /* structures, and functions           */

struct iosb
    {                                   /* i/o status block                */
    unsigned short status;              /* i/o completion status           */
    unsigned short bytcnt;              /* bytes transferred if read/write */
    void *details;                      /* address of buffer or parameter  */
    };

struct sockchar
    {                                   /* socket characteristics          */
    unsigned short prot;                /* protocol                        */
    unsigned char type;                 /* type                            */
    unsigned char af;                   /* address format                  */
    };

int main( void )
{
    struct iosb iosb;                   /* i/o status block                */
    unsigned int status;                /* system service return status    */
    unsigned short channel;             /* network device i/o channel      */
1   struct sockchar sockchar;           /* socket characteristics buffer   */
    $DESCRIPTOR( inet_device,           /* string descriptor with logical  */
                 "TCPIP$DEVICE:" );     /* name of network pseudodevice    */

    /*
     * initialize socket characteristics
     */
2
    sockchar.prot = TCPIP$C_TCP;
    sockchar.type = TCPIP$C_STREAM;
    sockchar.af   = TCPIP$C_AF_INET;

    /*
     * assign i/o channel to network device
     */
3
    status = sys$assign( &inet_device,      /* device name                 */
                         &channel,          /* i/o channel                 */
                         0,                 /* access mode                 */
                         0                  /* not used                    */
                       );

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to assign i/o channel\n" );
        exit( status );
        }
```

(continued on next page)

Example 2–2 (Cont.) Creating a Socket (System Services)

```
    /*
     * create a socket
     */
4
    status = sys$qiow( EFN$C_ENF,           /* event flag                    */
                       channel,             /* i/o channel                   */
                       IO$_SETMODE,         /* i/o function code             */
                       &iosb,               /* i/o status block              */
                       0,                   /* ast service routine           */
                       0,                   /* ast parameter                 */
                       &sockchar,           /* p1 - socket characteristics   */
                       0,                   /* p2                            */
                       0,                   /* p3                            */
                       0,                   /* p4                            */
                       0,                   /* p5                            */
                       0                    /* p6                            */
                     );
    if ( status & STS$M_SUCCESS )
        status = iosb.status;
    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to create socket\n" );
        exit( status );
        }
    exit( EXIT_SUCCESS );
}
```

1 Define a sockchar structure to contain the characteristics of the type of socket.

2 Initialize the sockchar structure with the address family, protocol family and type of socket.

3 Assign a channel to a network device using a string descriptor with the logical name of the network pseudodevice and a structure to receive the I/O channel.

4 Create the socket with a call to SYS$QIOW supplying the channel, the socket characteristics and specifying an IO$_SETMODE function code.

2.3 Binding a Socket (Optional for Clients)

Binding a socket associates a local protocol address (that is, a 32-bit IPv4 address and a 16-bit TCP or UDP port number) with a socket. To bind a socket, specify a local interface address and local port number for the socket.

With the TCP protocol, you can specify an IP address, a port number, both an IP address and port number or neither.

If the application is using the UDP protocol and needs to receive incoming multicast or broadcast datagrams destined for a specific UDP port, see Section 2.10 for information about specifying the SO_REUSEPORT option when binding the socket.

2.3.1 Binding a Socket (Sockets API)

Example 2–3 shows an example of a TCP application using the bind( ) function to bind a socket name.

Note

The process must have a system user identification code (UIC) and the SYSPRV, BYPASS, or OPER privilege to bind port numbers 1 to 1023.

Example 2–3 Binding a Socket (Sockets API)

```
#include <in.h>                     /* define internet related constants,   */
                                    /* functions, and structures            */
#include <socket.h>                 /* define BSD socket api                */
#include <stdio.h>                  /* define standard i/o functions        */
#include <stdlib.h>                 /* define standard library functions    */
#include <string.h>                 /* define string handling functions     */

#define PORTNUM     12345               /* server port number               */

int main( void )
{
    int sockfd;
    struct sockaddr_in addr;

    /*
     * initialize socket address structure
     */

    memset( &addr, 0, sizeof(addr) );
    addr.sin_family      = AF_INET;
    addr.sin_port        = htons( PORTNUM );
    addr.sin_addr.s_addr = INADDR_ANY;

    /*
     * create a socket
     */

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }

    /*
     * bind ip address and port number to socket
     */

    if ( bind(sockfd,1  (struct sockaddr *) &addr,2  sizeof(addr)3 ) < 0 )
        {
        perror( "Failed to bind socket" );
        exit( EXIT_FAILURE );
        }

    exit( EXIT_SUCCESS );
}
```

In this example, the bind( ) function:

1. sockfd specifies the socket descriptor previously created with a call to the socket( ) function.
2. addr specifies the address of the sockaddr_in structure that assigns a name to the socket.
3. sizeof(addr) specifies the size of the sockaddr_in structure.

2.3.2 Binding a Socket (System Services)

Use the IO$_SETMODE or IO$_SETCHAR function of the $QIO system service to bind a socket.

Note

The process must have a system user identification code (UIC), SYSPRV, BYPASS, or OPER privileges to bind port numbers 1 to 1023.

Example 2–4 shows a TCP server using the IO$_SETMODE function to bind sockets.

Example 2–4 Binding a Socket (System Services)

```
#include <descrip.h>              /* define OpenVMS descriptors          */
#include <efndef.h>               /* define 'EFN$C_ENF' event flag       */
#include <in.h>                   /* define internet related constants,  */
                                  /* functions, and structures           */
#include <iodef.h>                /* define i/o function codes           */
#include <ssdef.h>                /* define system service status codes  */
#include <starlet.h>              /* define system service calls         */
#include <stdio.h>                /* define standard i/o functions       */
#include <stdlib.h>               /* define standard library functions   */
#include <string.h>               /* define string handling functions    */
#include <stsdef.h>               /* define condition value fields       */
#include <tcpip$inetdef.h>        /* define tcp/ip network constants,    */
                                  /* structures, and functions           */

#define PORTNUM     12345             /* server port number              */

struct iosb
    {                                 /* i/o status block                */
    unsigned short status;            /* i/o completion status           */
    unsigned short bytcnt;            /* bytes transferred if read/write */
    void *details;                    /* address of buffer or parameter  */
    };

struct itemlst_2
    {                                 /* item-list 2 descriptor/element  */
    unsigned short length;            /* length                          */
    unsigned short type;              /* parameter type                  */
    void *address;                    /* address of item list            */
    };
```

(continued on next page)

Example 2–4 (Cont.) Binding a Socket (System Services)

```
struct sockchar
    {                                   /* socket characteristics          */
    unsigned short prot;                /* protocol                        */
    unsigned char type;                 /* type                            */
    unsigned char af;                   /* address format                  */
    };

int main( void )
{
    struct iosb iosb;                   /* i/o status block                */
    unsigned int status;                /* system service return status    */
    unsigned short channel;             /* network device i/o channel      */
    struct sockchar sockchar;           /* socket characteristics buffer   */
    struct sockaddr_in addr;            /* socket address structure        */
    struct itemlst_2 addr_itemlst;      /* socket address item-list        */
    $DESCRIPTOR( inet_device,           /* string descriptor with logical  */
                 "TCPIP$DEVICE:" );     /* name of network pseudodevice    */

    /*
     * initialize socket characteristics
     */

    sockchar.prot = TCPIP$C_TCP;
    sockchar.type = TCPIP$C_STREAM;
    sockchar.af   = TCPIP$C_AF_INET;

    /*
     * initialize socket address item-list descriptor
     */

    addr_itemlst.length  = sizeof( addr );
    addr_itemlst.type    = TCPIP$C_SOCK_NAME;
    addr_itemlst.address = &addr;

    /*
     * initialize socket address structure
     */

    memset( &addr, 0, sizeof(addr) );
    addr.sin_family      = TCPIP$C_AF_INET;
    addr.sin_port        = htons( PORTNUM );
    addr.sin_addr.s_addr = TCPIP$C_INADDR_ANY;

    /*
     * assign i/o channel to network device
     */

    status = sys$assign( &inet_device,      /* device name                 */
                         &channel,          /* i/o channel                 */
                         0,                 /* access mode                 */
                         0                  /* not used                    */
                       );

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to assign i/o channel\n" );
        exit( status );
        }

    /*
     * create a socket
     */
```

(continued on next page)

Example 2–4 (Cont.) Binding a Socket (System Services)

```
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       channel,             /* i/o channel                  */
                       IO$_SETMODE,         /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       &sockchar,           /* p1 - socket characteristics  */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to create socket\n" );
        exit( status );
        }

    /*
     * bind ip address and port number to socket
     */

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       channel,             /* i/o channel                  */
                       IO$_SETMODE,         /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       &addr_itemlst,       /* p3 - local socket name       */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to bind socket\n" );
        exit( status );
        }

    exit( EXIT_SUCCESS );
}
```

2.4 Making a Socket a Listener (TCP Protocol)

Only server programs that use the TCP protocol need to set a socket to be a listener. This allows the program to receive incoming connection requests. As a connection-oriented protocol, TCP requires a connection; UDP, a connectionless protocol, does not.

The listen( ) function:

- Converts the unconnected socket into a passive socket.
- Changes the state of the socket to LISTEN.
- Remains open for the life of the server.
- Tells the kernel to accept incoming connections directed to this socket.

2.4.1 Setting a Socket to Listen (Sockets API)

Example 2–5 shows how a TCP server uses the listen( ) function to set a socket to listen for connection requests and to specify the number of incoming requests that can wait to be queued for processing.

Example 2–5 Setting a Socket to Listen (Sockets API)

```
#include <in.h>                    /* define internet related constants,   */
                                   /* functions, and structures            */
#include <socket.h>                /* define BSD socket api                */
#include <stdio.h>                 /* define standard i/o functions        */
#include <stdlib.h>                /* define standard library functions    */
#include <string.h>                /* define string handling functions     */

#define BACKLOG     1                  /* server backlog                   */
#define PORTNUM     12345              /* server port number               */

int main( void )
{
    int sockfd;
    struct sockaddr_in addr;

    /*
     * initialize socket address structure
     */

    memset( &addr, 0, sizeof(addr) );
    addr.sin_family      = AF_INET;
    addr.sin_port        = htons( PORTNUM );
    addr.sin_addr.s_addr = INADDR_ANY;

    /*
     * create a socket
     */

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }

    /*
     * bind ip address and port number to socket
     */

    if ( bind(sockfd, (struct sockaddr *) &addr, sizeof(addr)) < 0 )
        {
        perror( "Failed to bind socket" );
        exit( EXIT_FAILURE );
        }
```

(continued on next page)

Example 2–5 (Cont.) Setting a Socket to Listen (Sockets API)

```
    /*
     * set socket as a listen socket
     */
    if ( listen(sockfd, 1  BACKLOG 2  ) < 0 )
        {
        perror( "Failed to set socket passive" );
        exit( EXIT_FAILURE );
        }
    exit( EXIT_SUCCESS );
}
```

In this example of a listen( ) function:

1 sockfd is the socket descriptor previously defined by a call to the socket( ) function.

2 BACKLOG specifies that only one pending connection can be queued at any given time. The maximum number of connections is specified by the system configuration variable somaxconn. The default value for somaxconn is 1024. Refer to the *Compaq TCP/IP Services for OpenVMS Tuning and Troubleshooting* manual for how to display and change the somaxconn value dynamically.

2.4.2 Setting a Socket to Listen (System Services)

Example 2–6 shows how to use the IO$_SETMODE function to set the socket to listen for requests.

Example 2–6 Setting a Socket to Listen (System Services)

```
#include <descrip.h>              /* define OpenVMS descriptors          */
#include <efndef.h>               /* define 'EFN$C_ENF' event flag       */
#include <in.h>                   /* define internet related constants,  */
                                  /* functions, and structures           */
#include <iodef.h>                /* define i/o function codes           */
#include <ssdef.h>                /* define system service status codes  */
#include <starlet.h>              /* define system service calls         */
#include <stdio.h>                /* define standard i/o functions       */
#include <stdlib.h>               /* define standard library functions   */
#include <string.h>               /* define string handling functions    */
#include <stsdef.h>               /* define condition value fields       */
#include <tcpip$inetdef.h>        /* define tcp/ip network constants,    */
                                  /* structures, and functions           */


#define BACKLOG     1                 /* server backlog                  */
#define PORTNUM     12345             /* server port number              */

struct iosb
    {                                 /* i/o status block                */
    unsigned short status;            /* i/o completion status           */
    unsigned short bytcnt;            /* bytes transferred if read/write */
    void *details;                    /* address of buffer or parameter  */
    };
```

(continued on next page)

Example 2–6 (Cont.) Setting a Socket to Listen (System Services)

```
struct itemlst_2
    {                                   /* item-list 2 descriptor/element   */
    unsigned short length;              /* length                           */
    unsigned short type;                /* parameter type                   */
    void *address;                      /* address of item list             */
    };

struct sockchar
    {                                   /* socket characteristics           */
    unsigned short prot;                /* protocol                         */
    unsigned char type;                 /* type                             */
    unsigned char af;                   /* address format                   */
    };

int main( void )
{
    struct iosb iosb;                   /* i/o status block                 */
    unsigned int status;                /* system service return status     */
    unsigned short channel;             /* network device i/o channel       */
    struct sockchar sockchar;           /* socket characteristics buffer    */
    struct sockaddr_in addr;            /* socket address structure         */
    struct itemlst_2 addr_itemlst;      /* socket address item-list         */
    $DESCRIPTOR( inet_device,           /* string descriptor with logical   */
                 "TCPIP$DEVICE:" );     /* name of network pseudodevice     */

    /*
     * initialize socket characteristics
     */

    sockchar.prot = TCPIP$C_TCP;
    sockchar.type = TCPIP$C_STREAM;
    sockchar.af   = TCPIP$C_AF_INET;

    /*
     * initialize socket address item-list descriptor
     */

    addr_itemlst.length  = sizeof( addr );
    addr_itemlst.type    = TCPIP$C_SOCK_NAME;
    addr_itemlst.address = &addr;

    /*
     * initialize socket address structure
     */

    memset( &addr, 0, sizeof(addr) );
    addr.sin_family      = TCPIP$C_AF_INET;
    addr.sin_port        = htons( PORTNUM );
    addr.sin_addr.s_addr = TCPIP$C_INADDR_ANY;

    /*
     * assign i/o channel to network device
     */

    status = sys$assign( &inet_device,      /* device name                  */
                         &channel,          /* i/o channel                  */
                         0,                 /* access mode                  */
                         0                  /* not used                     */
                       );
```

(continued on next page)

Example 2–6 (Cont.) Setting a Socket to Listen (System Services)

```
if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to assign i/o channel\n" );
    exit( status );
    }

/*
 * create a socket
 */

status = sys$qiow( EFN$C_ENF,           /* event flag                    */
                   channel,             /* i/o channel                   */
                   IO$_SETMODE,         /* i/o function code             */
                   &iosb,               /* i/o status block              */
                   0,                   /* ast service routine           */
                   0,                   /* ast parameter                 */
                   &sockchar,           /* p1 - socket characteristics   */
                   0,                   /* p2                            */
                   0,                   /* p3                            */
                   0,                   /* p4                            */
                   0,                   /* p5                            */
                   0                    /* p6                            */
                 );

if ( status & STS$M_SUCCESS )
    status = iosb.status;

if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to create socket\n" );
    exit( status );
    }

/*
 * bind ip address and port number to socket
 */

status = sys$qiow( EFN$C_ENF,           /* event flag                    */
                   channel,             /* i/o channel                   */
                   IO$_SETMODE,         /* i/o function code             */
                   &iosb,               /* i/o status block              */
                   0,                   /* ast service routine           */
                   0,                   /* ast parameter                 */
                   0,                   /* p1                            */
                   0,                   /* p2                            */
                   &addr_itemlst,       /* p3 - local socket name        */
                   0,                   /* p4                            */
                   0,                   /* p5                            */
                   0                    /* p6                            */
                 );

if ( status & STS$M_SUCCESS )
    status = iosb.status;

if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to bind socket\n" );
    exit( status );
    }

/*
 * set socket as a listen socket
 */
```

(continued on next page)

Example 2–6 (Cont.) Setting a Socket to Listen (System Services)

```
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       channel,             /* i/o channel                  */
                       IO$_SETMODE,         /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       BACKLOG,             /* p4 - connection backlog      */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to set socket passive\n" );
        exit( status );
        }

    exit( EXIT_SUCCESS );
}
```

Note

Although you can use separate $QIO calls for socket create, bind, and listen operations, you can also perform these operations with one $QIO call.

2.5 Initiating a Connection (TCP Protocol)

A TCP client establishes a connection with a TCP server by issuing the `connect( )` function. The `connect( )` function initiates a three-way handshake between the client and the server. This must be successful to establish the connection.

2.5.1 Initiating a Connection (Sockets API)

To initiate a connection to a a TCP server, use the `connect( )` function. Example 2–7 shows a TCP client using the `connect( )` function to initiate a connection to a TCP server.

Example 2–7 Initiating a Connection (Sockets API)

```
#include <in.h>                    /* define internet related constants,   */
                                   /* functions, and structures            */
#include <inet.h>                  /* define network address info          */
#include <netdb.h>                 /* define network database library info */
#include <socket.h>                /* define BSD socket api                */
#include <stdio.h>                 /* define standard i/o functions        */
#include <stdlib.h>                /* define standard library functions    */
#include <string.h>                /* define string handling functions     */

#define BUFSZ       1024               /* user input buffer size           */
#define PORTNUM     12345              /* server port number               */

void get_servaddr( void *addrptr )
{
    char buf[BUFSIZ];
    struct in_addr val;
    struct hostent *host;

    while ( TRUE )
        {
        printf( "Enter remote host: " );

        if ( fgets(buf, sizeof(buf), stdin) == NULL )
            {
            printf( "Failed to read user input\n" );
            exit( EXIT_FAILURE );
            }

        buf[strlen(buf)-1] = 0;

        val.s_addr = inet_addr( buf );

        if ( val.s_addr != INADDR_NONE )
            {
            memcpy( addrptr, &val, sizeof(struct in_addr) );
            break;
            }

        if ( (host = gethostbyname(buf)) )
            {
            memcpy( addrptr, host->h_addr, sizeof(struct in_addr) );
            break;
            }
        }
}

int main( void )
{
    int sockfd;
    struct sockaddr_in addr;

    /*
     * initialize socket address structure
     */

    memset( &addr, 0, sizeof(addr) );
    addr.sin_family = AF_INET;
    addr.sin_port   = htons( PORTNUM );
    get_servaddr( &addr.sin_addr );

    /*
     * create a socket
     */
```

(continued on next page)

Example 2–7 (Cont.) Initiating a Connection (Sockets API)

```
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }

    /*
     * connect to specified host and port number
     */

    printf( "Initiated connection to host: %s, port: %d\n",
            inet_ntoa(addr.sin_addr), ntohs(addr.sin_port)
          );

    if ( connect(sockfd, (struct sockaddr *) &addr, sizeof(addr)) < 0 )
        {
        perror( "Failed to connect to server" );
        exit( EXIT_FAILURE );
        }

    exit( EXIT_SUCCESS );
}
```

2.5.2 Initiating a Connection (System Services)

To initiate a connection to a TCP server, use the $QIO system service with the IO$_ACCESS function and the **p3** argument. The **p3** argument of the IO$_ACCESS function is the address of an item_list_2 descriptor that points to the remote socket name.

Example 2–8 shows a TCP client using the IO_$ACCESS function to initiate a connection.

Example 2–8 Initiating a Connection (System Services)

```
#include <descrip.h>              /* define OpenVMS descriptors           */
#include <efndef.h>               /* define 'EFN$C_ENF' event flag        */
#include <in.h>                   /* define internet related constants,   */
                                  /* functions, and structures            */
#include <inet.h>                 /* define network address info          */
#include <iodef.h>                /* define i/o function codes            */
#include <netdb.h>                /* define network database library info */
#include <ssdef.h>                /* define system service status codes   */
#include <starlet.h>              /* define system service calls          */
#include <stdio.h>                /* define standard i/o functions        */
#include <stdlib.h>               /* define standard library functions    */
#include <string.h>               /* define string handling functions     */
#include <stsdef.h>               /* define condition value fields        */
#include <tcpip$inetdef.h>        /* define tcp/ip network constants,     */

                                  /* structures, and functions            */

#define BUFSZ       1024              /* user input buffer size           */
#define PORTNUM     12345             /* server port number               */
```

(continued on next page)

Example 2–8 (Cont.) Initiating a Connection (System Services)

```
struct iosb
    {                                   /* i/o status block                */
    unsigned short status;              /* i/o completion status           */
    unsigned short bytcnt;              /* bytes transferred if read/write  */
    void *details;                      /* address of buffer or parameter   */
    };

struct itemlst_2
    {                                   /* item-list 2 descriptor/element   */
    unsigned short length;              /* length                          */
    unsigned short type;                /* parameter type                  */
    void *address;                      /* address of item list            */
    };

struct sockchar
    {                                   /* socket characteristics          */
    unsigned short prot;                /* protocol                        */
    unsigned char type;                 /* type                            */
    unsigned char af;                   /* address format                  */
    };

void get_servaddr( void *addrptr )
{
    char buf[BUFSIZ];
    struct in_addr val;
    struct hostent *host;

    while ( TRUE )
        {
        printf( "Enter remote host: " );

        if ( fgets(buf, sizeof(buf), stdin) == NULL )
            {
            printf( "Failed to read user input\n" );
            exit( EXIT_FAILURE );
            }

        buf[strlen(buf)-1] = 0;

        val.s_addr = inet_addr( buf );

        if ( val.s_addr != INADDR_NONE )
            {
            memcpy( addrptr, &val, sizeof(struct in_addr) );
            break;
            }

        if ( (host = gethostbyname(buf)) )
            {
            memcpy( addrptr, host->h_addr, sizeof(struct in_addr) );
            break;
            }
        }
}
```

(continued on next page)

Example 2–8 (Cont.) Initiating a Connection (System Services)

```
int main( void )
{
    struct iosb iosb;                   /* i/o status block                */
    unsigned int status;                /* system service return status    */
    unsigned short channel;             /* network device i/o channel      */
    struct sockchar sockchar;           /* socket characteristics buffer   */
    struct sockaddr_in addr;            /* socket address structure        */
    struct itemlst_2 addr_itemlst;      /* socket address item-list        */
    $DESCRIPTOR( inet_device,           /* string descriptor with logical  */
                 "TCPIP$DEVICE:" );     /* name of network pseudodevice    */

    /*
     * initialize socket characteristics
     */

    sockchar.prot = TCPIP$C_TCP;
    sockchar.type = TCPIP$C_STREAM;
    sockchar.af   = TCPIP$C_AF_INET;

    /*
     * initialize socket address item-list descriptor
     */

    addr_itemlst.length  = sizeof( addr );
    addr_itemlst.type    = TCPIP$C_SOCK_NAME;
    addr_itemlst.address = &addr;

    /*
     * initialize socket address structure
     */

    memset( &addr, 0, sizeof(addr) );
    addr.sin_family = TCPIP$C_AF_INET;
    addr.sin_port   = htons( PORTNUM );
    get_servaddr( &addr.sin_addr );

    /*
     * assign i/o channel to network device
     */

    status = sys$assign( &inet_device,      /* device name                */
                         &channel,          /* i/o channel                */
                         0,                 /* access mode                */
                         0                  /* not used                   */
                       );

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to assign i/o channel\n" );
        exit( status );
        }

    /*
     * create a socket
     */
```

(continued on next page)

Example 2–8 (Cont.) Initiating a Connection (System Services)

```
    status = sys$qiow( EFN$C_ENF,           /* event flag                    */
                       channel,             /* i/o channel                   */
                       IO$_SETMODE,         /* i/o function code             */
                       &iosb,               /* i/o status block              */
                       0,                   /* ast service routine           */
                       0,                   /* ast parameter                 */
                       &sockchar,           /* p1 - socket characteristics   */
                       0,                   /* p2                            */
                       0,                   /* p3                            */
                       0,                   /* p4                            */
                       0,                   /* p5                            */
                       0                    /* p6                            */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to create socket\n" );
        exit( status );
        }

    /*
     * connect to specified host and port number
     */

    printf( "Initiated connection to host: %s, port: %d\n",
            inet_ntoa(addr.sin_addr), ntohs(addr.sin_port)
          );

    status = sys$qiow( EFN$C_ENF,           /* event flag                    */
                       channel,             /* i/o channel                   */
                       IO$_ACCESS,          /* i/o function code             */
                       &iosb,               /* i/o status block              */
                       0,                   /* ast service routine           */
                       0,                   /* ast parameter                 */
                       0,                   /* p1                            */
                       0,                   /* p2                            */
                       &addr_itemlst,       /* p3 - remote socket name       */
                       0,                   /* p4                            */
                       0,                   /* p5                            */
                       0                    /* p6                            */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to connect to server\n" );
        exit( status );
        }

    exit( EXIT_SUCCESS );
}
```

2.6 Accepting a Connection (TCP Protocol)

A TCP server program must be able to accept incoming connection requests from client programs. The accept( ) function:

- Returns the next completed connection from the completed connection queue.

- Return a new socket descriptor that is connected with the client, called the **connected socket**. There is one connected socket for each client connected to the server. The connected socket remains until the server is finished serving the client.

2.6.1 Accepting a Connection (Sockets API)

Example 2–9 shows how to use the accept( ) function.

Example 2–9 Accepting a Connection (Sockets API)

```
#include <in.h>                    /* define internet related constants,   */
                                   /* functions, and structures            */
#include <inet.h>                  /* define network address info          */
#include <netdb.h>                 /* define network database library info */
#include <socket.h>                /* define BSD socket api                */
#include <stdio.h>                 /* define standard i/o functions        */
#include <stdlib.h>                /* define standard library functions    */
#include <string.h>                /* define string handling functions     */

#define SERV_BACKLOG    1              /* server backlog                    */
#define SERV_PORTNUM    12345          /* server port number                */

int main( void )
{
    int conn_sockfd;                   /* connection socket descriptor     */
    int listen_sockfd;                 /* listen socket descriptor         */

    unsigned int cli_addrlen;          /* returned length of client socket */
                                       /* address structure                */
    struct sockaddr_in cli_addr;       /* client socket address structure  */
    struct sockaddr_in serv_addr;      /* server socket address structure  */

    /*
     * initialize client's socket address structure
     */

    memset( &cli_addr, 0, sizeof(cli_addr) );

    /*
     * initialize server's socket address structure
     */

    memset( &serv_addr, 0, sizeof(serv_addr) );
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_port        = htons( SERV_PORTNUM );
    serv_addr.sin_addr.s_addr = INADDR_ANY;

    /*
     * create a listen socket
     */

    if ( (listen_sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }
```

(continued on next page)

Example 2–9 (Cont.) Accepting a Connection (Sockets API)

```
    /*
     * bind server's ip address and port number to listen socket
     */

    if ( bind(listen_sockfd,
              (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0 )
        {
        perror( "Failed to bind socket" );
        exit( EXIT_FAILURE );
        }

    /*
     * set socket as a listen socket
     */

    if ( listen(listen_sockfd, SERV_BACKLOG) < 0 )
        {
        perror( "Failed to set socket passive" );
        exit( EXIT_FAILURE );
        }

    /*
     * accept connection from a client
     */

    printf( "Waiting for a client connection on port: %d\n",
            ntohs(serv_addr.sin_port)
          );

    cli_addrlen = sizeof(cli_addr);

    conn_sockfd = accept( listen_sockfd,     1
                          (struct sockaddr *) &cli_addr, 2
                          &cli_addrlen       3
                        );
    if ( conn_sockfd < 0 )
        {
        perror( "Failed to accept client connection" );
        exit( EXIT_FAILURE );
        }

    printf( "Accepted connection from host: %s, port: %d\n",
            inet_ntoa(cli_addr.sin_addr), ntohs(cli_addr.sin_port)
          );

    exit( EXIT_SUCCESS );
}
```

In this example of an `accept( )` function:

1 `listen_sockfd` is the socket descriptor returned by the previous call to the `socket( )` function. This socket is bound to an address with the `bind( )` function. The `listen( )` function changes the socket state from CLOSED to LISTEN (converts the unconnected socket to a passive socket).

2 `cli_addr` contains the protocol address of the client.

3 `cli_addrlen` is a value/result parameter that initially contains the size of the `cli_addr` structure. On return of the `accept( )` function, the `cli_addr` structure contains the actual length, in bytes, of the socket address structure returned by the kernel for the connected socket.

2.6.2 Accepting a Connection (System Services)

To accept a connection request, use the following procedure:

1. Use the $ASSIGN system service to create a channel for the new connection.
2. Use the $QIO system service using the IO$_ACCESS function with the IO$M_ACCEPT function modifier.

   The **p4** argument specifies the address of a word written with the channel number of the new connection. If **p3** specifies a valid output buffer, the $QIO service returns the remote socket name.

Note

Specifying the IO$_ACCESS function is mandatory for TCP/IP. The IO$_ACCESS function uses the **p4** argument only with the IO$M_ACCEPT modifier.

Example 2–10 shows a TCP server using the IO$_ACCESS function with the IO$M_ACCEPT function modifier to accept incoming connection requests.

Example 2–10 Accepting a Connection (System Services)

```
#include <descrip.h>                /* define OpenVMS descriptors           */
#include <efndef.h>                 /* define 'EFN$C_ENF' event flag        */
#include <in.h>                     /* define internet related constants,   */
                                    /* functions, and structures            */
#include <inet.h>                   /* define network address info          */
#include <iodef.h>                  /* define i/o function codes            */
#include <netdb.h>                  /* define network database library info */
#include <ssdef.h>                  /* define system service status codes   */
#include <starlet.h>                /* define system service calls          */
#include <stdio.h>                  /* define standard i/o functions        */
#include <stdlib.h>                 /* define standard library functions    */
#include <string.h>                 /* define string handling functions     */
#include <stsdef.h>                 /* define condition value fields        */
#include <tcpip$inetdef.h>          /* define tcp/ip network constants,     */
                                    /* structures, and functions            */

#define SERV_BACKLOG    1               /* server backlog                   */
#define SERV_PORTNUM    12345           /* server port number               */

struct iosb
    {                                   /* i/o status block                 */
    unsigned short status;              /* i/o completion status            */
    unsigned short bytcnt;              /* bytes transferred if read/write  */
    void *details;                      /* address of buffer or parameter   */
    };

struct itemlst_2
    {                                   /* item-list 2 descriptor/element   */
    unsigned short length;              /* length                           */
    unsigned short type;                /* parameter type                   */
    void *address;                      /* address of item list             */
    };
```

(continued on next page)

Example 2–10 (Cont.) Accepting a Connection (System Services)

```
struct itemlst_3
    {                                   /* item-list 3 descriptor/element   */
    unsigned short length;              /* length                           */
    unsigned short type;                /* parameter type                   */
    void *address;                      /* address of item list             */
    unsigned int *retlen;               /* address of returned length       */
    };

struct sockchar
    {                                   /* socket characteristics           */
    unsigned short prot;                /* protocol                         */
    unsigned char type;                 /* type                             */
    unsigned char af;                   /* address format                   */
    };


int main( void )
{
    struct iosb iosb;                   /* i/o status block                 */
    unsigned int status;                /* system service return status     */

    unsigned short conn_channel;        /* connect inet device i/o channel  */

    unsigned short listen_channel;      /* listen inet device i/o channel   */
    struct sockchar listen_sockchar;    /* listen socket characteristics    */

    unsigned int cli_addrlen;           /* returned length of client socket */
                                        /* address structure                */
    struct sockaddr_in cli_addr;        /* client socket address structure  */
    struct itemlst_3 cli_itemlst;       /* client socket address item-list  */

    struct sockaddr_in serv_addr;       /* server socket address structure  */
    struct itemlst_2 serv_itemlst;      /* server socket address item-list  */

    $DESCRIPTOR( inet_device,           /* string descriptor with logical   */
                 "TCPIP$DEVICE:" );     /* name of network pseudodevice     */
    /*
     * initialize socket characteristics
     */

    listen_sockchar.prot = TCPIP$C_TCP;
    listen_sockchar.type = TCPIP$C_STREAM;
    listen_sockchar.af   = TCPIP$C_AF_INET;

    /*
     * initialize client's item-list descriptor
     */

    memset( &cli_itemlst, 0, sizeof(cli_itemlst) );
    cli_itemlst.length  = sizeof( cli_addr );
    cli_itemlst.address = &cli_addr;
    cli_itemlst.retlen  = &cli_addrlen;

    /*
     * initialize client's socket address structure
     */

    memset( &cli_addr, 0, sizeof(cli_addr) );

    /*
     * initialize server's item-list descriptor
     */

    serv_itemlst.length  = sizeof( serv_addr );
    serv_itemlst.type    = TCPIP$C_SOCK_NAME;
    serv_itemlst.address = &serv_addr;
```

(continued on next page)

Example 2–10 (Cont.) Accepting a Connection (System Services)

```
/*
 * initialize server's socket address structure
 */

memset( &serv_addr, 0, sizeof(serv_addr) );
serv_addr.sin_family      = TCPIP$C_AF_INET;
serv_addr.sin_port        = htons( SERV_PORTNUM );
serv_addr.sin_addr.s_addr = TCPIP$C_INADDR_ANY;

/*
 * assign i/o channels to network device
 */

status = sys$assign( &inet_device,      /* device name                  */
                     &listen_channel,   /* i/o channel                  */
                     0,                 /* access mode                  */
                     0                  /* not used                     */
                   );

if ( status & STS$M_SUCCESS )
    status = sys$assign( &inet_device,  /* device name                  */
                         &conn_channel, /* i/o channel                  */
                         0,             /* access mode                  */
                         0              /* not used                     */
                       );
if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to assign i/o channel(s)\n" );
    exit( status );
    }

/*
 * create a listen socket
 */

status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                   listen_channel,      /* i/o channel                  */
                   IO$_SETMODE,         /* i/o function code            */
                   &iosb,               /* i/o status block             */
                   0,                   /* ast service routine          */
                   0,                   /* ast parameter                */
                   &listen_sockchar,    /* p1 - socket characteristics  */
                   0,                   /* p2                           */
                   0,                   /* p3                           */
                   0,                   /* p4                           */
                   0,                   /* p5                           */
                   0                    /* p6                           */
                 );

if ( status & STS$M_SUCCESS )
    status = iosb.status;

if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to create socket\n" );
    exit( status );
    }

/*
 * bind server's ip address and port number to listen socket
 */
```

(continued on next page)

Example 2–10 (Cont.) Accepting a Connection (System Services)

```
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       listen_channel,      /* i/o channel                  */
                       IO$_SETMODE,         /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       &serv_itemlst,       /* p3 - local socket name       */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;
    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to bind socket\n" );
        exit( status );
        }

    /*
     * set socket as a listen socket
     */

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       listen_channel,      /* i/o channel                  */
                       IO$_SETMODE,         /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       SERV_BACKLOG,        /* p4 - connection backlog      */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to set socket passive\n" );
        exit( status );
        }

    /*
     * accept connection from a client
     */

    printf( "Waiting for a client connection on port: %d\n",
            ntohs(serv_addr.sin_port)
          );
```

(continued on next page)

Example 2–10 (Cont.) Accepting a Connection (System Services)

```
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       listen_channel,      /* i/o channel                  */
                       IO$_ACCESS|IO$M_ACCEPT,
                                            /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       &cli_itemlst,        /* p3 - remote socket name      */
                       &conn_channel,       /* p4 - i/o channel for new     */
                                            /*      connection              */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );
    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to accept client connection\n" );
        exit( status );
        }

    printf( "Accepted connection from host: %s, port: %d\n",
            inet_ntoa(cli_addr.sin_addr), ntohs(cli_addr.sin_port)
          );

    exit( EXIT_SUCCESS );
}
```

2.7 Getting Socket Options

Obtaining socket information is useful if your program has management functions, or if you have a complex program that uses multiple connections you need to track.

2.7.1 Getting Socket Information (Sockets API)

You can use any of the following Sockets API functions to get socket information:

- `getpeername( )`
- `getsockname( )`
- `getsockopt( )`

Example 2–11 shows a TCP server using the getpeername( ) function to get the remote IP address and port number associated with a socket.

Example 2–11 Getting Socket Information (Sockets API)

```
#include <in.h>                    /* define internet related constants,   */
                                   /* functions, and structures            */
#include <inet.h>                  /* define network address info          */
#include <netdb.h>                 /* define network database library info */
#include <socket.h>                /* define BSD socket api                */
#include <stdio.h>                 /* define standard i/o functions        */
#include <stdlib.h>                /* define standard library functions    */
#include <string.h>                /* define string handling functions     */

#define SERV_BACKLOG    1              /* server backlog                    */
#define SERV_PORTNUM    12345          /* server port number                */

int main( void )
{
    int conn_sockfd;                   /* connection socket descriptor     */
    int listen_sockfd;                 /* listen socket descriptor         */

    unsigned int cli_addrlen;          /* returned length of client socket */
                                       /* address structure                */
    struct sockaddr_in cli_addr;       /* client socket address structure  */
    struct sockaddr_in serv_addr;      /* server socket address structure  */

    /*
     * initialize server's socket address structure
     */

    memset( &serv_addr, 0, sizeof(serv_addr) );
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_port        = htons( SERV_PORTNUM );
    serv_addr.sin_addr.s_addr = INADDR_ANY;

    /*
     * create a listen socket
     */

    if ( (listen_sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }
    /*
     * bind server's ip address and port number to listen socket
     */

    if ( bind(listen_sockfd,
              (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0 )
        {
        perror( "Failed to bind socket" );
        exit( EXIT_FAILURE );
        }

    /*
     * set socket as a listen socket
     */
```

(continued on next page)

Example 2–11 (Cont.) Getting Socket Information (Sockets API)

```
    if ( listen(listen_sockfd, SERV_BACKLOG) < 0 )
        {
        perror( "Failed to set socket passive" );
        exit( EXIT_FAILURE );
        }

    /*
     * accept connection from a client
     */
    printf( "Waiting for a client connection on port: %d\n",
            ntohs(serv_addr.sin_port)
          );

    conn_sockfd = accept( listen_sockfd, (struct sockaddr *) 0, 0 );

    if ( conn_sockfd < 0 )
        {
        perror( "Failed to accept client connection" );
        exit( EXIT_FAILURE );
        }

    /*
     * log client connection request
     */

    cli_addrlen = sizeof(cli_addr);
    memset( &cli_addr, 0, sizeof(cli_addr) );

    if ( getpeername(conn_sockfd 1 ,
                    (struct sockaddr * ) &cli_addr, 2  &cli_addrlen 3 ) < 0 )
        {
        perror( "Failed to get client name" );
        exit( EXIT_FAILURE );
        }

    printf( "Accepted connection from host: %s, port: %d\n",
            inet_ntoa(cli_addr.sin_addr), ntohs(cli_addr.sin_port) 4
          );

    exit( EXIT_SUCCESS );
}
```

1 `conn_sockaddr` is the socket descriptor returned by the previous call to the `accept( )` function.

2 `cli_addr` is the address structure for the connected socket.

3 `cli_addrlen` is the length of the address structure for the connected socket.

4 The `printf` statement accesses the information stored in the address structure for the connected socket and displays the client's IP address and port number. The `inet_ntoa ( )` and the `ntohs( )` functions are used to convert the IP address and port number from their network byte order to the host byte order.

2.7.2 Getting Socket Information (System Services)

To obtain information about the parts of a socket, use the $QIO system service with IO$_SENSEMODE to get socket information.

Example 2–12 shows a TCP client using the IO$_SENSEMODE function to get the client's IP address and port number.

Example 2–12 Getting Socket Information (System Services)

```
#include <descrip.h>                /* define OpenVMS descriptors           */
#include <efndef.h>                 /* define 'EFN$C_ENF' event flag        */
#include <in.h>                     /* define internet related constants,   */
<valid_break>
                                    /* functions, and structures            */
#include <inet.h>                   /* define network address info          */
#include <iodef.h>                  /* define i/o function codes            */
#include <netdb.h>                  /* define network database library info */
#include <ssdef.h>                  /* define system service status codes   */
#include <starlet.h>                /* define system service calls          */
#include <stdio.h>                  /* define standard i/o functions        */
#include <stdlib.h>                 /* define standard library functions    */
#include <string.h>                 /* define string handling functions     */
#include <stsdef.h>                 /* define condition value fields        */
#include <tcpip$inetdef.h>          /* define tcp/ip network constants,     */
                                    /* structures, and functions            */

#define SERV_BACKLOG    1               /* server backlog                   */
#define SERV_PORTNUM    12345           /* server port number               */

struct iosb
    {                                   /* i/o status block                 */
    unsigned short status;              /* i/o completion status            */
    unsigned short bytcnt;              /* bytes transferred if read/write  */
    void *details;                      /* address of buffer or parameter   */
    };

struct itemlst_2
    {                                   /* item-list 2 descriptor/element   */
    unsigned short length;              /* length                           */
    unsigned short type;                /* parameter type                   */
    void *address;                      /* address of item list             */
    };

struct itemlst_3
    {                                   /* item-list 3 descriptor/element   */
    unsigned short length;              /* length                           */
    unsigned short type;                /* parameter type                   */
    void *address;                      /* address of item list             */
    unsigned int *retlen;               /* address of returned length       */
    };

struct sockchar
    {                                   /* socket characteristics           */
    unsigned short prot;                /* protocol                         */
    unsigned char type;                 /* type                             */
    unsigned char af;                   /* address format                   */
    };

int main( void )
{
    struct iosb iosb;                   /* i/o status block                 */
    unsigned int status;                /* system service return status     */
```

(continued on next page)

Example 2–12 (Cont.) Getting Socket Information (System Services)

```
    unsigned short conn_channel;        /* connect inet device i/o channel  */

    unsigned short listen_channel;      /* listen inet device i/o channel   */
    struct sockchar listen_sockchar;    /* listen socket characteristics    */

    unsigned int cli_addrlen;           /* returned length of client socket */
                                        /* address structure                */
    struct sockaddr_in cli_addr;        /* client socket address structure  */
    struct itemlst_3 cli_itemlst;       /* client socket address item-list  */

    struct sockaddr_in serv_addr;       /* server socket address structure  */
    struct itemlst_2 serv_itemlst;      /* server socket address item-list  */

    $DESCRIPTOR( inet_device,           /* string descriptor with logical   */
                 "TCPIP$DEVICE:" );     /* name of network pseudodevice     */
    /*
     * initialize socket characteristics
     */

    listen_sockchar.prot = TCPIP$C_TCP;
    listen_sockchar.type = TCPIP$C_STREAM;
    listen_sockchar.af   = TCPIP$C_AF_INET;

    /*
     * initialize client's item-list descriptor
     */

    cli_itemlst.length  = sizeof( cli_addr );
    cli_itemlst.type    = TCPIP$C_SOCK_NAME;
    cli_itemlst.address = &cli_addr;
    cli_itemlst.retlen  = &cli_addrlen;

    /*
     * initialize server's item-list descriptor
     */

    serv_itemlst.length  = sizeof( serv_addr );
    serv_itemlst.type    = TCPIP$C_SOCK_NAME;
    serv_itemlst.address = &serv_addr;

    /*
     * initialize server's socket address structure
     */

    memset( &serv_addr, 0, sizeof(serv_addr) );
    serv_addr.sin_family      = TCPIP$C_AF_INET;
    serv_addr.sin_port        = htons( SERV_PORTNUM );
    serv_addr.sin_addr.s_addr = TCPIP$C_INADDR_ANY;

    /*
     * assign i/o channels to network device
     */

    status = sys$assign( &inet_device,      /* device name                  */
                         &listen_channel,   /* i/o channel                  */
                         0,                 /* access mode                  */
                         0                  /* not used                     */
                       );

    if ( status & STS$M_SUCCESS )
        status = sys$assign( &inet_device,  /* device name                  */
                             &conn_channel, /* i/o channel                  */
                             0,             /* access mode                  */
                             0              /* not used                     */
                           );
```

(continued on next page)

Example 2–12 (Cont.) Getting Socket Information (System Services)

```
if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to assign i/o channel(s)\n" );
    exit( status );
    }
/*
 * create a listen socket
 */
status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                   listen_channel,      /* i/o channel                  */
                   IO$_SETMODE,         /* i/o function code            */
                   &iosb,               /* i/o status block             */
                   0,                   /* ast service routine          */
                   0,                   /* ast parameter                */
                   &listen_sockchar,    /* p1 - socket characteristics  */
                   0,                   /* p2                           */
                   0,                   /* p3                           */
                   0,                   /* p4                           */
                   0,                   /* p5                           */
                   0                    /* p6                           */
                 );
if ( status & STS$M_SUCCESS )
    status = iosb.status;
if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to create socket\n" );
    exit( status );
    }
/*
 * bind server's ip address and port number to listen socket
 */
status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                   listen_channel,      /* i/o channel                  */
                   IO$_SETMODE,         /* i/o function code            */
                   &iosb,               /* i/o status block             */
                   0,                   /* ast service routine          */
                   0,                   /* ast parameter                */
                   0,                   /* p1                           */
                   0,                   /* p2                           */
                   &serv_itemlst,       /* p3 - local socket name       */
                   0,                   /* p4                           */
                   0,                   /* p5                           */
                   0                    /* p6                           */
                 );
if ( status & STS$M_SUCCESS )
    status = iosb.status;
if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to bind socket\n" );
    exit( status );
    }
```

(continued on next page)

Example 2–12 (Cont.) Getting Socket Information (System Services)

```
    /*
     * set socket as a listen socket
     */

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       listen_channel,      /* i/o channel                  */
                       IO$_SETMODE,         /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       SERV_BACKLOG,        /* p4 - connection backlog      */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to set socket passive\n" );
        exit( status );
        }

    /*
     * accept connection from a client
     */

    printf( "Waiting for a client connection on port: %d\n",
            ntohs(serv_addr.sin_port)
          );

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       listen_channel,      /* i/o channel                  */
                       IO$_ACCESS|IO$M_ACCEPT,
                                            /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       &conn_channel,       /* p4 - i/o channel for new     */
                                            /*      connection              */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;
```

(continued on next page)

Example 2–12 (Cont.) Getting Socket Information (System Services)

```
    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to accept client connection\n" );
        exit( status );
        }

    /*
     * log client connection request
     */

    memset( &cli_addr, 0, sizeof(cli_addr) );

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       conn_channel,        /* i/o channel                  */
                       IO$_SENSEMODE,       /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       &cli_itemlst,        /* p4 - peer socket name        */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to get client name\n" );
        exit( status );
        }

    printf( "Accepted connection from host: %s, port: %d\n",
            inet_ntoa(cli_addr.sin_addr), ntohs(cli_addr.sin_port)
          );

    exit( EXIT_SUCCESS );
}
```

2.8 Setting Socket Options

With TCP/IP Services, you use the setsockopt( ) function to set binary socket options and socket options that return a value. Calls to setsockopt( ) specifying unsupported options return an error code of ENOPROTOOPT.

2.8.1 Setting Socket Options (Sockets API)

Example 2–13 shows a TCP server using the setsockopt ( ) function to set the SO_REUSEADDR option.

Example 2–13 Setting Socket Options (Sockets API)

```
#include <in.h>                    /* define internet related constants,   */
                                   /* functions, and structures            */
#include <inet.h>                  /* define network address info          */
#include <netdb.h>                 /* define network database library info */
#include <socket.h>                /* define BSD socket api                */
#include <stdio.h>                 /* define standard i/o functions        */
#include <stdlib.h>                /* define standard library functions    */
#include <string.h>                /* define string handling functions     */

#define SERV_BACKLOG    1              /* server backlog                    */
#define SERV_PORTNUM    12345          /* server port number                */

int main( void )
{
    int optval = 1;                    /* SO_REUSEADDR's option value (on) */

    int conn_sockfd;                   /* connection socket descriptor     */
    int listen_sockfd;                 /* listen socket descriptor         */

    unsigned int cli_addrlen;          /* returned length of client socket */
                                       /* address structure                */
    struct sockaddr_in cli_addr;       /* client socket address structure  */
    struct sockaddr_in serv_addr;      /* server socket address structure  */

    /*
     * initialize server's socket address structure
     */

    memset( &serv_addr, 0, sizeof(serv_addr) );
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_port        = htons( SERV_PORTNUM );
    serv_addr.sin_addr.s_addr = INADDR_ANY;

    /*
     * create a listen socket
     */

    if ( (listen_sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }

    /*
     * bind server's ip address and port number to listen socket
     */

    if ( setsockopt(listen_sockfd,  1
         SOL_SOCKET 2 , SO_REUSEADDR 3 , &optval 4 , sizeof(optval) 5 ) < 0 )
        {
        perror( "Failed to set socket option" );
        exit( EXIT_FAILURE );
        }

    if ( bind(listen_sockfd,
              (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0 )
        {
        perror( "Failed to bind socket" );
        exit( EXIT_FAILURE );
        }

    /*
     * set socket as a listen socket
     */
```

(continued on next page)

Example 2–13 (Cont.) Setting Socket Options (Sockets API)

```
    if ( listen(listen_sockfd, SERV_BACKLOG) < 0 )
        {
        perror( "Failed to set socket passive" );
        exit( EXIT_FAILURE );
        }

    /*
     * accept connection from a client
     */

    printf( "Waiting for a client connection on port: %d\n",
            ntohs(serv_addr.sin_port)
          );

    conn_sockfd = accept( listen_sockfd, (struct sockaddr *) 0, 0 );

    if ( conn_sockfd < 0 )
        {
        perror( "Failed to accept client connection" );
        exit( EXIT_FAILURE );
        }

    /*
     * log client connection request
     */

    cli_addrlen = sizeof(cli_addr);
    memset( &cli_addr, 0, sizeof(cli_addr) );

    if ( getpeername(conn_sockfd,
                    (struct sockaddr *) &cli_addr, &cli_addrlen) < 0 )
        {
        perror( "Failed to get client name" );
        exit( EXIT_FAILURE );
        }

    printf( "Accepted connection from host: %s, port: %d\n",
            inet_ntoa(cli_addr.sin_addr), ntohs(cli_addr.sin_port)
          );

    exit( EXIT_SUCCESS );
}
```

1 `listen_sockfd` refers to an open socket descriptor returned by the previous call to the `socket ( )` function.

2 `SOL_SOCKET` specifies the code in the system to interpret the option. In this case, the level is the general socket code.

3 `SO_REUSEADDR` is the socket option to be set. In this case, the socket option allows reuse of local addresses.

4 `optval` is the value to set for the option. In this case, the value is 1, which enables the option.

5 `sizeof(optval)` is the size of the option value.

2.8.2 Setting Socket Options (System Services)

Example 2–14 shows how to set socket options using $QIO system services.

Example 2–14 Setting Socket Options (System Services)

```
#include <descrip.h>              /* define OpenVMS descriptors           */
#include <efndef.h>               /* define 'EFN$C_ENF' event flag        */
#include <in.h>                   /* define internet related constants,   */
                                  /* functions, and structures            */
#include <inet.h>                 /* define network address info          */
#include <iodef.h>                /* define i/o function codes            */
#include <netdb.h>                /* define network database library info */
#include <ssdef.h>                /* define system service status codes   */
#include <starlet.h>              /* define system service calls          */
#include <stdio.h>                /* define standard i/o functions        */
#include <stdlib.h>               /* define standard library functions    */
#include <string.h>               /* define string handling functions     */
#include <stsdef.h>               /* define condition value fields        */
#include <tcpip$inetdef.h>        /* define tcp/ip network constants,     */
                                  /* structures, and functions            */

#define SERV_BACKLOG    1             /* server backlog                   */
#define SERV_PORTNUM    12345         /* server port number               */

struct iosb
    {                                 /* i/o status block                 */
    unsigned short status;            /* i/o completion status            */
    unsigned short bytcnt;            /* bytes transferred if read/write  */
    void *details;                    /* address of buffer or parameter   */
    };

struct itemlst_2
    {                                 /* item-list 2 descriptor/element   */
    unsigned short length;            /* length                           */
    unsigned short type;              /* parameter type                   */
    void *address;                    /* address of item list             */
    };

struct itemlst_3
    {                                 /* item-list 3 descriptor/element   */
    unsigned short length;            /* length                           */
    unsigned short type;              /* parameter type                   */
    void *address;                    /* address of item list             */
    unsigned int *retlen;             /* address of returned length       */
    };

struct sockchar
    {                                 /* socket characteristics           */
    unsigned short prot;              /* protocol                         */
    unsigned char type;               /* type                             */
    unsigned char af;                 /* address format                   */
    };


int main( void )
{
    int optval = 1;                   /* reuseaddr option value (on)      */

    struct iosb iosb;                 /* i/o status block                 */
    unsigned int status;              /* system service return status     */

    unsigned short conn_channel;      /* connect inet device i/o channel  */
```

(continued on next page)

Example 2–14 (Cont.) Setting Socket Options (System Services)

```
    unsigned short listen_channel;      /* listen inet device i/o channel   */
    struct sockchar listen_sockchar;    /* listen socket characteristics    */

    unsigned int cli_addrlen;           /* returned length of client socket */
                                        /* address structure                */
    struct sockaddr_in cli_addr;        /* client socket address structure  */
    struct itemlst_3 cli_itemlst;       /* client socket address item-list  */

    struct sockaddr_in serv_addr;       /* server socket address structure  */
    struct itemlst_2 serv_itemlst;      /* server socket address item-list  */

    struct itemlst_2 sockopt_itemlst;   /* server socket option item-list   */
    struct itemlst_2 reuseaddr_itemlst; /* reuseaddr option item-list       */
    $DESCRIPTOR( inet_device,           /* string descriptor with logical   */
                 "TCPIP$DEVICE:" );     /* name of network pseudodevice     */

    /*
     * initialize socket characteristics
     */

    listen_sockchar.prot = TCPIP$C_TCP;
    listen_sockchar.type = TCPIP$C_STREAM;
    listen_sockchar.af   = TCPIP$C_AF_INET;

    /*
     * initialize reuseaddr's item-list element
     */

    reuseaddr_itemlst.length  = sizeof( optval );
    reuseaddr_itemlst.type    = TCPIP$C_REUSEADDR;
    reuseaddr_itemlst.address = &optval;

    /*
     * initialize setsockopt's item-list descriptor
     */

    sockopt_itemlst.length  = sizeof( reuseaddr_itemlst );
    sockopt_itemlst.type    = TCPIP$C_SOCKOPT;
    sockopt_itemlst.address = &reuseaddr_itemlst;

    /*
     * initialize client's item-list descriptor
     */

    cli_itemlst.length  = sizeof( cli_addr );
    cli_itemlst.type    = TCPIP$C_SOCK_NAME;
    cli_itemlst.address = &cli_addr;
    cli_itemlst.retlen  = &cli_addrlen;

    /*
     * initialize server's item-list descriptor
     */

    serv_itemlst.length  = sizeof( serv_addr );
    serv_itemlst.type    = TCPIP$C_SOCK_NAME;
    serv_itemlst.address = &serv_addr;

    /*
     * initialize server's socket address structure
     */

    memset( &serv_addr, 0, sizeof(serv_addr) );
    serv_addr.sin_family      = TCPIP$C_AF_INET;
    serv_addr.sin_port        = htons( SERV_PORTNUM );
    serv_addr.sin_addr.s_addr = TCPIP$C_INADDR_ANY;
```

(continued on next page)

Example 2–14 (Cont.) Setting Socket Options (System Services)

```
/*
 * assign i/o channels to network device
 */
status = sys$assign( &inet_device,      /* device name                  */
                     &listen_channel,   /* i/o channel                  */
                     0,                 /* access mode                  */
                     0                  /* not used                     */
                   );

if ( status & STS$M_SUCCESS )
    status = sys$assign( &inet_device,  /* device name                  */
                         &conn_channel, /* i/o channel                  */
                         0,             /* access mode                  */
                         0              /* not used                     */
                       );

if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to assign i/o channel(s)\n" );
    exit( status );
    }

/*
 * create a listen socket
 */

status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                   listen_channel,      /* i/o channel                  */
                   IO$_SETMODE,         /* i/o function code            */
                   &iosb,               /* i/o status block             */
                   0,                   /* ast service routine          */
                   0,                   /* ast parameter                */
                   &listen_sockchar,    /* p1 - socket characteristics  */
                   0,                   /* p2                           */
                   0,                   /* p3                           */
                   0,                   /* p4                           */
                   0,                   /* p5                           */
                   0                    /* p6                           */
                 );

if ( status & STS$M_SUCCESS )
    status = iosb.status;

if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to create socket\n" );
    exit( status );
    }

/*
 * bind server's ip address and port number to listen socket
 */
```

(continued on next page)

Example 2–14 (Cont.) Setting Socket Options (System Services)

```
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       listen_channel,      /* i/o channel                  */
                       IO$_SETMODE,         /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       0,                   /* p4                           */
                       &sockopt_itemlst,    /* p5 - socket options          */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to set socket option\n" );
        exit( status );
        }

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       listen_channel,      /* i/o channel                  */
                       IO$_SETMODE,         /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       &serv_itemlst,       /* p3 - local socket name       */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to bind socket\n" );
        exit( status );
        }

    /*
     * set socket as a listen socket
     */
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       listen_channel,      /* i/o channel                  */
                       IO$_SETMODE,         /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       SERV_BACKLOG,        /* p4 - connection backlog      */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );
```

(continued on next page)

Example 2–14 (Cont.) Setting Socket Options (System Services)

```
    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to set socket passive\n" );
        exit( status );
        }

    /*
     * accept connection from a client
     */

    printf( "Waiting for a client connection on port: %d\n",
            ntohs(serv_addr.sin_port)
          );

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       listen_channel,      /* i/o channel                  */
                       IO$_ACCESS|IO$M_ACCEPT,
                                            /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       &conn_channel,       /* p4 - i/o channel for new     */
                                            /*      connection              */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to accept client connection\n" );
        exit( status );
        }
    /*
     * log client connection request
     */

    memset( &cli_addr, 0, sizeof(cli_addr) );

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       conn_channel,        /* i/o channel                  */
                       IO$_SENSEMODE,       /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       &cli_itemlst,        /* p4 - peer socket name        */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;
```

(continued on next page)

Example 2–14 (Cont.) Setting Socket Options (System Services)

```
    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to get client name\n" );
        exit( status );
        }

    printf( "Accepted connection from host: %s, port: %d\n",
            inet_ntoa(cli_addr.sin_addr), ntohs(cli_addr.sin_port)
          );

    exit( EXIT_SUCCESS );
}
```

2.9 Reading Data

TCP/IP Services allows the application to read data after it performs the following operations:

- Create a socket
- Bind a socket name to the socket
- Establish a connection

2.9.1 Reading Data (Sockets API)

Example 2–15 shows a TCP client using the `recv( )` function to read data.

Example 2–15 Reading Data (Sockets API)

```
#include <in.h>                     /* define internet related constants,   */
                                    /* functions, and structures            */
#include <inet.h>                   /* define network address info          */
#include <netdb.h>                  /* define network database library info */
#include <socket.h>                 /* define BSD socket api                */
#include <stdio.h>                  /* define standard i/o functions        */
#include <stdlib.h>                 /* define standard library functions    */
#include <string.h>                 /* define string handling functions     */

#define BUFSZ       1024                /* user input buffer size           */
#define PORTNUM     12345               /* server port number               */

void get_servaddr( void *addrptr )
{
    char buf[BUFSIZ];
    struct in_addr val;
    struct hostent *host;

    while ( TRUE )
        {
        printf( "Enter remote host: " );

        if ( fgets(buf, sizeof(buf), stdin) == NULL )
            {
            printf( "Failed to read user input\n" );
            exit( EXIT_FAILURE );
            }

        buf[strlen(buf)-1] = 0;
```

(continued on next page)

Example 2–15 (Cont.) Reading Data (Sockets API)

```
        val.s_addr = inet_addr( buf );
        if ( val.s_addr != INADDR_NONE )
            {
            memcpy( addrptr, &val, sizeof(struct in_addr) );
            break;
            }
        if ( (host = gethostbyname(buf)) )
            {
            memcpy( addrptr, host->h_addr, sizeof(struct in_addr) );
            break;
            }
        }
}

int main( void )
{
    char buf[512];
    int nbytes, sockfd;
    struct sockaddr_in addr;
    /*
     * initialize socket address structure
     */
    memset( &addr, 0, sizeof(addr) );
    addr.sin_family = AF_INET;
    addr.sin_port   = htons( PORTNUM );
    get_servaddr( &addr.sin_addr );
    /*
     * create a socket
     */
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }
    /*
     * connect to specified host and port number
     */
    printf( "Initiated connection to host: %s, port: %d\n",
            inet_ntoa(addr.sin_addr), ntohs(addr.sin_port)
          );
    if ( connect(sockfd, (struct sockaddr *) &addr, sizeof(addr)) < 0 )
        {
        perror( "Failed to connect to server" );
        exit( EXIT_FAILURE );
        }
    /*
     * read data from connection
     */
    nbytes = recv( sockfd,1  buf,2  sizeof(buf),3  0 4  );
    if ( nbytes < 0 )
        {
        perror( "Failed to read data from connection" );
        exit( EXIT_FAILURE );
        }
```

(continued on next page)

Example 2–15 (Cont.) Reading Data (Sockets API)

```
    buf[nbytes] = 0;
    printf( "Data received: %s\n", buf );

    exit( EXIT_SUCCESS );
}
```

1 `sockfd` is the socket descriptor previously defined by a call to the `connect( )` function.

2 `buf` points to the receive buffer where the data is placed.

3 `sizeof (buf)` is the size of the receive buffer.

4 `0` indicates that out-of-band data is not being received.

2.9.2 Reading Data (System Services)

The $QIO IO$_READVBLK function transfers data received from the internet host (and kept in system dynamic memory) into the address space of the user's process. After the read operation completes, the data in dynamic memory is discarded.

Example 2–16 shows a TCP client using the IO$_READVBLK function to read data into a single I/O buffer.

Example 2–16 Reading Data (System Services)

```
#include <descrip.h>                /* define OpenVMS descriptors           */
#include <efndef.h>                 /* define 'EFN$C_ENF' event flag        */
#include <in.h>                     /* define internet related constants,   */
                                    /* functions, and structures            */
#include <inet.h>                   /* define network address info          */
#include <iodef.h>                  /* define i/o function codes            */
#include <netdb.h>                  /* define network database library info */
#include <ssdef.h>                  /* define system service status codes   */
#include <starlet.h>                /* define system service calls          */
#include <stdio.h>                  /* define standard i/o functions        */
#include <stdlib.h>                 /* define standard library functions    */
#include <string.h>                 /* define string handling functions     */
#include <stsdef.h>                 /* define condition value fields        */
#include <tcpip$inetdef.h>          /* define tcp/ip network constants,     */
                                    /* structures, and functions            */

#define BUFSZ       1024                /* user input buffer size           */
#define PORTNUM     12345               /* server port number               */

struct iosb
    {                                   /* i/o status block                 */
    unsigned short status;              /* i/o completion status            */
    unsigned short bytcnt;              /* bytes transferred if read/write  */
    void *details;                      /* address of buffer or parameter   */
    };
```

(continued on next page)

Example 2–16 (Cont.) Reading Data (System Services)

```
struct itemlst_2
    {                                   /* item-list 2 descriptor/element   */
    unsigned short length;              /* length                           */
    unsigned short type;                /* parameter type                   */
    void *address;                      /* address of item list             */
    };

struct sockchar
    {                                   /* socket characteristics           */
    unsigned short prot;                /* protocol                         */
    unsigned char type;                 /* type                             */
    unsigned char af;                   /* address format                   */
    };

void get_servaddr( void *addrptr )
{
    char buf[BUFSIZ];
    struct in_addr val;
    struct hostent *host;

    while ( TRUE )
        {
        printf( "Enter remote host: " );

        if ( fgets(buf, sizeof(buf), stdin) == NULL )
            {
            printf( "Failed to read user input\n" );
            exit( EXIT_FAILURE );
            }

        buf[strlen(buf)-1] = 0;

        val.s_addr = inet_addr( buf );

        if ( val.s_addr != INADDR_NONE )
            {
            memcpy( addrptr, &val, sizeof(struct in_addr) );
            break;
            }
        if ( (host = gethostbyname(buf)) )
            {
            memcpy( addrptr, host->h_addr, sizeof(struct in_addr) );
            break;
            }
        }
}

int main( void )
{
    char buf[512];                      /* data buffer                      */
    int buflen = sizeof( buf );         /* length of data buffer            */

    struct iosb iosb;                   /* i/o status block                 */
    unsigned int status;                /* system service return status     */
    unsigned short channel;             /* network device i/o channel       */
    struct sockchar sockchar;           /* socket characteristics buffer    */
    struct sockaddr_in addr;            /* socket address structure         */
    struct itemlst_2 addr_itemlst;      /* socket address item-list         */
    $DESCRIPTOR( inet_device,           /* string descriptor with logical   */
                 "TCPIP$DEVICE:" );     /* name of network pseudodevice     */
```

(continued on next page)

Example 2–16 (Cont.) Reading Data (System Services)

```
/*
 * initialize socket characteristics
 */

sockchar.prot = TCPIP$C_TCP;
sockchar.type = TCPIP$C_STREAM;
sockchar.af   = TCPIP$C_AF_INET;

/*
 * initialize socket address item-list descriptor
 */

addr_itemlst.length  = sizeof( addr );
addr_itemlst.type    = TCPIP$C_SOCK_NAME;
addr_itemlst.address = &addr;

/*
 * initialize socket address structure
 */

memset( &addr, 0, sizeof(addr) );
addr.sin_family = TCPIP$C_AF_INET;
addr.sin_port   = htons( PORTNUM );
get_servaddr( &addr.sin_addr );

/*
 * assign i/o channel to network device
 */
status = sys$assign( &inet_device,      /* device name                  */
                     &channel,          /* i/o channel                  */
                     0,                 /* access mode                  */
                     0                  /* not used                     */
                   );

if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to assign i/o channel\n" );
    exit( status );
    }

/*
 * create a socket
 */

status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                   channel,             /* i/o channel                  */
                   IO$_SETMODE,         /* i/o function code            */
                   &iosb,               /* i/o status block             */
                   0,                   /* ast service routine          */
                   0,                   /* ast parameter                */
                   &sockchar,           /* p1 - socket characteristics  */
                   0,                   /* p2                           */
                   0,                   /* p3                           */
                   0,                   /* p4                           */
                   0,                   /* p5                           */
                   0                    /* p6                           */
                 );

if ( status & STS$M_SUCCESS )
    status = iosb.status;
```

(continued on next page)

Example 2–16 (Cont.) Reading Data (System Services)

```
    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to create socket\n" );
        exit( status );
        }

    /*
     * connect to specified host and port number
     */

    printf( "Initiated connection to host: %s, port: %d\n",
            inet_ntoa(addr.sin_addr), ntohs(addr.sin_port)
          );
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       channel,             /* i/o channel                  */
                       IO$_ACCESS,          /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       &addr_itemlst,       /* p3 - remote socket name      */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to connect to server\n" );
        exit( status );
        }

    /*
     * read data from connection
     */

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       channel,             /* i/o channel                  */
                       IO$_READVBLK,        /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       buf,                 /* p1 - buffer address          */
                       buflen,              /* p2 - buffer length           */
                       0,                   /* p3                           */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to read data from connection\n" );
        exit( status );
        }
```

(continued on next page)

Example 2–16 (Cont.) Reading Data (System Services)

```
    buf[iosb.bytcnt] = 0;
    printf( "Data received: %s\n", buf );

    exit( EXIT_SUCCESS );
}
```

You can also specify a list of read buffers by omitting the **p1** and **p2** arguments and passing the list of buffers as the **p6** parameter. See Section 5.5.2 for more information.

2.10 Receiving IP Multicast Datagrams

Before a host can receive (read) IP multicast datagrams destined for a particular multicast group other than `all hosts group`, the application must direct the host it is running on to become a member of that multicast group.

To join a group or drop membership from a group, specify the following options. Make sure you include the IN.H header file.

- To join a multicast group, specify the appropriate option to the `setsockopt( )` system call:
  - IP_ADD_MEMBERSHIP (Sockets API)
  - TCPIP$C_IP_ADD_MEMBERSHIP (System Services)

  For example:

```
struct ip_mreq mreq;
if (setsockopt( sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, & mreq,
          sizeof(mreq)) == -1)
          perror("setsockopt");
```

  The `mreq` variable has the following structure:

```
struct ip_mreq {
          struct in_addr (imr_multiaddr); /* IP multicast address of group */
          struct in_addr (imr_interface); /* local IP address of interface */
};
```

  In this structure, `imr_interface` can be specified as INADDR_ANY, which allows an application to choose the default multicast interface.

  Each multicast group membership is associated with a particular interface, and multiple interfaces can join the same group. Alternatively, specifying one of the host's local addresses allows an application to select a particular, multicast-capable interface. The maximum number of memberships that can be added on a single socket is subject to the IP_MAX_MEMBERSHIPS value, which is defined in the IN.H header file.

  If multiple sockets request that a host join a multicast group, the host remains a member of that multicast group until the last of those sockets is closed.

- To drop membership from a multicast group, specify the appropriate option to the `setsockopt( )` system call:
  - IP_DROP_MEMBERSHIP (Sockets API)
  - TCPIP$C_IP_DROP_MEMBERSHIP (System Services)

For example:

```
struct ip_mreq mreq;
if (setsockopt( sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq,
        sizeof(mreq))== -1)
perror("setsockopt");
```

The mreq variable contains the same structure values used for adding membership.

To receive multicast datagrams sent to a specific UDP port, the receiving socket must have been bound to that port using the $QIO(IO$_SETMODE) system service function or the bind( ) Sockets API function. More than one process can receive UDP datagrams destined for the same port if the function is preceded by a setsockopt( ) system call that specifies the SO_REUSEPORT option.

For example:

```
int setreuse = 1;
if (setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &setreuse,
        sizeof(setreuse)) == -1)
 perror("setsockopt");
```

When the SO_REUSEPORT option is set, every incoming multicast or broadcast UDP datagram destined for the shared port is delivered to all sockets bound to that port.

Delivery of IP multicast datagrams to SOCK_RAW sockets is determined by the protocol type of the destination.

2.11 Reading Out-of-Band Data (TCP Protocol)

Only stream-type (TCP/IP) sockets can receive out-of-band (OOB) data. Upon receiving a TCP/IP OOB character, TCP/IP Services stores a pointer in the received stream to the character that precedes the OOB character.

A read operation with a user buffer size larger than the size of the received stream up to the OOB character completes by returning to the user the received stream up to, but not including, the OOB character.

Poll the socket to determine whether additional read operations are needed before getting all the characters from the stream that precedes the OOB character.

2.11.1 Reading OOB Data (Sockets API)

You can use the recv( ) socket function with the MSG_OOB flag set to receive out-of-band data regardless of how many of the preceding characters in the stream you have received.

Example 2–17 shows a TCP server using the recv( ) function to receive out-of-band data.

Example 2–17 Reading OOB Data (Sockets API)

```
retval = recv(sock_3,1  message,2 sizeof(message),3 flag);4
if (retval == -1)
        {
    perror ("receive");
    cleanup( 2, sock_2, sock_3);
        }
else
    printf (" %s\n", message);
```

1 sock_3 specifies that OOB data is received from socket 2.

2 message points to the read buffer where the data is placed.

3 sizeof (message) indicates the size of the read buffer.

4 flag, when set to MSG_OOB, indicates that OOB data is being received in the specified buffer.

2.11.2 Reading OOB Data (System Services)

To receive OOB data from a remote process, use the IO$_READVBLK function with the IO$M_INTERRUPT modifier.

To poll the socket, use a $QIO command with the IO$_SENSEMODE function and the TCPIP$C_IOCTL subfunction that specifies the SIOCATMARK operation.

If the SIOCATMARK returns a value of 0, use additional read QIOs to read more data before reading the OOB character. If the SIOCATMARK returns a value of 1, the next read QIO returns the OOB character.

These functions are useful if a socket has the OOBINLINE socket option set. The OOB character is read with the characters in the stream (IO$_READVBLK) but is not read before the preceding characters. To determine whether or not the first character in the user buffer on the next read is an OOB, poll the socket.

To get a received OOB character for a socket with the socket option OOBINLINE clear, use one of the following functions:

- $QIO with the function IO$_READVBLK | IO$M_INTERRUPT
- IO$_READVBLK with the **P4** parameter TCPIP$C_MSG_OOB flag set

Example 2–18 shows how to use the IO$M_INTERRUPT modifier to read out-of-band data.

Example 2–18 Reading OOB Data (System Services)

```
/*
**  Attempt to receive the OOB data from the client.
**  Use the function code of IO$_READVBLK, passing the address of the
**  input buffer to P1, and the OOB code, TCPIP$C_MSG_OOB, to P4.
**  We support the sending and receiving of a one byte of OOB data.
*/
        sysSrvSts = sys$qiow( 0,              /* efn.v | 0             */
                              IOChanClient,   /* chan.v                */
                              IO$_READVBLK,   /* func.v                */
                              &iosb,          /* iosb.r | 0            */
                              0, 0,           /* astadr, astprm: UNUSED */
                              &OOBBuff,       /* p1.r IO buffer        */
                              MaxBuff,        /* p2.v IO buffer  size  */
                              0,              /* p3 UNUSED             */
                              TCPIP$C_MSG_OOB, /* p4.v IO options flag  */
                              0, 0            /* p5, p6 UNUSED         */
                             );
        if((( sysSrvSts & 1  ) != 1 ) || /* Validate the system service. */
           ((  iosb.cond_value & 1  ) != 1))  /* Validate the IO status. */
            {
            cleanup( IOChanClient );
            cleanup( IOChannel );
            errorExit( sysSrvSts, iosb.cond_value );
            }
        else
            if( iosb.count == 0 )
                printf( "    FAILED to receive the message, no connection.\n" );
            else
                printf( "    SUCCEEDED in receiving '%d'\n", OOBBuff );
```

2.12 Peeking at Queued Messages

You can use a read operation to look at data in a socket receive queue without removing the data from the buffer. This is called **peeking**.

2.12.1 Peeking at Data (Sockets API)

Use the MSG_PEEK flag with the recv( ) function to peek at data in the socket receive queue. Example 2–19 shows a TCP server using the recv( ) function with the MSG_PEEK flag to peek at received data.

Example 2–19 Peeking at Data (Sockets API)

```
#include <in.h>                    /* define internet related constants,   */
                                   /* functions, and structures            */
#include <inet.h>                  /* define network address info          */
#include <netdb.h>                 /* define network database library info */
#include <socket.h>                /* define BSD socket api                */
#include <stdio.h>                 /* define standard i/o functions        */
#include <stdlib.h>                /* define standard library functions    */
#include <string.h>                /* define string handling functions     */
#include <unixio.h>                /* define unix i/o                      */

#define BUFSZ           128            /* user input buffer size          */
#define SERV_BACKLOG    1              /* server backlog                  */
#define SERV_PORTNUM    1234           /* server port number              */

int main( void )
{
    char buf[BUFSIZ];                  /* user input buffer               */
    int conn_sockfd;                   /* connection socket descriptor    */
    int listen_sockfd;                 /* listen socket descriptor        */

    int optval = 1;                    /* SO_REUSEADDR'S option value (on) */

    unsigned int cli_addrlen;          /* returned length of client socket */
                                       /* address structure               */
    struct sockaddr_in cli_addr;       /* client socket address structure  */
    struct sockaddr_in serv_addr;      /* server socket address structure  */

    /*
     * initialize client's socket address structure
     */

    memset( &cli_addr, 0, sizeof(cli_addr) );

    /*
     * initialize server's socket address structure
     */

    memset( &serv_addr, 0, sizeof(serv_addr) );
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_port        = htons( SERV_PORTNUM );
    serv_addr.sin_addr.s_addr = INADDR_ANY;

    /*
     * create a listen socket
     */

    if ( (listen_sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }

    /*
     * bind server's ip address and port number to listen socket
     */

    if ( setsockopt(listen_sockfd,
                    SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval)) < 0 )
        {
        perror( "Failed to set socket option" );
        exit( EXIT_FAILURE );
        }
```

(continued on next page)

Example 2–19 (Cont.) Peeking at Data (Sockets API)

```
    if ( bind(listen_sockfd,
              (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0 )
        {
        perror( "Failed to bind socket" );
        exit( EXIT_FAILURE );
        }

    /*
     * set socket as a listen socket
     */

    if ( listen(listen_sockfd, SERV_BACKLOG) < 0 )
        {
        perror( "Failed to set socket passive" );
        exit( EXIT_FAILURE );
        }

    /*
     * accept connection from a client
     */

    printf( "Waiting for a client connection on port: %d\n",
            ntohs(serv_addr.sin_port)
          );

    cli_addrlen = sizeof(cli_addr);

    conn_sockfd = accept( listen_sockfd,
                          (struct sockaddr *) &cli_addr,
                          &cli_addrlen
                        );
    if ( conn_sockfd < 0 )
        {
        perror( "Failed to accept client connection" );
        exit( EXIT_FAILURE );
        }

    /*
     * ask client to pick a character
     */

    sprintf( buf, "Please pick a character: \r\n");

    if ( send(conn_sockfd, buf, strlen(buf), 0) != strlen(buf) )
        {
        perror( "Failed to write data to connection" );
        exit( EXIT_FAILURE );
        }

    /*
     * peek at client's reply
     */
    if ( recv(conn_sockfd 1 , buf 2 , 1 3 , MSG_PEEK 4 ) != 1 )
        {
        perror( "Failed to read data from connection" );
        exit( EXIT_FAILURE );
        }

    sprintf( buf, "Before receiving, I see you picked '%c'.\r\n", buf[0] );

    if ( send(conn_sockfd, buf, strlen(buf), 0) != strlen(buf) )
        {
        perror( "Failed to write data to connection" );
        exit( EXIT_FAILURE );
        }
```

(continued on next page)

Example 2–19 (Cont.) Peeking at Data (Sockets API)

```
    /*
     * now, read client's reply
     */

    if ( recv(conn_sockfd, buf, 1, 0) != 1 )
        {
        perror( "Failed to read data from connection" );
        exit( EXIT_FAILURE );
        }

    sprintf( buf, "Sure enough, I received '%c'.\r\n", buf[0] );

    if ( send(conn_sockfd, buf, strlen(buf), 0) != strlen(buf) )
        {
        perror( "Failed to write data to connection" );
        exit( EXIT_FAILURE );
        }

    /*
     * close sockets
     */

    if ( close(conn_sockfd) < 0 )
        {
        perror( "Failed to close socket" );
        exit( EXIT_FAILURE );
        }

    if ( close(listen_sockfd) < 0 )
        {
        perror( "Failed to close socket" );
        exit( EXIT_FAILURE );
        }

    exit( EXIT_SUCCESS );
}
```

The `recv( )` function receives data from a connected socket and places it in a buffer, as follows:

1. `conn_sockfd` is the socket descriptor created as a result of a call to the `accept( )` function.
2. `buf` points to the buffer into which received data is placed.
3. `1` indicates the size of the buffer.
4. `MSG_PEEK` is the flag that specifies the character entered is looked at without removing it from the buffer.

2.12.2 Peeking at Data (System Services)

To peek at data that is next in the socket receive queue, use the IO$_READVBLK function of the $QIO system service and use the TCPIP$C_MSG_PEEK flag. This allows you to use multiple read operations on the same data.

2.13 Writing Data

For programs that use TCP, data writing occurs after a client program initiates a connection and after the server program accepts the connection. When using UDP, you also have the option of establishing a default peer address with a specific socket, but this is not required for data transfer.

2.13.1 Writing Data (Sockets API)

Example 2–20 shows a TCP server using the send( ) function to transmit data.

Example 2–20 Writing Data (Sockets API)

```
#include <in.h>                    /* define internet related constants,   */
                                   /* functions, and structures            */
#include <inet.h>                  /* define network address info          */
#include <netdb.h>                 /* define network database library info */
#include <socket.h>                /* define BSD socket api                */
#include <stdio.h>                 /* define standard i/o functions        */
#include <stdlib.h>                /* define standard library functions    */
#include <string.h>                /* define string handling functions     */

#define SERV_BACKLOG    1              /* server backlog                   */
#define SERV_PORTNUM    12345          /* server port number               */

int main( void )
{
    int optval = 1;                    /* SO_REUSEADDR's option value (on) */

    int conn_sockfd;                   /* connection socket descriptor     */
    int listen_sockfd;                 /* listen socket descriptor         */

    unsigned int cli_addrlen;          /* returned length of client socket */
                                       /* address structure                */
    struct sockaddr_in cli_addr;       /* client socket address structure  */
    struct sockaddr_in serv_addr;      /* server socket address structure  */

    char buf[] = "Hello, world!";      /* data buffer                      */

    /*
     * initialize server's socket address structure
     */

    memset( &serv_addr, 0, sizeof(serv_addr) );
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_port        = htons( SERV_PORTNUM );
    serv_addr.sin_addr.s_addr = INADDR_ANY;

    /*
     * create a listen socket
     */

    if ( (listen_sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }
```

(continued on next page)

Example 2–20 (Cont.) Writing Data (Sockets API)

```
    /*
     * bind server's ip address and port number to listen socket
     */

    if ( setsockopt(listen_sockfd,
                    SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval)) < 0 )
        {
        perror( "Failed to set socket option" );
        exit( EXIT_FAILURE );
        }

    if ( bind(listen_sockfd,
              (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0 )
        {
        perror( "Failed to bind socket" );
        exit( EXIT_FAILURE );
        }
    /*
     * set socket as a listen socket
     */

    if ( listen(listen_sockfd, SERV_BACKLOG) < 0 )
        {
        perror( "Failed to set socket passive" );
        exit( EXIT_FAILURE );
        }

    /*
     * accept connection from a client
     */

    printf( "Waiting for a client connection on port: %d\n",
            ntohs(serv_addr.sin_port)
          );

    conn_sockfd = accept( listen_sockfd, (struct sockaddr *) 0, 0 );

    if ( conn_sockfd < 0 )
        {
        perror( "Failed to accept client connection" );
        exit( EXIT_FAILURE );
        }

    /*
     * log client connection request
     */

    cli_addrlen = sizeof(cli_addr);
    memset( &cli_addr, 0, sizeof(cli_addr) );

    if ( getpeername(conn_sockfd,
                     (struct sockaddr *) &cli_addr, &cli_addrlen) < 0 )
        {
        perror( "Failed to get client name" );
        exit( EXIT_FAILURE );
        }

    printf( "Accepted connection from host: %s, port: %d\n",
            inet_ntoa(cli_addr.sin_addr), ntohs(cli_addr.sin_port)
          );
```

(continued on next page)

Example 2–20 (Cont.) Writing Data (Sockets API)

```
    /*
     * write data to connection
     */
    if ( send(conn_sockfd, 1  buf, 2  sizeof(buf),3  0 4 ) < 0 )
        {
        perror( "Failed to write data to connection" );
        exit( EXIT_FAILURE );
        }

    printf( "Data sent: %s\n", buf );

    exit( EXIT_SUCCESS );
}
```

1 conn_sockfd specifies the connected socket that is to receive the data.

2 buf is the address of the send buffer where the data to be sent is placed.

3 sizeof (buf) indicates the size of the send buffer.

4 flag, when set to 0, indicates that OOB data is not being sent.

2.13.2 Writing Data (System Services)

The IO$_WRITEVBLK function of the $QIO system service copies data from the address space of the user's process to system dynamic memory and then transfers the data to an internet host or port.

Example 2–21 shows a TCP server using the IO$_WRITEVBLK function to transmit a single data buffer. The $QIO(IO$_ACCESS | IO$M_ACCEPT) function was previously executed to establish the connection with the client.

Example 2–21 Writing Data (System Services)

```
#include <descrip.h>                /* define OpenVMS descriptors           */
#include <efndef.h>                 /* define 'EFN$C_ENF' event flag        */
#include <in.h>                     /* define internet related constants,   */
                                    /* functions, and structures            */
#include <inet.h>                   /* define network address info          */
#include <iodef.h>                  /* define i/o function codes            */
#include <netdb.h>                  /* define network database library info */
#include <ssdef.h>                  /* define system service status codes   */
#include <starlet.h>                /* define system service calls          */
#include <stdio.h>                  /* define standard i/o functions        */
#include <stdlib.h>                 /* define standard library functions    */
#include <string.h>                 /* define string handling functions     */
#include <stsdef.h>                 /* define condition value fields        */
#include <tcpip$inetdef.h>          /* define tcp/ip network constants,     */
                                    /* structures, and functions            */

#define SERV_BACKLOG    1               /* server backlog                   */
#define SERV_PORTNUM    12345           /* server port number               */

struct iosb
    {                                   /* i/o status block                 */
    unsigned short status;              /* i/o completion status            */
    unsigned short bytcnt;              /* bytes transferred if read/write  */
    void *details;                      /* address of buffer or parameter   */
    };
```

(continued on next page)

Example 2–21 (Cont.) Writing Data (System Services)

```
struct itemlst_2
    {                                   /* item-list 2 descriptor/element   */
    unsigned short length;              /* length                           */
    unsigned short type;                /* parameter type                   */
    void *address;                      /* address of item list             */
    };

struct itemlst_3
    {                                   /* item-list 3 descriptor/element   */
    unsigned short length;              /* length                           */
    unsigned short type;                /* parameter type                   */
    void *address;                      /* address of item list             */
    unsigned int *retlen;               /* address of returned length       */
    };

struct sockchar
    {                                   /* socket characteristics           */
    unsigned short prot;                /* protocol                         */
    unsigned char type;                 /* type                             */
    unsigned char af;                   /* address format                   */
    };

int main( void )
{
    int optval = 1;                     /* reuseaddr option value (on)      */

    struct iosb iosb;                   /* i/o status block                 */
    unsigned int status;                /* system service return status     */

    unsigned short conn_channel;        /* connect inet device i/o channel  */

    unsigned short listen_channel;      /* listen inet device i/o channel   */
    struct sockchar listen_sockchar;    /* listen socket characteristics    */

    unsigned int cli_addrlen;           /* returned length of client socket */
                                        /* address structure                */
    struct sockaddr_in cli_addr;        /* client socket address structure  */
    struct itemlst_3 cli_itemlst;       /* client socket address item-list  */

    struct sockaddr_in serv_addr;       /* server socket address structure  */
    struct itemlst_2 serv_itemlst;      /* server socket address item-list  */

    struct itemlst_2 sockopt_itemlst;   /* server socket option item-list   */
    struct itemlst_2 reuseaddr_itemlst; /* reuseaddr option item-list       */
    char buf[] = "Hello, world!";       /* data buffer                      */
    int buflen = sizeof( buf );         /* length of data buffer            */

    $DESCRIPTOR( inet_device,           /* string descriptor with logical   */
                 "TCPIP$DEVICE:" );     /* name of network pseudodevice     */

    /*
     * initialize socket characteristics
     */

    listen_sockchar.prot = TCPIP$C_TCP;
    listen_sockchar.type = TCPIP$C_STREAM;
    listen_sockchar.af   = TCPIP$C_AF_INET;

    /*
     * initialize reuseaddr's item-list element
     */

    reuseaddr_itemlst.length  = sizeof( optval );
    reuseaddr_itemlst.type    = TCPIP$C_REUSEADDR;
    reuseaddr_itemlst.address = &optval;
```

(continued on next page)

Example 2–21 (Cont.) Writing Data (System Services)

```
/*
 * initialize setsockopt's item-list descriptor
 */

sockopt_itemlst.length  = sizeof( reuseaddr_itemlst );
sockopt_itemlst.type    = TCPIP$C_SOCKOPT;
sockopt_itemlst.address = &reuseaddr_itemlst;

/*
 * initialize client's item-list descriptor
 */

cli_itemlst.length  = sizeof( cli_addr );
cli_itemlst.type    = TCPIP$C_SOCK_NAME;
cli_itemlst.address = &cli_addr;
cli_itemlst.retlen  = &cli_addrlen;

/*
 * initialize server's item-list descriptor
 */

serv_itemlst.length  = sizeof( serv_addr );
serv_itemlst.type    = TCPIP$C_SOCK_NAME;
serv_itemlst.address = &serv_addr;

/*
 * initialize server's socket address structure
 */

memset( &serv_addr, 0, sizeof(serv_addr) );
serv_addr.sin_family      = TCPIP$C_AF_INET;
serv_addr.sin_port        = htons( SERV_PORTNUM );
serv_addr.sin_addr.s_addr = TCPIP$C_INADDR_ANY;
/*
 * assign i/o channels to network device
 */

status = sys$assign( &inet_device,      /* device name                  */
                     &listen_channel,   /* i/o channel                  */
                     0,                 /* access mode                  */
                     0                  /* not used                     */
                   );

if ( status & STS$M_SUCCESS )
    status = sys$assign( &inet_device,  /* device name                  */
                         &conn_channel, /* i/o channel                  */
                         0,             /* access mode                  */
                         0              /* not used                     */
                       );

if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to assign i/o channel(s)\n" );
    exit( status );
    }

/*
 * create a listen socket
 */
```

(continued on next page)

Example 2–21 (Cont.) Writing Data (System Services)

```
status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                   listen_channel,      /* i/o channel                  */
                   IO$_SETMODE,         /* i/o function code            */
                   &iosb,               /* i/o status block             */
                   0,                   /* ast service routine          */
                   0,                   /* ast parameter                */
                   &listen_sockchar,    /* p1 - socket characteristics  */
                   0,                   /* p2                           */
                   0,                   /* p3                           */
                   0,                   /* p4                           */
                   0,                   /* p5                           */
                   0                    /* p6                           */
                 );

if ( status & STS$M_SUCCESS )
    status = iosb.status;

if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to create socket\n" );
    exit( status );
    }

/*
 * bind server's ip address and port number to listen socket
 */
status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                   listen_channel,      /* i/o channel                  */
                   IO$_SETMODE,         /* i/o function code            */
                   &iosb,               /* i/o status block             */
                   0,                   /* ast service routine          */
                   0,                   /* ast parameter                */
                   0,                   /* p1                           */
                   0,                   /* p2                           */
                   0,                   /* p3                           */
                   0,                   /* p4                           */
                   &sockopt_itemlst,    /* p5 - socket options          */
                   0                    /* p6                           */
                 );

if ( status & STS$M_SUCCESS )
    status = iosb.status;

if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to set socket option\n" );
    exit( status );
    }

status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                   listen_channel,      /* i/o channel                  */
                   IO$_SETMODE,         /* i/o function code            */
                   &iosb,               /* i/o status block             */
                   0,                   /* ast service routine          */
                   0,                   /* ast parameter                */
                   0,                   /* p1                           */
                   0,                   /* p2                           */
                   &serv_itemlst,       /* p3 - local socket name       */
                   0,                   /* p4                           */
                   0,                   /* p5                           */
                   0                    /* p6                           */
                 );
```

(continued on next page)

Example 2–21 (Cont.) Writing Data (System Services)

```
    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to bind socket\n" );
        exit( status );
        }

    /*
     * set socket as a listen socket
     */
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       listen_channel,      /* i/o channel                  */
                       IO$_SETMODE,         /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       SERV_BACKLOG,        /* p4 - connection backlog      */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to set socket passive\n" );
        exit( status );
        }

    /*
     * accept connection from a client
     */

    printf( "Waiting for a client connection on port: %d\n",
            ntohs(serv_addr.sin_port)
          );

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       listen_channel,      /* i/o channel                  */
                       IO$_ACCESS|IO$M_ACCEPT,
                                            /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       &conn_channel,       /* p4 - i/o channel for new     */
                                            /*      connection              */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;
```

(continued on next page)

Example 2–21 (Cont.) Writing Data (System Services)

```
if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to accept client connection\n" );
    exit( status );
    }
/*
 * log client connection request
 */

memset( &cli_addr, 0, sizeof(cli_addr) );

status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                   conn_channel,        /* i/o channel                  */
                   IO$_SENSEMODE,       /* i/o function code            */
                   &iosb,               /* i/o status block             */
                   0,                   /* ast service routine          */
                   0,                   /* ast parameter                */
                   0,                   /* p1                           */
                   0,                   /* p2                           */
                   0,                   /* p3                           */
                   &cli_itemlst,        /* p4 - peer socket name        */
                   0,                   /* p5                           */
                   0                    /* p6                           */
                 );

if ( status & STS$M_SUCCESS )
    status = iosb.status;

if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to get client name\n" );
    exit( status );
    }

printf( "Accepted connection from host: %s, port: %d\n",
        inet_ntoa(cli_addr.sin_addr), ntohs(cli_addr.sin_port)
      );

/*
 * write data to connection
 */

status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                   conn_channel,        /* i/o channel                  */
                   IO$_WRITEVBLK,       /* i/o function code            */
                   &iosb,               /* i/o status block             */
                   0,                   /* ast service routine          */
                   0,                   /* ast parameter                */
                   buf,                 /* p1 - buffer address          */
                   buflen,              /* p2 - buffer length           */
                   0,                   /* p3                           */
                   0,                   /* p4                           */
                   0,                   /* p5                           */
                   0                    /* p6                           */
                 );

if ( status & STS$M_SUCCESS )
    status = iosb.status;
```

(continued on next page)

Example 2–21 (Cont.) Writing Data (System Services)

```
    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to write data to connection\n" );
        exit( status );
        }

    printf( "Data sent: %s\n", buf );

    exit( EXIT_SUCCESS );
}
```

You can also specify a list of write buffers by omitting the **p1** and **p2** parameters and instead passing the list of buffers as the **p5** parameter. Note that, when writing a list of buffers, the **p5** parameter is used; when reading a list, the **p6** parameter is used. For more information, see Section 5.5.1.

2.14 Writing OOB Data (TCP Protocol)

If your application uses TCP, you can send OOB data to a remote process. At the remote process, the message is delivered to the user through either the data receive or the out-of-band data receive mechanism. You can write only 1 byte of OOB data at a time.

2.14.1 Writing OOB Data (Sockets API)

To send OOB data to a remote process, use the MSG_OOB flag with the send( ), sendmsg( ), and sendto( ) functions.

Example 2–22 shows a TCP server using the MSG_OOB flag with the send( ) function.

Example 2–22 Writing OOB Data (Sockets API)

```
/* This program accepts a connection on TCP port 1234, sends the string,
   "Hello, world!", waits two seconds, sends an urgent BEL (^G), waits
   another two seconds, repeats the Hello message, and terminates. */

#include <types.h>
#include <in.h>
#include <socket.h>
#include <unixio.h>

#define PORTNUM 123
main() {
   struct sockaddr_in lcladdr;
   int r, s, one = 1;
   char *message = "Hello, world!\r\n",
        *oob_message = "\007";
   memset()
```

(continued on next page)

Example 2–22 (Cont.) Writing OOB Data (Sockets API)

```
   lcladdr.sin_family = AF_INET;
   lcladdr.sin_addr.s_addr = INADDR_ANY;
   lcladdr.sin_port = htons(PORTNUM);
   if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) perror("socket");
   if (setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one)))
      perror("setsockopt");
   if (bind(s, &lcladdr, sizeof(lcladdr))) perror("bind");
   if (listen(s, 1)) perror("listen");
   if ((r = accept(s, 0, 0)) < 0) perror("accept");
   if (send(r, message, strlen(message), 0) != strlen(message))
      perror("send");
   sleep(2);
   if (send(r,1  oob_message,2  strlen(oob_message),3  MSG_OOB 4 ) !=
             strlen(oob_message)) perror("send");
   sleep(2);
   if (send(r, message, strlen(message), 0) != strlen(message))
      perror("send");
   sleep(2);
   if (close(r)) perror("close");
   if (close(s)) perror("close");
}
```

The send( ) function is used to send OOB data to a remote socket, as follows:

1. r specifies the remote socket descriptor connected to the local socket as a result of a call to the socket( ) function.
2. oob_message is the buffer containing the OOB data.
3. strlen(oob_message) specifies the length, in bytes, of the buffer containing the out-of-band data.
4. MSG_OOB is the flag that indicates the data will be sent out of band.

2.14.2 Writing OOB Data (System Services)

To send out-of-band data to a remote process, use the $QIO system service and use the IO$_WRITEVBLK function with the IO$M_INTERRUPT modifier. Example 2–23 shows a TCP server using the MSG_OOB flag with the send( ) function.

Example 2–23 Writing OOB Data (System Services)

```
/*
**
**  Attempt to send Out Of Band data to a previously established network
**  connection. Use the function code of IO$_WRITEVBLK, passing the address
**  of the buffer to P1, and the OOB code, TCPIP$C_MSG_OOB, to P4.
**
*/
    OOBBuff = 7;
    sysSrvSts = sys$qiow( 0,                /* efn.v | 0              */
                          IOChannel,        /* chan.v                 */
                          IO$_WRITEVBLK,    /* func.v                 */
                          &iosb,            /* iosb.r | 0             */
                          0, 0,             /* astadr, astprm: UNUSED */
                          &OOBBuff,         /* p1.r IO buffer         */
                          1,                /* p2.v IO buffer  size   */
                          0,                /* p3 UNUSED              */
                          TCPIP$C_MSG_OOB,  /* p4.v IO options flag   */
                          0, 0              /* p5, p6 UNUSED          */
                        );
    if((( sysSrvSts & 1  ) != 1 ) || /* Validate the system service status. */
       ((  iosb.cond_value & 1  ) != 1))  /* Validate the IO status. */
        {
        cleanup( IOChannel );
        errorExit( sysSrvSts, iosb.cond_value );
        }
    else
        if( iosb.count == 0 )
            printf( "    FAILED to send the OOB message, no connection.\n" );
        else
            printf( "    SUCCEEDED in sending the OOB message.\n" );
```

2.15 Sending Datagrams (UDP Protocol)

An application that uses UDP can send a datagram to a remote host, send broadcast datagrams to multiple remote hosts, or send multicast datagrams to members of a group.

With broadcasting, you send datagrams in one operation to multiple remote hosts on the specified subnetwork. With multicasting, you send datagrams in one operation to all hosts that are members of a particular group. The member hosts can be located on the local network or on remote networks, as long as the routers are configured to support multicasting.

2.15.1 Sending Datagrams (System Services)

You can use either of the following methods to send datagrams:

- To send datagrams from the local host to one remote host, use the $QIO system service with the IO$_ACCESS function modifier. This allows you to specify the remote socket name once, and then to use the IO$_WRITEVBLK function to send each datagram without specifying the socket name again.
- To send datagrams from the local host to several remote hosts, use the $QIO system service with the IO$_WRITEVBLK function modifier, and specify the remote socket name in the **p3** argument field.

2.15.2 Sending Broadcast Datagrams (Sockets API)

You can broadcast datagrams by calling the `sendto( )` function.

2.15.3 Sending Broadcast Datagrams (System Services)

To broadcast datagrams, use a $QIO system service command with the IO$_WRITEVBLK function.

Before issuing broadcast messages, the application must issue the IO$_SETMODE function. This sets the broadcast option in the socket. The process must have a system UIC, and a SYSPRV, BYPASS, or OPER privilege to issue broadcast messages. However, the system manager can disable privilege checking with the management command SET PROTOCOL UDP /BROADCAST. For more information, refer to the *Compaq TCP/IP Services for OpenVMS Management* guide.

2.15.4 Sending Multicast Datagrams

To send IP multicast datagrams, specify the IP destination address in the range of 224.0.0.0 to 239.255.255.255 using the $QIO(IO$_WRITEVBLK) system service function or the `sendto( )` Sockets API function. Make sure you include the IN.H header file.

The system maps the specified IP destination address to the appropriate Ethernet or FDDI multicast address before it transmits the datagram.

You can control multicast options by specifying the following arguments to the `setsockopt( )` system call, as appropriate:

- IP_MULTICAST_TTL (Sockets API)
  TCPIP$C_IP_MULTICAST_TTL (OpenVMS system services)

  Time to live (TTL). Takes an integer value between 0 and 255.

Value	Result
0	Restricts distribution to applications running on the local host.
1	Forwards the multicast datagram to hosts on the local subnet.
1—255	With a multicast router attached to the sending host's network, forwards multicast datagrams beyond the local subnet. Multicast routers forward the datagram to known networks that have hosts belonging to the specified multicast group. The TTL value is decremented by each multicast router in the path. When the TTL value reaches 0, the datagram is no longer forwarded.

For example:

```
u_char ttl;
ttl=2;

if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl,
        sizeof(ttl)) == -1)
 perror("setsockopt");
```

- IP_MULTICAST_IF (Sockets API)
  TCPIP$C_MULTICAST_IF (OpenVMS system services)

  Multicast interface. Specifies a network interface other than that specified by the route in the kernel routing table.

  Unless the application specifies that an alternate network interface is associated with the socket, the datagram addressed to an IP multicast destination is transmitted from the default network interface. The default interface is determined by the interface associated with the default route in the kernel routing table or by the interface associated with an explicit route, if one exists.

  For example:

```
int sock;

struct in_addr ifaddress;

char *if_to_use = "16.141.64.251";
.
.
.
  ifaddress.s_addr = inet_addr(if_to_use);

  if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF, &ifaddress,
                               sizeof(ifaddress)) == -1)
     perror ("error from setsockopt IP_MULTICAST_IF");

  else
     printf ("new interface set for sending multicast datagrams\n");
```

- IP_MULTICAST_LOOP (Sockets API)
  TCPIP$C_MULTICAST_LOOP (OpenVMS system services)

  Disables loopback of local delivery. If a multicast datagram is sent to a group of which the sending host is a member, a copy of the datagram is looped back by the IP layer for local delivery (default). To disable loopback delivery, specify the loop value as 0.

  For example:

```
u_char loop=0;
if (setsockopt( sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop
        sizeof(loop)) == -1)
 perror("setsockopt");
```

  To enable loopback delivery, specify a `loop` value of 1. For improved performance, Compaq recommends that you disable the default unless the host must receive copies of the datagrams.

2.16 Using the Berkeley Internet Name Domain Service

The Berkeley Internet Name Domain (BIND) service is a host name and address lookup service for the Internet. If BIND is enabled on your system, you can make a call to the BIND resolver to obtain host names and addresses.

Typically, you make a call to the BIND resolver either before you bind a socket or before you make a connection to a socket. You can also use this service to translate either the local or remote host name to an address before making a connection.

2.16.1 BIND Lookups (Sockets API)

If the BIND resolver is enabled on your system and the host name is not found in the local database, you can use either of the following functions to search the BIND database:

- `gethostbyaddr( )` gets a host record from the local host or BIND database when given the host address.
- `gethostbyname( )` gets a host record from the local host or BIND database when given the host name.

The host record contains both name and address information.

Example 2–24 shows how to use the `gethostname( )`, `gethostbyname( )`, and `gethostbyaddr( )` functions to find a local host name and address.

Example 2–24 BIND Lookup (Sockets API)

```
#include <in.h>                     /* define internet related constants,   */
                                    /* functions, and structures            */
#include <inet.h>                   /* define network address info          */
#include <netdb.h>                  /* define network database library info */
#include <stdio.h>                  /* define standard i/o functions        */
#include <stdlib.h>                 /* define standard library functions    */

int main( void )
{
    char host[1024];
    struct in_addr addr;
    struct hostent *hptr;
    /*
     * get name of local host
     */
    if ( (gethostname(host, sizeof(host))) < 0 )  1
        {
        perror( "Failed to get host's local name" );
        exit( EXIT_FAILURE );
        }

    printf( "Local hostname: %s\n", host );

    /*
     * lookup local host record by name
     */
```

(continued on next page)

Example 2–24 (Cont.) BIND Lookup (Sockets API)

```
    if ( !(hptr = gethostbyname(host)) )   2
        {
        perror( "Failed to find record for local host" );
        exit( EXIT_FAILURE );
        }

    addr.s_addr = *(int *) hptr->h_addr;
    printf( "Official hostname: %s  address: %s\n",
            hptr->h_name, inet_ntoa(addr) );

    /*
     * lookup local host record by address
     */

    hptr = gethostbyaddr( &addr.s_addr, sizeof(addr.s_addr), AF_INET );  3
    if ( !hptr )
        {
        perror( "Failed to find record for local host" );
        exit( EXIT_FAILURE );
        }

    printf( "Back-translated hostname: %s\n", hptr->h_name );

    exit( EXIT_SUCCESS );
}
```

In this example, the following functions and arguments were used to find a local host name and address:

1 `gethostname( )` gets the local host name.

`host` is the address of the buffer that receives the host name.

`sizeof(host)` is the size of the buffer that receives the host name.

2 `gethostbyname( )` looks for the host record that has the specified name.

On successful return of the `gethostbyname( )` function, `hptr` receives the address of a `hostent` structure containing the host name, alias names, host address type, length of address (4 or 16), and an array of IPv4 addresses of the host being sought.

3 `gethostbyaddr( )` looks for the host record that has the specified address.

`addr.s_addr` specifies the address of the host being sought. It points to a series of bytes in network order, not to an ASCII string.

`sizeof(addr.s_addr)` specifies the number of bytes in the address to which the first argument points.

`AF_INET` points to the supported address family.

2.16.2 BIND Lookups (System Services)

If BIND is enabled on your system, the IO$_ACPCONTROL function searches the BIND database for the host name if it does not find the name in the local host database. The **p1** argument allows you to specify the `gethostbyaddr( )` or `gethostbyname( )` network ACP subfunctions to control how the function searches the database.

Example 2–25 shows how to use OpenVMS system services to find a host name and address.

Example 2–25 BIND Lookup (System Services)

```
#include <descrip.h>                /* define OpenVMS descriptors           */
#include <efndef.h>                 /* define 'EFN$C_ENF' event flag        */
#include <in.h>                     /* define internet related constants,   */
                                    /* functions, and structures            */
#include <inet.h>                   /* define network address info          */
#include <iodef.h>                  /* define i/o function codes            */
#include <netdb.h>                  /* define network database library info */
#include <ssdef.h>                  /* define system service status codes   */
#include <starlet.h>                /* define system service calls          */
#include <stdio.h>                  /* define standard i/o functions        */
#include <stdlib.h>                 /* define standard library functions    */
#include <string.h>                 /* define string handling functions     */
#include <stsdef.h>                 /* define condition value fields        */
#include <tcpip$inetdef.h>          /* define tcp/ip network constants,     */
                                    /* structures, and functions            */

struct iosb
    {                                   /* i/o status block                 */
    unsigned short status;              /* i/o completion status            */
    unsigned short bytcnt;              /* bytes transferred if read/write  */
    void *details;                      /* address of buffer or parameter   */
    };

struct acpfunc
    {                                   /* acp subfunction                  */
    unsigned char code;                 /* subfunction code                 */
    unsigned char type;                 /* call code                        */
    unsigned short reserved;            /* reserved (must be zero)          */
    };

int main( void )
{
    char host[1024];
    char hostent[2048];
    struct in_addr addr;
    struct hostent *hptr;

    struct iosb iosb;                   /* i/o status block                 */
    unsigned int status;                /* system service return status     */
    unsigned short channel;             /* network device i/o channel       */

    struct acpfunc func_byaddr =        /* acp gethostbyaddr function code  */
        { INETACP_FUNC$C_GETHOSTBYADDR, INETACP$C_HOSTENT_OFFSET, 0 };

    struct acpfunc func_byname =        /* acp gethostbyname function code  */
        { INETACP_FUNC$C_GETHOSTBYNAME, INETACP$C_HOSTENT_OFFSET, 0 };

    struct dsc$descriptor p1_dsc =      /* acp function descriptor          */
        { 0, DSC$K_CLASS_S, DSC$K_DTYPE_T, 0 };

    struct dsc$descriptor p2_dsc =      /* acp p2 argument descriptor       */
        { 0, DSC$K_CLASS_S, DSC$K_DTYPE_T, 0 };

    struct dsc$descriptor p4_dsc =      /* acp p4 argument descriptor       */
        { 0, DSC$K_CLASS_S, DSC$K_DTYPE_T, 0 };

    $DESCRIPTOR( inet_device,           /* string descriptor with logical   */
                 "TCPIP$DEVICE:" );     /* name of network pseudodevice     */
```

(continued on next page)

Example 2–25 (Cont.) BIND Lookup (System Services)

```
/*
 * get name of local host
 */

if ( (gethostname(host, sizeof(host))) < 0 )
    {
    perror( "Failed to get host's local name" );
    exit( EXIT_FAILURE );
    }

printf( "Local hostname: %s\n", host );
/*
 * assign i/o channel to network device
 */

status = sys$assign( &inet_device,      /* device name                  */
                     &channel,          /* i/o channel                  */
                     0,                 /* access mode                  */
                     0                  /* not used                     */
                   );

if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to assign i/o channel\n" );
    exit( status );
    }

/*
 * lookup local host record by name
 */

p1_dsc.dsc$w_length  = sizeof(func_byname);
p1_dsc.dsc$a_pointer = (char *) &func_byname;

p2_dsc.dsc$w_length  = strlen( host );
p2_dsc.dsc$a_pointer = host;

p4_dsc.dsc$w_length  = sizeof(hostent);
p4_dsc.dsc$a_pointer = hostent;

status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                   channel,             /* i/o channel                  */
                   IO$_ACPCONTROL,      /* i/o function code            */
                   &iosb,               /* i/o status block             */
                   0,                   /* ast service routine          */
                   0,                   /* ast parameter                */
                   &p1_dsc,             /* p1 - acp subfunction code    */
                   &p2_dsc,             /* p2 - hostname to lookup      */
                   &p4_dsc.dsc$w_length,/* p3 - return length address   */
                   &p4_dsc,             /* p4 - output buffer address   */
                   0,                   /* p5                           */
                   0                    /* p6                           */
                 );

if ( status & STS$M_SUCCESS )
    status = iosb.status;

if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to find record for local host\n" );
    exit( status );
    }
```

(continued on next page)

Example 2–25 (Cont.) BIND Lookup (System Services)

```
    hptr = (struct hostent *) hostent;
    hptr->h_name += (unsigned int) hptr;
    *(char **) &hptr->h_addr_list += (unsigned int) hptr;
    *(char **) hptr->h_addr_list  += (unsigned int) hptr;

    addr.s_addr = *(int *) hptr->h_addr;
    printf( "Official hostname: %s  address: %s\n",
            hptr->h_name, inet_ntoa(addr) );

    /*
     * lookup local host record by address
     */

    p1_dsc.dsc$w_length  = sizeof(func_byaddr);
    p1_dsc.dsc$a_pointer = (char *) &func_byaddr;

    p2_dsc.dsc$w_length  = strlen( inet_ntoa(addr) );
    p2_dsc.dsc$a_pointer = inet_ntoa( addr );

    p4_dsc.dsc$w_length  = sizeof(hostent);
    p4_dsc.dsc$a_pointer = hostent;

    status = sys$qiow( EFN$C_ENF,           /* event flag                  */
                       channel,             /* i/o channel                 */
                       IO$_ACPCONTROL,      /* i/o function code           */
                       &iosb,               /* i/o status block            */
                       0,                   /* ast service routine         */
                       0,                   /* ast parameter               */
                       &p1_dsc,             /* p1 - acp subfunction code   */
                       &p2_dsc,             /* p2 - ip address to lookup   */
                       &p4_dsc.dsc$w_length,/* p3 - return length address  */
                       &p4_dsc,             /* p4 - output buffer address  */
                       0,                   /* p5                          */
                       0                    /* p6                          */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to find record for local host\n" );
        exit( status );
        }

    hptr = (struct hostent *) hostent;
    hptr->h_name += (unsigned int) hptr;

    printf( "Back-translated hostname: %s\n", hptr->h_name );

    /*
     * deassign i/o channel to network device
     */

    status = sys$dassgn( channel );
    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to deassign i/o channel\n" );
        exit( status );
        }

    exit( EXIT_SUCCESS );
}
```

2.17 Closing and Deleting a Socket

Closing a socket means that the program can no longer transmit data. Depending on how you close the socket, the program can receive data until the peer program also closes the socket.

When a remote system closes a socket, notification is not immediate, and another thread can erroneously attempt to use the socket.

If you send data to a closed socket, you might not receive an appropriate error message. Set the TCPIP$FULL_DUPLEX_CLOSE socket option if you want to have your application notified of an error when it sends data on a socket that has already been closed by the peer.

When you delete a socket, all pending messages queued for transmission are sent to the receiving socket before closing the connection.

2.17.1 Closing and Deleting (Sockets API)

Example 2–26 shows a TCP application using the `close( )` function to close and delete a socket.

Example 2–26 Closing and Deleting a Socket (Sockets API)

```
#include <socket.h>                /* define BSD socket api                */
#include <stdio.h>                 /* define standard i/o functions        */
#include <stdlib.h>                /* define standard library functions    */
#include <unixio.h>                /* define unix i/o                      */

int main( void )
{
    int sockfd;

    /*
     * create a socket
     */

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }

    /*
     * close socket
     */

    if ( close(sockfd) < 0 )  1
        {
        perror( "Failed to close socket" );
        exit( EXIT_FAILURE );
        }

    exit( EXIT_SUCCESS );
}
```

1 The `sockfd` argument for the `close( )` function closes the socket and deletes the socket descriptor previously defined by the `socket( )` function.

2.17.2 Closing and Deleting (System Services)

Make the following calls to close and delete a socket:

1. $QIO(IO$_DEACCESS) — Stops transmitting data and closes the socket.
2. $DASSGN — Deletes the network device and deassigns the I/O channel previously acquired with the $ASSIGN service.

Example 2–27 shows a TCP application using OpenVMS system services to close and delete a socket.

Example 2–27 Closing and Deleting a Socket (System Services)

```
#include <descrip.h>                /* define OpenVMS descriptors           */
#include <efndef.h>                 /* define 'EFN$C_ENF' event flag        */
#include <iodef.h>                  /* define i/o function codes            */
#include <ssdef.h>                  /* define system service status codes   */
#include <starlet.h>                /* define system service calls          */
#include <stdio.h>                  /* define standard i/o functions        */
#include <stdlib.h>                 /* define standard library functions    */
#include <stsdef.h>                 /* define condition value fields        */
#include <tcpip$inetdef.h>          /* define tcp/ip network constants,     */
                                    /* structures, and functions            */

struct iosb
    {                                   /* i/o status block                 */
    unsigned short status;              /* i/o completion status            */
    unsigned short bytcnt;              /* bytes transferred if read/write  */
    void *details;                      /* address of buffer or parameter   */
    };

struct sockchar
    {                                   /* socket characteristics           */
    unsigned short prot;                /* protocol                         */
    unsigned char type;                 /* type                             */
    unsigned char af;                   /* address format                   */
    };

int main( void )
{
    struct iosb iosb;                   /* i/o status block                 */
    unsigned int status;                /* system service return status     */
    unsigned short channel;             /* network device i/o channel       */
    struct sockchar sockchar;           /* socket characteristics buffer    */
    $DESCRIPTOR( inet_device,           /* string descriptor with logical   */
                 "TCPIP$DEVICE:" );     /* name of network pseudodevice     */

    /*
     * initialize socket characteristics
     */

    sockchar.prot = TCPIP$C_TCP;
    sockchar.type = TCPIP$C_STREAM;
    sockchar.af   = TCPIP$C_AF_INET;

    /*
     * assign i/o channel to network device
     */
```

(continued on next page)

Example 2–27 (Cont.) Closing and Deleting a Socket (System Services)

```
    status = sys$assign( &inet_device,      /* device name                  */
                         &channel,          /* i/o channel                  */
                         0,                 /* access mode                  */
                         0                  /* not used                     */
                       );
    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to assign i/o channel\n" );
        exit( status );
        }
    /*
     * create a socket
     */
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       channel,             /* i/o channel                  */
                       IO$_SETMODE,         /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       &sockchar,           /* p1 - socket characteristics  */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );
    if ( status & STS$M_SUCCESS )
        status = iosb.status;
    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to create socket\n" );
        exit( status );
        }
    /*
     * close socket
     */
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       channel,             /* i/o channel                  */
                       IO$_DEACCESS,        /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );
    if ( status & STS$M_SUCCESS )
        status = iosb.status;
```

(continued on next page)

Example 2–27 (Cont.) Closing and Deleting a Socket (System Services)

```
    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to close socket\n" );
        exit( status );
        }

    /*
     * deassign i/o channel to network device
     */

    status = sys$dassgn( channel );
    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to deassign i/o channel\n" );
        exit( status );
        }

    exit( EXIT_SUCCESS );
}
```

2.18 Shutting Down Sockets

You can shut down a socket before closing and deleting it. The shutdown operation allows you to shut down communication one process at a time. This maintains unidirectional rather than the normal bidirectional connections, allowing you to shut down communications on receive or transmit data queues, or both. For example, if you no longer want to transmit data but want to continue receiving data, shut down the transmit side of the socket connection and keep open the receive side.

2.18.1 Shutting Down a Socket (Sockets API)

Example 2–28 shows a TCP application using the shutdown( ) function.

Example 2–28 Shutting Down a Socket (Sockets API)

```
#include <socket.h>                /* define BSD socket api               */
#include <stdio.h>                 /* define standard i/o functions       */
#include <stdlib.h>                /* define standard library functions   */
#include <unixio.h>                /* define unix i/o                     */

int main( void )
{
    int sockfd;

    /*
     * create a socket
     */

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }
```

(continued on next page)

Example 2–28 (Cont.) Shutting Down a Socket (Sockets API)

```
    /*
     * shutdown a socket
     */

    if ( shutdown(sockfd,1  2 2 ) < 0 )
        {
        perror( "Failed to shutdown socket connections" );
        exit( EXIT_FAILURE );
        }

    /*
     * close socket
     */

    if ( close(sockfd) < 0 )
        {
        perror( "Failed to close socket" );
        exit( EXIT_FAILURE );
        }

    exit( EXIT_SUCCESS );
}
```

1 sockfd specifies the socket descriptor for the socket being shut down.

Other valid socket descriptor values are:

- 1 closes the receive socket queue.
- 3 closes both the transmit and receive socket queues.

The close( ) function then closes the socket and deletes the socket descriptor.

2.18.2 Shutting Down a Socket (System Services)

To shut down a socket, use the IO$_DEACCESS function with the IO$M_SHUTDOWN function modifier. This function shuts down all or part of the full-duplex connection on the socket.

The application uses subfunctions or flags to specify whether pending I/O operations are completed or discarded before the IO$_DEACCESS function completes. After the IO$_DEACCESS function completes, messages can no longer be transmitted or received.

Example 2–29 shows a TCP server using the IO$_DEACCESS function with the IO$M_SHUTDOWN function modifier to shut down all communications. In this example, no data is received or transmitted and all queued data is discarded.

Example 2–29 Shutting Down a Socket (System Services)

```
#include <descrip.h>                /* define OpenVMS descriptors           */
#include <efndef.h>                 /* define 'EFN$C_ENF' event flag        */
#include <iodef.h>                  /* define i/o function codes            */
#include <ssdef.h>                  /* define system service status codes   */
#include <starlet.h>                /* define system service calls          */
#include <stdio.h>                  /* define standard i/o functions        */
#include <stdlib.h>                 /* define standard library functions    */
#include <stsdef.h>                 /* define condition value fields        */
#include <tcpip$inetdef.h>          /* define tcp/ip network constants,     */
                                    /* structures, and functions            */

struct iosb
    {                                   /* i/o status block                 */
    unsigned short status;              /* i/o completion status            */
    unsigned short bytcnt;              /* bytes transferred if read/write  */
    void *details;                      /* address of buffer or parameter   */
    };

struct sockchar
    {                                   /* socket characteristics           */
    unsigned short prot;                /* protocol                         */
    unsigned char type;                 /* type                             */
    unsigned char af;                   /* address format                   */
    };

int main( void )
{
    struct iosb iosb;                   /* i/o status block                 */
    unsigned int status;                /* system service return status     */
    unsigned short channel;             /* network device i/o channel       */
    struct sockchar sockchar;           /* socket characteristics buffer    */
    $DESCRIPTOR( inet_device,           /* string descriptor with logical   */
                 "TCPIP$DEVICE:" );     /* name of network pseudodevice     */

    /*
     * initialize socket characteristics
     */

    sockchar.prot = TCPIP$C_TCP;
    sockchar.type = TCPIP$C_STREAM;
    sockchar.af   = TCPIP$C_AF_INET;

    /*
     * assign i/o channel to network device
     */

    status = sys$assign( &inet_device,      /* device name                  */
                         &channel,          /* i/o channel                  */
                         0,                 /* access mode                  */
                         0                  /* not used                     */
                       );

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to assign i/o channel\n" );
        exit( status );
        }

    /*
     * create a socket
     */
```

(continued on next page)

Example 2–29 (Cont.) Shutting Down a Socket (System Services)

```
    status = sys$qiow( EFN$C_ENF,           /* event flag                    */
                       channel,             /* i/o channel                   */
                       IO$_SETMODE,         /* i/o function code             */
                       &iosb,               /* i/o status block              */
                       0,                   /* ast service routine           */
                       0,                   /* ast parameter                 */
                       &sockchar,           /* p1 - socket characteristics   */
                       0,                   /* p2                            */
                       0,                   /* p3                            */
                       0,                   /* p4                            */
                       0,                   /* p5                            */
                       0                    /* p6                            */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to create socket\n" );
        exit( status );
        }

    /*
     * shutdown a socket
     */

    status = sys$qiow( EFN$C_ENF,           /* event flag                    */
                       channel,             /* i/o channel                   */
                       IO$_DEACCESS|IO$M_SHUTDOWN,
                                            /* i/o function code             */
                       &iosb,               /* i/o status block              */
                       0,                   /* ast service routine           */
                       0,                   /* ast parameter                 */
                       0,                   /* p1                            */
                       0,                   /* p2                            */
                       0,                   /* p3                            */
                       TCPIP$C_DSC_ALL,     /* p4 - discard all packets      */
                       0,                   /* p5                            */
                       0                    /* p6                            */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to shutdown socket connections\n" );
        exit( status );
        }

    /*
     * close socket
     */
```

(continued on next page)

Example 2–29 (Cont.) Shutting Down a Socket (System Services)

```
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       channel,             /* i/o channel                  */
                       IO$_DEACCESS,        /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to close socket\n" );
        exit( status );
        }

    /*
     * deassign i/o channel to network device
     */

    status = sys$dassgn( channel );

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to deassign i/o channel\n" );
        exit( status );
        }

    exit( EXIT_SUCCESS );
}
```

2.19 Canceling I/O Operations

The $CANCEL system service cancels pending I/O requests on a specific channel or socket. This includes all I/O requests queued and in progress.

There is no Sockets API function for this operation; the Sockets API library functions are synchronous.

3
Using the Sockets API

This chapter contains information to help you increase the portability of the network application programs that you write using the TCP/IP Services implementation of the Sockets API.

3.1 Internet Protocols

The IP (Internet Protocol) family is a collection of protocols on the Transport layer that use the internet address format. This section describes TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) sockets.

3.1.1 TCP Sockets

TCP provides reliable, flow-controlled, two-way transmission of data. A byte-stream protocol used to support the SOCK_STREAM abstraction, TCP uses the standard IP address format and provides a per-host collection of **port addresses**. Thus, each address consists of an internet address specifying the host and network, with a specific TCP port on the host identifying the peer entity.

Sockets using TCP are either **active** or **passive**, as described in Table 3–1.

Table 3–1 TCP Socket Types

Socket Type	Description
Active	Initiates connections to passive sockets. By default, TCP sockets are active.

Active sockets use the connect( ) function to initiate connections.)

Passive	Listens for connection requests from active sockets. To create a passive socket, use the bind( ) function and then the listen( ) function. Passive sockets use the accept( ) function to accept incoming connections. If the server is running on a multihomed system, you can specify **wildcard addressing**. Wildcard addressing allows a single server to provide service to clients on multiple networks. (See Section 3.1.1.1.)

3.1.1.1 Wildcard Addressing

When a server is running on a host that has more than one network interface installed, you can use wildcard addressing to configure it to accept incoming connections on all the interfaces.

The wildcard address is the any-interface choice. You specify this address by setting the IP address in the socket address structure to INADDR_ANY before calling the bind( ) function.

To create a socket that listens to all hosts on any network interface, perform these steps:

1. Bind the internet address INADDR_ANY.
2. Specify the TCP port.

   If you do not specify the port, the system assigns a unique port, starting at port number 49152. Once connected, the socket's address is fixed by the peer entity's location.

The address assigned to the socket is the address associated with the network interface through which packets from the peer are being transmitted and received. This address corresponds to the peer entity's network.

TCP supports the setting of socket options with the setsockopt( ) function and the checking of current option settings with the getsockopt function. Under most circumstances, TCP sends data when it is presented. When outstanding data has not been acknowledged, TCP gathers small amounts of output and sends it in a single packet when an acknowledgment is received.

For a small number of clients, such as window systems that send a stream of mouse events that receive no replies, this packetization can cause significant delays. Therefore, TCP provides a Boolean option, TCP_NODELAY (from TCP.H), to defeat this algorithm. The option level for the setsockopt( ) function is the protocol number for TCP, which is available from getprotobyname( ). In this situation, servers may want to use TCP_NODELAY; however, network traffic may increase significantly as a result.

3.1.2 UDP Sockets

UDP is a protocol that supports the SOCK_DGRAM abstraction for the internet protocol family. UDP sockets are connectionless and are normally used with the sendto( ) and recvfrom( ) functions. You can also use the connect( ) function to establish the destination address for future datagrams; then you use the read( ), write( ), send( ), rec( ), or recv( ) function to transmit or receive datagrams.

UDP address formats are identical to those used by TCP. In particular, UDP provides a port identifier in addition to the normal internet address format. Note that the UDP port space is separate from the TCP port space (for example, a UDP port cannot be connected to a TCP port). Also, you can send broadcast packets (assuming the underlying network supports this) by using a reserved broadcast address. This address is network-interface dependent. The SO_BROADCAST option must be set on the socket, and the process must have a privileged UIC or the SYSPRV, BYPASS, or OPER privilege for broadcasting to succeed.

3.2 Structures

This section describes, in alphabetical order, the structures you supply as arguments to the various Sockets API functions. Table 3–2 lists these structures.

Table 3–2 Structures for Sockets API

Structure	Description
hostent	This structure holds a canonical host name, alias names, a host address type, the length of the address, and a pointer to a list of host addresses. This structure is a parameter value for host name and address lookup functions.
in_addr	This structure holds a 32-bit IPv4 address stored in network byte order.
iovec	This structure holds the beginning address and length of an I/O buffer.
linger	This structure holds option information for the close function.
msghdr	This structure holds the protocol address, the size of the protocol address, a scatter-and-gather array, the number of elements in the scatter-and-gather array, ancillary data, the length of the ancillary data, and returned flags. The structure is a parameter of the recvmsg( ) and sendmsg( ) functions.
netent	This structure holds a network name, a list of aliases associated with the network, and the network number.
sockaddr	The socket functions use this generic socket address structure to function with any of the supported protocol families.
sockaddr_in	This IPv4 socket address structure holds the length of the structure, the address family, either a TCP or a UDP port number, and a 32-bit IPv4 address stored in network byte order. The structure has a fixed length of 16 bytes.
timeval	This structure holds a time interval specified in seconds and microseconds.

3.2.1 hostent Structure

The hostent structure, defined in the NETDB.H header file, holds a host name, a list of aliases associated with the network, and the network's number as specified in an internet address from the hosts database.

The hostent structure definition is as follows:

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

The hostent structure members are as follows:

- h_name is a pointer to a null-terminated character string that is the official (canonical) name of the host.
- h_aliases is a pointer to an array of pointers to alias names for the host.
- h_addrtype is the type of host address being returned (AF_INET).
- h_length is the length, in bytes, of the address. (For IPv4, this value is 4 bytes.)
- h_addr_list is a pointer to an array of pointers to the network addresses for the host. Each host address is represented by a series of bytes in network order. The list is terminated with a null pointer value.

- h_addr is the first address in the h_addr_list.

3.2.2 in_addr Structure

The in_addr structure, defined in the IN.H header file, holds an internet address. The address format can be any of the supported internet address notation formats.

The in_addr structure definition is as follows:

```
struct in_addr {
       union {
                struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
                struct { u_short s_w1,s_w2; } S_un_w;
                u_long S_addr;
}      S_un;
#define s_addr  S_un.S_addr            /* can be used for most tcp & ip code */
#define s_host  S_un.S_un_b.s_b2       /* host on imp                        */
#define s_net   S_un.S_un_b.s_b1       /* network                            */
#define s_imp   S_un.S_un_w.s_w2       /* imp                                */
#define s_impno S_un.S_un_b.s_b4       /* imp #                              */
#define s_lh    S_un.S_un_b.s_b3       /* logical host                       */
};
```

3.2.3 iovec Structure

The iovec structure holds one scatter-and-gather buffer. Multiple scatter-and-gather buffer descriptors are stored as an array of iovec elements.

The iovec structure definition is defined in the SOCKET.H header file.

The iovec structure definition is as follows:

```
struct iovec {
       char *iov_base;
       int  iov_len;
}
```

The iovec structure members are as follows:

- iov_base is a pointer to a buffer.
- iov_len contains the size of the buffer to which iov_base points.

3.2.4 linger Structure

The linger structure, defined in the SOCKET.H header file, specifies the setting or resetting of the socket option for the time interval that the socket lingers for data. The linger structure is supported only by STREAM-type sockets.

The linger structure definition is as follows:

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

The linger structure members are as follows:

- l_onoff=1 sets linger; l_onoff=0 resets linger.
- l_linger is the number of seconds to linger. (The default is 120 seconds, or 2 minutes.)

3.2.5 msghdr Structure

The msghdr structure specifies the buffer parameter for the recvmsg and sendmsg I/O functions. The structure allows you to specify an array of scatter and gather buffers. The recvmsg function scatters the data to several user receive buffers, and the sendmsg function gathers data from several user transmit buffers before being transmitted.

The SOCKET.H header file defines the following structures for BSD Versions 4.3 and 4.4:

- msghdr structure (BSD Version 4.4)
- omsghdr structure (BSD Version 4.3)

3.2.5.1 BSD Version 4.4

The msghdr structure definition for use with BSD Version 4.4 is as follows:

```
struct msghdr {
       void         *msg_name;       /* protocol address               */
       int          msg_namelen;     /* size of address                */
       struct iovec *msg_iov;        /* scatter/gather array           */
       int          msg_iovlen;      /* number of elements in msg_iov  */
       void         *msg_control;    /* ancillary data; must be aligned
                                        for a cmsghdr structure        */
       int          msg_controllen;  /* length of ancillary data buffer */
       int          msg_flags;       /* flags on received message      */
};
```

The msghdr structure members are as follows:

- msg_name is the address of the destination socket if the socket is not connected. If no address is required, you can set this field to null.
- msg_namelen is the length of the msg_name field.
- msg_iov is an array of I/O buffer pointers of the iovec structure form. See Section 3.2.3 for a description of the iovec structure.
- msg_iovlen is the number of buffers in the msg_iov array.
- msg_control specifies the location of the optional ancillary data or control information.
- msg_controllen is the size of the ancillary data in the msg_control buffer.
- mgs_flags, used only with the recvmsg function, is the value used by the kernel to drive its receive processing.

3.2.5.2 BSD Version 4.3

The omsghdr structure definition for use with BSD Version 4.3 is as follows:

```
struct omsghdr {
       char          *msg_name;         /* protocol address              */
       int           msg_namelen;       /* size of address               */
       struct iovec  *msg_iov;          /* scatter/gather array          */
       int           msg_iovlen;        /* number of elements in msg_iov  */
       char          *msg_accrights;    /* access rights sent/received   */
       int           msg_accrightslen;  /* length of access rights buffer */
};
```

The omsghdr structure members are as follows:

- msg_name is the address of the destination socket if the socket is not connected. If no address is required, you can set this field to null.
- msg_namelen is the length of the msg_name field.
- msg_iov is an array of I/O buffer pointers of the iovec structure form. See Section 3.2.3 for a description of the iovec structure.
- msg_iovlen is the number of buffers in the msg_iov array.
- msg_accrights points to a buffer containing access rights sent with the message.
- msg_accrightslen is the length of the msg_accrights buffer.

3.2.6 netent Structure

The netent structure, defined in the NETDB.H header file, holds a network name, a list of aliases associated with the network, and the network's number specified as an internet address from the network database.

The netent structure definition is as follows:

```
struct  netent {
        char    *n_name;        /* official name of net */
        char    **n_aliases;    /* alias list           */
        int     n_addrtype;     /* net address type     */
        long    n_net;          /* net number           */
};
```

The netent structure members are as follows:

- n_name is the official network name.
- n_aliases is a null-terminated list of pointers to alternate names for the network.
- n_addrtype is the type of the network number returned (AF_INET).
- n_net is the network number returned in host byte order.

3.2.7 sockaddr Structure

The sockaddr structure, defined in the SOCKET.H header file, holds a general address family.

The SOCKET.H header file defines the following structures for BSD Versions 4.3 and 4.4:

- sockaddr structure (BSD Version 4.4)
- osockaddr structure (BSD Version 4.3)

3.2.7.1 BSD Version 4.4

The sockaddr structure definition for use with BSD Version 4.4 is as follows:

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

The sockaddr structure members are as follows:

- sa_len is the length of the structure.

- sa_family is the address family or domain in which the socket was created.
- sa_data is the data string of up to 14 bytes of direct address.

3.2.7.2 BSD Version 4.3

The osockaddr structure definition for use with BSD Version 4.3 is as follows:

```
   struct osockaddr {
          u_short sa_family;     /* address family                   */
          char    sa_data[14];   /* up to 14 bytes of direct address */
};
```

The osockaddr structure members are as follows:

- sa_family is the address family or domain in which the socket was created.
- sa_data is the data string of up to 14 bytes of direct address.

3.2.8 sockaddr_in Structure

The sockaddr_in structure, defined in the IN.H header file, specifies an internet address family.

The sockaddr_in structure definition is as follows:

```
struct sockaddr_in {
        short   sin_family;           /* address family             */
        u_short sin_port;             /* port number                */
        struct  in_addr sin_addr;     /* internet address           */
        char    sin_zero[8];          /* 8-byte field of all zeroes */
};
```

The sockaddr_in structure members are as follows:

- sin_family is the address family (AF_INET).
- sin_port is the port number in network order.
- sin_addr is the internet address in network order.
- sin_zero is an 8-byte field containing all zeros.

3.2.9 timeval Structure

The timeval structure, defined in the SOCKET.H header file, specifies time intervals. The timeval structure definition is as follows:

```
struct timeval {
    long tv_sec;
    long tv_usec;
};
```

The timeval structure members are as follows:

- tv_sec specifies the number of seconds to wait.
- tv_usec specifies the number of microseconds to wait.

3.3 Header Files

You can include header files on a OpenVMS system using any one of the following preprocessor directive statements:

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

The `#include types` form of the `#include` preprocessor directive is possible on OpenVMS systems because all header files are located in a text library in the SYS$LIBRARY directory. On Compaq *Tru64* UNIX systems, you must specify header files (and subdirectories that locate a header file) within angle brackets (< >) or double quotes (" ") .

For example, to include the header file TYPES.H, use the following form of the `#include` directive:

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

3.4 Calling a Socket Function from an AST State

Calls to various Sockets API functions return information within a static area. The OpenVMS environment allows an asynchronous system trap (AST) routine to interrupt a Sockets API function during its execution. In addition, the ASTs of more privileged modes can interrupt ASTs of less privileged modes. Therefore, be careful when calling a Sockets API function from an AST state while a similar Sockets API function is being called from either a non-AST state or a less-privileged access mode. You can use the SYS$SETAST system service to enable and disable the reception of AST requests.

The Sockets API functions that use a static area are:

- `gethostbyaddr( )`
- `gethostbyname( )`
- `getnetbyaddr( )`
- `getnetbyname( )`
- `getservbyname( )`
- `getservbyport( )`
- `getprotobyname( )`
- `getprotobynumber( )`

Caution

Because these Sockets API functions access files to retrieve information, you should not call these functions from either the KERNEL or the EXEC mode when the ASTs are disabled.

3.5 Standard I/O Functions

You cannot use standard I/O functions with the Sockets API. Specifically, the `fdopen( )` function does not support sockets.

3.6 Event Flags

Socket functions can use event flags during their operation. To assign event flags, use the library function LIB$GET_EF. Event flags are released when the function no longer needs them.

3.7 Error Checking: errno Values

Most Sockets API functions return a value that indicates whether the function was successful or unsuccessful. A return value of zero (0) indicates success, and a value of –1 indicates the function was unsuccessful.

If the function is not successful, it stores an additional value in the external variable `errno`. The value stored in `errno` is valid only when the function is not successful. The error codes are defined in the ERRNO.H header file.

All function return codes and error values are of type `integer` unless otherwise noted.

The `errno` values can be translated to a message similar to those found on UNIX systems by using the `perror( )` function. The `perror( )` function writes a message on the standard error stream that describes the current setting of the external variable `errno`. The error message includes a character string containing the name of the function that caused the error followed by a colon (:), a blank space, the system message string, and a newline character.

Table 3–3 lists the possible `errno` values.

Table 3–3 errno Values

Error	Description
EADDRINUSE	Address already in use. Each address can be used only once.
EADDRNOTAVAIL	Cannot assign requested address. Normally, these values result from an attempt to create a socket with an address not on this machine.
EAFNOSUPPORT	Address family not supported by protocol family. An address incompatible with the requested protocol was used.
EALREADY	Operation already in progress. An operation was attempted on a nonblocking object that already had an operation in progress.
ECONNABORTED	Software caused connection abort. Indicates that the software caused a connection abort because there is no space on the socket's queue and the socket cannot receive further connections. A connection abort occurred internal to your host machine.
ECONNREFUSED	Connection refused.

(continued on next page)

Table 3–3 (Cont.) errno Values

Error	Description
	No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on a foreign host.
ECONNRESET	Connection reset by peer.
	A connection was forcibly closed by a peer. This usually results from the peer executing a `shutdown( )` call.
EDESTADDRREQ	Destination address required.
	A required address was omitted from an operation on a socket.
EHOSTDOWN	Host is down.
	A socket operation failed because the destination host was down.
EHOSTUNREACH	No route to host.
	A socket operation to an unreachable host was attempted.
EINPROGRESS	Operation now in progress.
	An operation that takes a long time to complete, such as `connect( )`, was attempted on a nonblocking object.
EISCONN	Socket is already connected.
	A `connect( )` request was made on a socket that was already connected, or a `sendto( )` or `sendmsg( )` request on a connected socket specified a destination other than the connected party.
	A path name lookup involved more than eight symbolic links.
EMSGSIZE	Message too long.
	A message sent on a socket was larger than the internal message buffer.
ENETDOWN	Network is down.
	A socket operation encountered a dead network.
ENETRESET	Network dropped connection on reset.
	The host you were connected to failed and rebooted.
ENETUNREACH	Network is unreachable.
	A socket operation to an unreachable network was attempted.
ENOBUFS	No buffer space available.
	An operation on a socket or pipe was not performed because the system lacked sufficient buffer space.
ENOPROTOOPT	Protocol not available.
	A bad option was specified in a `getsockopt( )` or `setsockopt( )` call.
ENOTSOCK	Socket operation on a nonsocket.
ENTOTCONN	Socket is not connected.

(continued on next page)

Table 3–3 (Cont.) errno Values

Error	Description
	Request to send or receive data was not allowed because the socket is not connected.
EOPNOTSUPP	Operation not supported. For example, trying to accept a connection on a datagram socket.
EPFNOSUPPORT	Protocol family not supported. The protocol family was not configured into the system or no implementation for it exists.
EPROTONOSUPPORT	Protocol not supported. The protocol was not configured into the system or no implementation for it exists.
EPROTOTYPE	Protocol wrong type for socket. A protocol was specified that does not support the semantics of the socket type requested. For example you cannot use the ARPA Internet UDP protocol with type SOCK_STREAM.
ESHUTDOWN	Cannot send after socket shutdown. A request to send data was not allowed because the socket had already been shut down with a previous shutdown( ) call.
ESOCKTNOSUPPORT	Socket type not supported. Support for the socket type was not configured into the system or no implementation for it exists.
ETIMEDOUT	Connection timed out. A connect( ) request failed because the connected party did not respond properly after a period of time. (The timeout period is dependent on the communication protocol.)
EVMSERR	OpenVMS error code is nontranslatable.

4
Sockets API Reference

This chapter describes the Sockets API functions that are listed in Table 4–1.

Table 4–1 Sockets API Functions

Function	Description
`accept( )`	Accepts a connection on a passive socket.
`bind( )`	Binds a name to a socket.
`close( )`	Closes a connection and deletes a socket descriptor.
`connect( )`	Initiates a connection on a socket.
`decc$get_sdc( )`	Returns the socket device's OpenVMS I/O channel associated with a socket descriptor (for use with Compaq C).
`gethostbyaddr( )`	Searches the hosts database for a host record with a given IPv4 address.
`gethostbyname( )`	Searches the hosts database for a host record with a given name or alias.
`gethostname( )`	Returns the fully qualified name of the local host.
`getnameinfo( )`	Gets a node name and service name for an address and port number.
`getnetbyaddr( )`	Searches the network database for a network record with a given address.
`getnetbyname( )`	Searches the network database for a network record with a given name or alias.
`getpeername( )`	Returns the name of the connected peer.
`getprotobyname( )`	Searches the protocols database until a matching protocol name is found or until end of file is encountered.
`getprotobynumber( )`	Searches the protocols database until a matching protocol number is found or until end of file is encountered.
`getprotoent( )`	Reads the next line in the protocols database.
`getservbyname( )`	Gets information on the named service from the network services database.
`getservbyport( )`	Gets information on the named port from the network services database.
`getsockname( )`	Returns the name associated with a socket.
`getsockopt( )`	Returns the options set on a socket.
`htonl( )`	Converts longwords from host byte order to network byte order.
`htons( )`	Converts short integers from host byte order to network byte order.

(continued on next page)

Table 4–1 (Cont.) Sockets API Functions

Function	Description
inet_addr( )	Converts internet addresses in text form into numeric internet addresses.
inet_lnaof( )	Returns the local network address portion of an internet address.
inet_makeaddr( )	Given a network address and a local address on that network, returns an internet address.
inet_netof( )	Returns the internet network address portion of an internet address.
inet_network( )	Converts a null-terminated text string representing an internet address into a network address in local host format.
inet_ntoa( )	Converts an internet address into an ASCII (null-terminated) string.
ioctl( )	Controls devices. Used for setting sockets for nonblocking I/O.
listen( )	Sets the maximum limit of outstanding connection requests for a TCP socket.
ntohl( )	Converts longwords from network byte order into host byte order.
ntohs( )	Converts short integers from network byte order into host byte order.
read( )	Reads bytes from a file or socket and places them into a user-defined buffer.
recv( )	Receives bytes from a connected socket and places them into a user-defined buffer.
recvfrom( )	Receives bytes for a socket from any source.
recvmsg( )	Receives bytes on a socket and places them into scattered buffers.
select( )	Allows the polling or checking of a group of sockets for I/O activity.
send( )	Sends bytes through a socket to a connected peer.
sendmsg( )	Sends gathered bytes through a socket to any other socket.
sendto( )	Sends bytes through a socket to any other socket.
setsockopt( )	Sets options on a socket.
shutdown( )	Shuts down all or part of a bidirectional connection on a socket.
socket( )	Creates an endpoint for communication by returning a socket descriptor.
write( )	Writes bytes from a buffer to a file or socket.

4.1 Sockets API

This section describes functions that comprise the Sockets API and that are supported by TCP/IP Services.

accept()

Accepts a connection on a passive socket.

The $QIO equivalent is the IO$_ACCESS function with the IO$M_ACCEPT function modifier.

Format

```
#include   <types.h>

#include   <socket.h>

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

Arguments

s
A socket descriptor returned by `socket( )`, subsequently bound to an address with `bind( )`, which is listening for connections after a `listen( )`.

addr
A result parameter filled in with the address of the connecting entity, as known to the Communications layer. The exact format of the structure to which the address parameter points is determined by the domain in which the communication is occurring. This version of Compaq C supports only the internet domain (AF_INET).

addrlen
A value/result argument. It should initially contain the size of the structure pointed to by **addr**. On return it will contain the actual length, in bytes, of the `sockaddr` structure that has been filled in by the Communications layer. See Section 3.2.7 for a description of the `sockaddr` structure.

Description

This function completes the first connection on the queue of pending connections, creates a new socket with the same properties as **s**, and allocates and returns a new descriptor for the socket. If no pending connections are present on the queue and the socket is not marked as nonblocking, `accept( )` blocks the caller until a connection request is present. If the socket is marked nonblocking by using a `setsockopt( )` call and no pending connections are present on the queue, `accept( )` returns an error. You cannot use the accepted socket to accept subsequent connections. The original socket **s** remains open (listening) for other connection requests. This call is used with connection-based socket types (`SOCK_STREAM`).

Related Functions
See also `bind( )`, `connect( )`, `listen( )`, `select( )`, and `socket( )`.

Return Values

x	A nonnegative integer that is a descriptor for the accepted socket.
-1	Error; `errno` is set to indicate the error.

Errors

EBADF	The socket descriptor is invalid.
ECONNABORTED	A connection has been aborted.
EFAULT	The **addr** argument is not in a writable part of the user address space.
EINTR	The `accept( )` function was interrupted by a signal before a valid connection arrived.
EINVAL	The socket is not accepting connections.
EMFILE	There are too many open file descriptors.
ENFILE	The maximum number of file descriptors in the system is already open.
ENETDOWN	TCP/IP Services was not started.
ENOBUFS	The system has insufficient resources to complete the call.
ENOMEM	The system was unable to allocate kernel memory.
ENOTSOCK	The socket descriptor is invalid.
EOPNOTSUPP	The reference socket is not of type `SOCK_STREAM`.
EPROTO	A protocol error occurred.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.

bind()

Binds a name to a socket.

The $QIO equivalent is the IO$_SETMODE function with the **p3** argument.

Format

```
#include   <types.h>

#include   <socket.h>

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

Arguments

s
A socket descriptor created with the `socket( )` function.

name
Address of a structure used to assign a name to the socket in the format specific to the family (AF_INET) socket address. See Section 3.2.7 for a description of the `sockaddr` structure.

namelen
The size, in bytes, of the structure pointed to by **name**.

Description

This function assigns a port number and IP address to an unnamed socket. When a socket is created with the `socket( )` function, it exists in a name space (address family) but has no name assigned. The `bind( )` function requests that a name be assigned to the socket.

Related Functions

See also `connect( )`, `getsockname( )`, `listen( )`, and `socket( )`.

Return Values

0	Successful completion.
-1	Error; `errno` is set to indicate the error.

Errors

EACCESS	The requested address is protected, and the current user has inadequate permission to access it.
EADDRINUSE	The specified internet address and ports are already in use.
EADDRNOTAVAIL	The specified address is not available from the local machine.
EAFNOSUPPORT	The specified address is invalid for the address family of the specified socket.

EBADF	The socket descriptor is invalid.
EDESTADDRREQ	The address argument is a null pointer.
EFAULT	The **name** argument is not a valid part of the user address space.
EINVAL	The socket is already bound to an address and the protocol does not support binding to a new address, the socket has been shut down, or the length or the **namelen** argument is invalid for the address family.
EISCONN	The socket is already connected.
EISDIR	The address argument is a null pointer.
ENOBUFS	The system has insufficient resources to complete the call.
ENOTSOCK	The socket descriptor is invalid.
EOPNOTSUPP	The socket type of the specified socket does not support binding to an address.

close()

Closes a connection and deletes a socket descriptor.

The $QIO equivalent is the $DASSGN system service.

Format

```
#include   <unixio.h>

int close   ( s );
```

Argument

s
A socket descriptor.

Description

This function deletes a descriptor from the per-process object (Compaq C structure) reference table. Associated TCP connections close first.

If a call to the `connect( )` function fails for a connection mode socket, applications should use `close( )` to deallocate the socket and descriptor.

Related Functions

See also `accept( )`, `socket( )`, and `write( )`.

Return Values

0	Successful completion.
-1	Error; `errno` is set to indicate the error.

Errors

EBADF	The socket descriptor is invalid.
EINTR	The `close( )` function was interrupted by a signal that was caught.

connect()

Initiates a connection on a socket.

The $QIO equivalent is the IO$_ACCESS function.

Format

```
#include   <types.h>

#include   <socket.h>

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

Arguments

s
A socket descriptor created with socket( ).

name
The address of a structure that specifies the name of the remote socket in the format specific to the address family (AF_INET).

namelen
The size, in bytes, of the structure pointed to by **name**.

Description

If **s** is a socket descriptor of type SOCK_DGRAM, then this call permanently specifies the peer where the data is sent. If **s** is of type SOCK_STREAM, then this call attempts to make a connection to the specified socket.

Each communications space interprets the **name** argument. This argument specifies the socket that is connected to the socket specified in **s**.

If the connect( ) function fails for a connection-mode socket, applications should use the close( ) function to deallocate the socket and descriptor. If attempting to reinitiate the connection, applications should create a new socket.

Related Functions

See also accept( ), select( ), socket( ), getsockname( ), and shutdown( ).

Return Values

0	Successful completion.
-1	Error; errno is set to indicate the error.

Errors

EADDRINUSE	Configuration problem. There are insufficient ports available for the attempted connection. The ipport_userreserved variable of the inet subsystem should be increased.
EADDRNOTAVAIL	The specified address is not available from the local machine.

EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
EALREADY	A connection request is already in progress for the specified socket.
EBADF	The socket descriptor is invalid.
ECONNREFUSED	The attempt to connect was rejected.
EFAULT	The **name** argument is not a valid part of the user address space.
EHOSTUNREACH	The specified host is not reachable.
EINPROGRESS	O_NONBLOCK is set for the file descriptor for the socket, and the connection cannot be immediately established; the connection will be established asynchronously.
EINTR	The `connect( )` function was interrupted by a signal while waiting for the connection to be established. Once established, the connection may continue asynchronously.
EINVAL	The value of the **namelen** argument is invalid for the specified address family, or the `sa_family` field in the socket address structure is invalid for the protocol.
EISCONN	The socket is already connected.
ELOOP	Too many symbolic links were encountered in translating the file specification in the address.
ENETDOWN	The local network connection is not operational.
ENETUNREACH	No route to the network or host is present.
ENOBUFS	The system has insufficient resources to complete the call.
ENOTSOCK	The socket descriptor is invalid.
EOPNOTSUPP	The socket is listening and cannot be connected.
EPROTOTYPE	The specified address has a different type than the socket bound to the specified peer address.
ETIMEDOUT	The connection request timed out without establishing a connection.
EWOULDBLOCK	The socket is nonblocking, and the connection cannot be completed immediately. It is possible to use the `select( )` function to select the socket for writing.

decc$get_sdc()

Returns the socket device channel (SDC) associated with a socket descriptor.

Format

```
#include   <socket.h>

short int decc$get_sdc   ( int s );
```

Argument

s
A socket descriptor.

Description

This function returns the SDC associated with a socket. Normally, socket descriptors are used either as file descriptors or with one of the functions that takes an explicit socket descriptor as its argument. Sockets are implemented using TCP/IP socket device channels. This function returns the SDC used by a given socket descriptor so you can use the TCP/IP facilities directly by means of various I/O system services ($QIO).

Return Values

0	Indicates that **s** is not an open socket descriptor.
x	The SDC number.

gethostbyaddr()

Searches the hosts database sequentially from the beginning for a host record with a given IPv4 address.

The $QIO equivalent is the IO$_ACPCONTROL function with the INETACP_FUNC$C_GETHOSTBYADDR subfunction code.

Format

```
#include   <netdb.h>

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

Arguments

addr
A pointer to a series of bytes in network order specifying the address of the host sought.

len
The number of bytes in the address pointed to by the **addr** argument.

type
The type of address format being sought. Currently, only AF_INET is supported.

Description

This function finds the first host record in the hosts database with the given address.

The gethostbyaddr( ) function uses a common static area for its return values. This means that subsequent calls to this function overwrite previously returned host entries. You must make a copy of the host entry if you want to save it.

Return Values

NULL	Indicates an error; errno is set to one of the following values.
x	A pointer to an object having the hostent structure. See Section 3.2.1 for a description of the hostent structure.

Errors

ENETDOWN	TCP/IP Services was not started.
HOST_NOT_FOUND	Host is unknown.
NO_DATA	The server recognized the request and the name, but no address is available for the name. Another type of name server request may be successful.
NO_RECOVERY	An unexpected server failure occurred. This is a nonrecoverable error.

TRY_AGAIN	A transient error occurred, for example, the server did not respond. A retry may be successful.

gethostbyname()

Searches the hosts database sequentially from the beginning for a host record with a given name or alias.

This function also invokes the BIND resolver to query the appropriate name server if the information requested is not in the hosts database.

The $QIO equivalent is the IO$_ACPCONTROL function with the INETACP_FUNC$C_GETHOSTBYNAME subfunction code.

Format

```
#include   <netdb.h>

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

Argument

name
A pointer to a null-terminated character string containing the name or an alias of the host being sought.

Description

This function finds the first host in the hosts database with the given name or alias.

The `gethostbyname( )` function uses a common static area for its return values. This means that subsequent calls to this function overwrite previously returned host entries. You must make a copy of the host entry if you want to save it.

Return Values

NULL	Indicates an error.
x	A pointer to an object having the `hostent` structure. See Section 3.2.1 for a description of the `hostent` structure.

Errors

ENETDOWN	TCP/IP Services was not started.
HOST_NOT_FOUND	Host is unknown.
NO_DATA	The server recognized the request and the name, but no address is available for the name. Another type of name server request may be successful.
NO_RECOVERY	An unexpected server failure occurred. This is a nonrecoverable error.
TRY_AGAIN	A transient error occurred, for example, the server did not respond. A retry may be successful.

gethostname()

Returns the fully qualified name of the local host.

Format

```
#include   <types.h>

#include   <socket.h>

int gethostname   ( char *name, int namelen) ;
```

Arguments

name
The address of a buffer where the name should be returned. The returned name is null terminated unless sufficient space is not provided.

namelen
The size of the buffer pointed to by **name**.

Description

This function returns the translation of the logical names TCPIP$INET_HOST and TCPIP$INET_DOMAIN when used with the TCP/IP Services software.

Return Values

0	Successful completion.
-1	Error; `errno` is set to indicate the error.

Errors

EFAULT	The buffer described by **name** and **namelen** is not a valid, writable part of the user address space.

getnetbyaddr()

Searches the network database sequentially from the beginning for a network record with a given address.

The $QIO equivalent is the IO$_ACPCONTROL function with the INETACP_FUNC$C_GETNETBYADDR subfunction code.

Format

```
#include   <netdb.h>

struct netent *getnetbyaddr   ( long net, int type) ;
```

Arguments

net
The network number, in host byte order, of the networks database entry required.

type
The type of network being sought (AF_INET).

Description

This function finds the first network record in the networks database with the given address.

The `getnetbyaddr( )` and `getnetbyname( )` functions use a common static area for their return values. Subsequent calls to any of these functions overwrite any previously returned network entry. You must make a copy of the network entry if you want to save it.

Return Values

NULL	Indicates end of file or an error.
x	A pointer to an object having the `netent` structure. See Section 3.2.6 for a description of the `netent` structure.

Errors

EINVAL	The **net** argument is invalid.
ESRCH	The search failed.

getnetbyname()

Searches the networks database sequentially from the beginning for a network record with a given name or alias.

The $QIO equivalent is the IO$_ACPCONTROL function with the INETACP_FUNC$C_GETNETBYNAME subfunction code.

Format

```
#include   <netdb.h>

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

Argument

name
A pointer to a null-terminated character string containing either the network name or an alias for the network name.

Description

This function finds the first network record in the networks database with the given name or alias.

The `getnetbyaddr( )` and `getnetbyname( )` functions use a common static area for their return values. Subsequent calls to any of these functions overwrite previously returned network entries. You must make a copy of the network entry if you want to save it.

Return Values

NULL	Indicates end of file or an error.
x	A pointer to an object having the `netent` structure. See Section 3.2.6 for a description of the `netent` structure.

Errors

EFAULT	The buffer described by **name** is not a valid, writable part of the user address space.
EINVAL	The **net** or **net_data** argument is invalid.
ESRCH	The search failed.

getpeername()

Returns the name of the connected peer.

The $QIO equivalent is the IO$_SENSEMODE function with the **p4** argument.

Format

```
#include   <types.h>

#include   <socket.h>

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

Arguments

s
A socket descriptor created using `socket( )`.

name
A pointer to a buffer where the peer name is to be returned.

namelen
An address of an integer that specifies the size of the **name** buffer. On return, it is modified to reflect the actual length, in bytes, of the **name** returned.

Description

This function returns the name of the peer connected to the specified socket descriptor.

Related Functions

See also `bind( )`, `socket( )`, and `getsockname( )`.

Return Values

0	Successful completion.
-1	Error; `errno` is set to indicate the error.

Errors

EBADF	The descriptor is invalid.
EFAULT	The **name** argument is not a valid part of the user address space.
EINVAL	The socket has been shut down.
ENOBUFS	The system has insufficient resources to complete the call.
ENOTCONN	The socket is not connected.
ENOTSOCK	The socket descriptor is invalid.
EOPNOTSUPP	The operation is not supported for the socket protocol.

getprotobyname()

Searches the protocols database until a matching protocol name is found or until end of file is encountered.

Format

```
#include   <netdb.h>

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

Argument

name
A pointer to a string containing the desired protocol name.

Description

This function returns a pointer to a `protoent` structure containing data from the protocols database:

```
struct  protoent {
        char   *p_name;       /* official name of protocol */
        char   **p_aliases;   /* alias list */
        long   p_proto;       /* protocol number */
};
```

The members of this structure are:

`p_name`	The official name of the protocol.
`p_aliases`	A zero-terminated list of alternate names for the protocol.
`p_proto`	The protocol number.

All information is contained in a static area, so it must be copied to be saved.

Related Functions
See also `getprotoent( )` and `getprotobynumber( )`.

Return Values

NULL	Indicates end of file or an error; `errno` is set to one of the following values.
x	A pointer to a `protoent` structure.

getprotobynumber()

Searches the protocols database until a matching protocol number is found or until end of file is encountered.

Format

```
#include   <netdb.h>

struct protoent *getprotobynumber   ( int *proto) ;
```

Argument

proto
A pointer to a string containing the desired protocol number.

Description

This function returns a pointer to a `protoent` structure containing the data from the protocols database:

```
struct  protoent {
        char   *p_name;       /* official name of protocol */
        char   **p_aliases;   /* alias list */
        long   p_proto;       /* protocol number */
};
```

The members of this structure are:

`p_name`	The official name of the protocol.
`p_aliases`	A zero-terminated list of alternate names for the protocol.
`p_proto`	The protocol number.

All information is contained in a static area, so it must be copied to be saved.

Related Functions
See also `getprotoent( )` and `getprotobyname( )`.

Return Values

NULL	Indicates end of file or an error.
x	A pointer to a `protoent` structure.

getprotoent()

Reads the next line from the protocols database.

Format

```
#include   <netdb.h>

struct protoent *getprotoent( );
```

Description

This function returns a pointer to a `protoent` structure containing the data from the protocols database:

```
struct  protoent {
        char   *p_name;       /* official name of protocol */
        char   **p_aliases;   /* alias list */
        long   p_proto;       /* protocol number */
};
```

The members of this structure are:

`p_name`	The official name of the protocol.
`p_aliases`	A zero-terminated list of alternate names for the protocol.
`p_proto`	The protocol number.

The `getprotoent( )` function keeps a pointer in the database, allowing successive calls to be used to search the entire file.

All information is contained in a static area, so it must be copied to be saved.

Related Functions

See also `getprotobyname( )` and `getprotobynumber( )`.

Return Values

NULL	Indicates an end of file or an error.
x	A pointer to a `protoent` structure.

getservbyname()

Gets information on the named service from the network services database.

Format

```
#include   <netdb.h>

struct servent *getservbyname   ( char *name, char *proto );
```

Arguments

name
A pointer to a string containing the name of the service about which information is required.

proto
A pointer to a string containing the name of the protocol (TCP or UDP) for which to search.

Description

This function searches sequentially from the beginning of the file until a matching service name is found, or until end of file is encountered. If a protocol name is also supplied, searches must also match the protocol.

This function returns a pointer to a `servent` structure containing the data from the network services database:

```
struct  servent {
        char    *s_name;        /* official name of service */
        char    **s_aliases;    /* alias list */
        long    s_port;         /* port service resides at */
        char    *s_proto;       /* protocol to use */
};
```

The members of this structure are:

`s_name`	The official name of the service.
`s_aliases`	A zero-terminated list of alternate names for the service.
`s_port`	The port number at which the service resides. Port numbers are returned in network byte order.
`s_proto`	The name of the protocol to use when contacting the service.

All information is contained in a static area, so it must be copied to be saved.

Related Functions
See also `getservbyport( )`.

Return Values

NULL	Indicates end of file or an error.
x	A pointer to a `servent` structure.

getservbyport()

Gets information on the specified port from the network services database.

Format

```
#include   <netdb.h>

struct servent *getservbyport   ( int port, char *proto );
```

Arguments

port
The port number for which to search. This port number should be specified in network byte order.

proto
A pointer to a string containing the name of the protocol (TCP or UDP) for which to search.

Description

This function searches sequentially from the beginning of the file until a matching port is found, or until end of file is encountered. If a protocol name is also supplied, searches must also match the protocol.

This function returns a pointer to a `servent` structure containing the broken-out fields of the requested line in the network services database:

```
struct  servent {
        char    *s_name;        /* official name of service */
        char    **s_aliases;    /* alias list */
        long    s_port;         /* port service resides at */
        char    *s_proto;       /* protocol to use */
};
```

The members of this structure are:

`s_name`	The official name of the service.
`s_aliases`	A zero-terminated list of alternate names for the service.
`s_port`	The port number at which the service resides. Port numbers are returned in network byte order.
`s_proto`	The name of the protocol to use when contacting the service.

All information is contained in a static area, so it must be copied to be saved.

Related Functions
See also `getservbyname( )`.

Return Values

NULL	Indicates end of file or an error.
x	A pointer to a `servent` structure.

getsockname()

Returns the name associated with a socket.

The $QIO equivalent is the IO$_SENSEMODE function with the **p3** argument.

Format

```
#include   <types.h>

#include   <socket.h>

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

Arguments

s
A socket descriptor created with `socket( )` function and bound to the socket name with the `bind( )` function.

name
A pointer to the buffer in which `getsockname( )` should return the socket name.

namelen
A pointer to an integer containing the size of the buffer pointed to by **name**. On return, the integer contains the actual size, in bytes, of the name returned.

Description

This function returns the current name for the specified socket descriptor. The name is a format specific to the address family (AF_INET) assigned to the socket.

The `bind( )` function, not the `getsockname( )` function, makes the association of the name to the socket.

Related Functions
See also `bind( )` and `socket( )`.

Return Values

0	Successful completion.
-1	Error; `errno` is set to indicate the error.

Errors

EBADF	The descriptor is invalid.
EFAULT	The **name** argument is not a valid part of the user address space.
ENOBUFS	The system has insufficient resources to complete the call.
ENOTSOCK	The socket descriptor is invalid.
EOPNOTSUPP	The operation is not supported for this socket's protocol.

getsockopt()

Returns the options set on a socket.

The $QIO equivalent is the IO$_SENSEMODE function.

Format

```
#include   <types.h>

#include   <socket.h>

int getsockopt   ( int s, int level, int optname, char *optval, unsigned int *optlen );
```

Arguments

s
A socket descriptor created by the `socket( )` function.

level
The protocol level for which the socket options are desired. It can have one of the following values:

SOL_SOCKET	Get the options at the socket level.
p	Any protocol number. Get the options for protocol level specified by *p*. See the IN.H header file for the various IPPROTO values.

optname
Interpreted by the protocol specified in the level. Options at each protocol level are documented with the protocol.

For descriptions of the supported socket level options, see the description of `setsockopt( )` in this chapter.

optval
Points to a buffer in which the value of the specified option should be placed by `getsockopt( )`.

optlen
Points to an integer containing the size of the buffer pointed to by **optval**. On return, the integer is modified to contain the actual size of the option value returned.

Description

This function gets information on socket options. See the appropriate protocol for information about available options at each protocol level.

Return Values

0	Successful completion.
-1	Error; `errno` is set to indicate the error.

Errors

EACCES	The calling process does not have appropriate permissions.
EBADF	The socket descriptor is invalid.
EDOM	The send and receive timeout values are too large to fit in the timeout fields of the socket structure.
EFAULT	The address pointed to by the **optval** argument is not in a valid (writable) part of the process space, or the **optlen** argument is not in a valid part of the process address space.
EINVAL	The **optval** or **optlen** argument is invalid; or the socket is shut down.
ENOBUFS	The system has insufficient resources to complete the call.
ENOTSOCK	The socket descriptor is invalid.
ENOPROTOOPT	The option is unknown or the protocol is unsupported.
EOPNOTSUPP	The operation is not supported by the socket protocol.
ENOPROTOOPT	The option is unknown.
ENOTSOCK	The socket descriptor is invalid.

htonl()

Converts longwords from host byte order to network byte order.

Format

```
#include  <in.h>

unsigned long int htonl  ( unsigned long int hostlong );
```

Argument

hostlong
A longword in host byte order (OpenVMS systems). All integers on OpenVMS systems are in host byte order unless otherwise specified.

Description

This function converts 32-bit unsigned integers from host byte order to network byte order.

Network byte order is the format in which data bytes are expected to be transmitted through a network. All hosts on a network should send data in network byte order. Not all hosts have an internal data representation format that is identical to the network byte order. The host byte order is the format in which bytes are ordered internally on a specific host. The host byte order on OpenVMS systems differs from the network byte order.

This function is most often used with internet addresses. Network byte order places the byte with the most significant bits at lower addresses, whereas OpenVMS systems place the most significant bits at the highest address.

Note

The 64-bit return from OpenVMS Alpha systems has zero-extended bits in the high 32 bits of R0.

Return Value

x	A longword in network byte order.

htons()

Converts short integers from host byte order to network byte order.

Format

```
#include  <in.h>

unsigned short int htons  ( unsigned short int hostshort );
```

Argument

hostshort
A short integer in host byte order (OpenVMS systems). All short integers on OpenVMS systems are in host byte order unless otherwise specified.

Description

This function converts 16-bit unsigned integers from host byte order to network byte order.

Network byte order is the format in which data bytes are expected to be transmitted through a network. All hosts on a network should send data in network byte order. Not all hosts have an internal data representation format that is identical to the network byte order. The host byte order is the format in which bytes are ordered internally on a specific host. The host byte order on OpenVMS systems differs from the network byte order.

This function is most often used with ports as returned by `getservent( )`. Network byte order places the byte with the most significant bits at lower addresses, whereas OpenVMS systems place the most significant bits at the highest address.

Note

The 64-bit return from OpenVMS Alpha systems has zero-extended bits in the high 32 bits of R0.

Return Value

x	A short integer in network byte order. Integers in network byte order cannot be used for arithmetic computation on OpenVMS systems.

inet_addr()

Converts internet addresses in text form into numeric (binary) internet addresses in dotted decimal format.

Format

```
#include   <in.h>

#include   <inet.h>

int inet_addr   ( char *cp );
```

Argument

cp
A pointer to a null-terminated character string containing an internet address in the standard internet dotted-decimal format.

Description

This function returns an internet address in network byte order when given an ASCII (null-terminated) string that represents the address in the internet standard dotted-decimal format as its argument.

Internet addresses specified with the dotted-decimal format take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the 4 bytes of an internet address. Note that when an internet address is viewed as a 32-bit integer quantity on an OpenVMS system, the bytes appear in binary as `d.c.b.a`. That is, OpenVMS bytes are ordered from least significant to most significant.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as parts in a dotted-decimal address can be decimal, octal, or hexadecimal, as specified in the C language. (That is, a leading 0x or 0X implies hexadecimal; a leading 0 implies octal; otherwise, the number is interpreted as decimal.)

Note

The 64-bit return from OpenVMS Alpha systems has zero-extended bits in the high 32 bits of R0.

Return Values

-1	Indicates that **cp** does not point to a proper internet address.
x	An internet address in network byte order.

inet_lnaof()

Returns the local network address portion of an internet address.

Format

```
#include   <in.h>

#include   <inet.h>

int inet_lnaof   ( struct in_addr in );
```

Argument

in
An internet address.

Description

This function returns the local network address portion of a full internet address.

Note

The 64-bit return from OpenVMS Alpha systems has zero-extended bits in the high 32 bits of R0.

Return Value

x	The local network address portion of an internet address, in host byte order.

inet_makeaddr()

Returns an internet address based on a particular local address and a network.

Format

```
#include   <in.h>

#include   <inet.h>

struct in_addr inet_makeaddr   ( int net, int lna );
```

Arguments

net
An internet network address in host byte order.

lna
A local network address on network **net** in host byte order.

Description

This function combines the **net** and **lna** arguments into a single internet address.

Note

The 64-bit return from OpenVMS Alpha systems has zero-extended bits in the high 32 bits of R0.

Return Value

x	An internet address in network byte order.

inet_netof()

Returns the internet network address portion of an internet address.

Format

```
#include   <in.h>

#include   <inet.h>

int inet_netof   ( struct in_addr in );
```

Argument

in
An internet address.

Description

This function returns the internet network address (NET) portion of a full internet address.

Note

The 64-bit return from OpenVMS Alpha systems has zero-extended bits in the high 32 bits of R0.

Return Value

x	The internet network portion of an internet address, in host byte order.

inet_network()

Converts a null-terminated text string representing an internet address into a network address in local host format.

Format

```
#include   <in.h>

#include   <inet.h>

int inet_network   ( char *cp );
```

Argument

cp
A pointer to an ASCII (null-terminated) character string containing a network address in the dotted-decimal format.

Description

This function returns an internet network address as local host integer value when an ASCII string representing the address in the internet standard dotted-decimal format is given as its argument.

Note

The 64-bit return from OpenVMS Alpha systems has zero-extended bits in the high 32 bits of R0.

Return Values

-1	Indicates that **cp** does not point to a proper internet network address.
x	An internet network address, in local host order.

inet_ntoa()

Converts an internet address into a text string representing the address in the standard internet dotted-decimal format.

Format

```
#include   <in.h>

#include   <inet.h>

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

Argument

in
An internet address in network byte order.

Description

This function converts an internet address into an ASCII (null-terminated) string that represents the address in standard internet dotted-decimal format.

Because the string is returned in a static buffer that is overwritten by subsequent calls to `inet_ntoa( )`, you should copy the string to a safe place.

Return Value

x	A pointer to a string containing the internet address in dotted-decimal format.

ioctl()

Controls I/O requests to obtain network information.

Format

```
#include   <ioctl.h>

int ioctl   ( int s, int request, ... /* arg */ );
```

Argument

s
Specifies the socket descriptor of the requested network device.

request
Specifies the type of `ioctl` command to be performed on the device. The **request** types are grouped as follows:

- Socket operations
- File operations
- Interface operations
- ARP cache operations
- Routing table operations

Refer to Appendix B for a complete list of `IOCTL` commands.

arg
Specifies arguments for this request. The type of **arg** is dependent on the specific `ioctl( )` request and device to which the `ioctl` call is targeted.

Description

The `ioctl( )` function performs a variety of control functions on devices. The functions performed are device-specific control functions. The **request** and **arg** arguments are passed to the file designated by **fildes** and then interpreted by the device driver. The basic I/O functions are performed through the `read( )` and `write( )` functions.

Encoded in an `ioctl( )` request is whether the argument is an **in** argument or an **out** argument, and the size of the **arg** argument in bytes. The macros and definitions used to specify an `ioctl( )` request are located in the `IOCTL.H` header file.

Return Values

-1	Error; `errno` is set to indicate the error.

Errors

EBADF	The **fildes** argument is not a valid open file descriptor.
EINTR	A signal was caught during the `ioctl( )` operation.

If an underlying device driver detects an error, `errno` might be set to one of the following values:

EINVAL	Either the **request** or the **arg** argument is not valid.
ENOTTY	Reserved for Compaq use. The **fildes** argument is not associated with a character special device, or the specified request does not apply to the type of object that the **fildes** argument references.
ENXIO	The **request** and **arg** arguments are valid for this device driver, but the service requested cannot be performed on the particular subdevice.

listen()

Converts an unconnected socket into a passive socket and indicates that the kernel should accept incoming connection requests directed to the socket.

Sets the maximum limit of outstanding connection requests for a socket that is connection-oriented.

The $QIO equivalent is the IO$_SETMODE function.

Format

int listen (int *s*, int *backlog*);

Arguments

s
A socket descriptor of type SOCK_STREAM created using the socket( ) function.

backlog
The maximum number of pending connections that can be queued on the socket at any given time. The maximum number of pending connections can be set using the sysconfig utility to set the value of somaxconn. The default value for the maximum number of pending connections is 1024.

Description

This function creates a queue for pending connection requests on socket **s** with a maximum size equal to the value of **backlog**. Connections can then be accepted with the accept( ) function.

If a connection request arrives with the queue full (that is, more connections pending than specified by the **backlog** argument), the request is ignored so that TCP retries can succeed. If the backlog has not cleared by the time TCP times out, the connect( ) function fails with an errno indication of ETIMEDOUT.

Related Functions
See also accept( ), connect( ), and socket( ).

Return Values

0	Successful completion.
-1	Error; errno is set to indicate the error.

Errors

EBADF	The socket descriptor is invalid.
EDESTADDRREQ	The socket is not bound to a local address, and the protocol does not support listening on an unbound socket.
EINVAL	The socket is already connected, or the socket is shut down.

ENOBUFS	The system has insufficient resources to complete the call.
ENOTSOCK	The socket descriptor is invalid.
EOPNOTSUPP	The referenced socket is not of a type that supports the operation `listen( )`.

ntohl()

Converts longwords from network byte order to host byte order.

Format

```
#include  <in.h>

unsigned long ntohl  ( unsigned long netlong );
```

Argument

netlong
A longword in network byte order. Integers in network byte order cannot be used for arithmetic computation on OpenVMS systems.

Description

This function converts 32-bit unsigned integers from network byte order to host byte order.

The network byte order is the format in which data bytes are expected to be transmitted through a network. All hosts on a network should send data in network byte order. Not all hosts have an internal data representation format that is identical to the network byte order. The host byte order is the format in which bytes are ordered internally on a specific host. The host byte order on OpenVMS systems differs from the network byte order.

This function is most often used with internet addresses. Network byte order places the byte with the most significant bits at lower addresses, whereas OpenVMS systems place the most significant bits at the highest address.

Return Value

x	A longword in host byte order.

ntohs()

Converts short integers from network byte order to host byte order.

Format

```
#include  <in.h>

unsigned short ntohs  ( unsigned short netshort );
```

Argument

netshort
A short integer in network byte order. Integers in network byte order cannot be used for arithmetic computation on OpenVMS systems.

Description

This function converts 16-bit unsigned integers from network byte order to host byte order.

The network byte order is the format in which data bytes are expected to be transmitted through a network. All hosts on a network should send data in network byte order. Not all hosts have an internal data representation format that is identical to the network byte order. The host byte order is the format in which bytes are ordered internally on a specific host. The host byte order on OpenVMS systems differs from the network byte order.

This function is most often used with internet ports as returned by `getservent( )`. Network byte order places the byte with the most significant bits at lower addresses, whereas OpenVMS systems place the most significant bits at the highest address.

Return Value

x	A short integer in host byte order (OpenVMS systems).

read()

Reads bytes from a socket or file and places them in a user-provided buffer.

The $QIO equivalent is the IO$_READVBLK function.

Format

```
#include   <unixio.h>

int read   ( int d, void *buffer, int nbytes );
```

Arguments

d
A descriptor that must refer to a socket or file currently opened for reading.

buffer
The address of a user-provided buffer in which the input data is placed.

nbytes
The maximum number of bytes allowed in the read operation.

Description

If the end of file is not reached, the `read( )` function returns **nbytes**. If the end of file occurs during the `read( )` function, it returns the number of bytes read.

Upon successful completion, `read( )` returns the number of bytes actually read and placed in the buffer.

Related Functions
See also `socket( )`.

Return Values

x	The number of bytes read and placed in the buffer.
0	Peer has closed the connection.
-1	Error; `errno` is set to indicate the error.

Errors

EBADF	The socket descriptor is invalid.
ECONNRESET	A connection was forcibly closed by a peer.
EFAULT	The data was specified to be received into a nonexistent or protected part of the process address space.
EINTR	A signal interrupted the `recv( )` function before any data was available.
EINVAL	The MSG_OOB flag is set and no out-of-band data is available.

ENOBUFS	The system has insufficient resources to complete the call.
ENOMEM	The system did not have sufficient memory to fulfill the request.
ENOTCONN	A receive is attempted on a connection-oriented socket that is not connected.
ENOTSOCK	The socket descriptor is invalid.
EOPNOTSUPP	The specified flags are not supported for this socket type or protocol.
EWOULDBLOCK	The socket is marked nonblocking, and no data is waiting to be received.

recv()

Receives bytes from a connected socket and places them into a user-provided buffer.

The $QIO equivalent is the IO$_READVBLK function.

Format

```
#include   <types.h>

#include   <socket.h>

int recv   ( int s, char *buf, int len, int flags );
```

Arguments

s
A socket descriptor created as the result of a call to `accept( )` or `connect( )`.

buf
A pointer to a user-provided buffer into which received data will be placed.

len
The size of the buffer pointed to by **buf**.

flags
A bit mask that can contain one or more of the following flags. The mask is built by using a logical OR operation on the appropriate values.

Flag	Description
MSG_OOB	Allows you to receive out-of-band data. If out-of-band data is available, it is read first. If no out-of-band data is available, the MSG_OOB flag is ignored. Use the `send( )`, `sendmsg( )`, and `sendto( )` functions to send out-of-band data.
MSG_PEEK	Allows you to examine data in the receive buffer without removing it from the system's buffers.

Description

This function receives data from a connected socket. To receive data on an unconnected socket, use the `recvfrom( )` or `recvmsg( )` functions. The received data is placed in the buffer **buf**.

Data is sent by the socket's peer using the `send`, `sendmsg( )`, `sendto( )`, or `write( )` functions.

Use the `select( )` function to determine when more data arrives.

If no data is available at the socket, the receive call waits for data to arrive, unless the socket is nonblocking. If the socket is nonblocking, a -1 is returned with the external variable `errno` set to `EWOULDBLOCK`.

Related Functions

See also `read( )`, `send( )`, `sendmsg( )`, `sendto( )`, and `socket( )`.

Return Values

x	The number of bytes received and placed in **buf**.
0	Peer has closed its send side of the connection.
-1	Error; `errno` is set to indicate the error.

Errors

EBADF	The socket descriptor is invalid.
ECONNRESET	A connection was forcibly closed by a peer.
EFAULT	The data was specified to be received into a nonexistent or protected part of the process address space.
EINTR	A signal interrupted the `recv( )` function before any data was available.
EINVAL	The MSG_OOB flag is set and no out-of-band data is available.
ENOBUFS	The system has insufficient resources to complete the call.
ENOMEM	The system did not have sufficient memory to fulfill the request.
ENOTCONN	A receive is attempted on a connection-oriented socket that is not connected.
ENOTSOCK	The socket descriptor is invalid.
EOPNOTSUPP	The specified flags are not supported for this socket type or protocol.
EWOULDBLOCK	The socket is marked nonblocking, and no data is waiting to be received.

recvfrom()

Receives bytes for a socket from any source.

Format

```
#include   <types.h>

#include   <socket.h>

int recvfrom   ( int s, char *buf, int len, int flags, struct sockaddr *from, int *fromlen)   ;
```

Arguments

s
A socket descriptor created with the `socket( )` function and bound to a name using the `bind( )` function or as a result of the `accept( )` function.

buf
A pointer to a buffer into which received data is placed.

len
The size of the buffer pointed to by **buf**.

flags
A bit mask that can contain one or more of the following flags. The mask is built by using a logical OR operation on the appropriate values.

Flag	Description
MSG_OOB	Allows you to receive out-of-band data. If out-of-band data is available, it is read first. If no out-of-band data is available, the MSG_OOB flag is ignored. To send out-of-band data, use the `send( )`, `sendmsg( )`, and `sendto( )` functions.
MSG_PEEK	Allows you to examine the data that is next in line to be received without actually removing it from the system's buffers.

from
A buffer that the `recvfrom( )` function uses to place the address of the sender who sent the data.

If **from** is non-null, the address is returned. If **from** is null, the address is not returned.

fromlen
Points to an integer containing the size of the buffer pointed to by **from**. On return, the integer is modified to contain the actual length of the socket address structure returned.

Description

This function allows a named, unconnected socket to receive data. The data is placed in the buffer pointed to by **buf**, and the address of the sender of the data is placed in the buffer pointed to by **from** if **from** is non-null. The structure that **from** points to is assumed to be as large as the `sockaddr` structure. See Section 3.2.7 for a description of the `sockaddr` structure.

To receive bytes from any source, the socket does not need to be connected.

You can use the `select( )` function to determine if data is available.

If no data is available at the socket, the `recvfrom( )` call waits for data to arrive, unless the socket is nonblocking. If the socket is nonblocking, a -1 is returned with the external variable `errno` set to `EWOULDBLOCK`.

Related Functions

See also `read( )`, `send( )`, `sendmsg( )`, `sendto( )`, and `socket( )`.

Return Values

x	The number of bytes of data received and placed in **buf**.
0	Successful completion.
-1	Error; `errno` is set to indicate the error.

Errors

EBADF	The socket descriptor is invalid.
ECONNRESET	A connection was forcibly closed by a peer.
EFAULT	A valid message buffer was not specified. Nonexistent or protected address space is specified for the message buffer.
EINTR	A signal interrupted the `recvfrom( )` function before any data was available.
EINVAL	The MSG_OOB flag is set, and no out-of-band data is available.
ENOBUFS	The system has insufficient resources to complete the call.
ENOMEM	The system did not have sufficient memory to fulfill the request.
ENOTCONN	A receive is attempted on a connection-oriented socket that is not connected.
ENOTSOCK	The socket descriptor is invalid.
EOPNOTSUPP	The specified flags are not supported for this socket type.
ETIMEDOUT	The connection timed out when trying to establish a connection or when a transmission timed out on an active connection.

EWOULDBLOCK	The NBIO (nonblocking) flag is set for the socket descriptor and the process delayed during the write operation.

recvmsg()

Receives bytes on a socket and places them into scattered buffers.

Format

#include <types.h>

#include <socket.h>

int recvmsg (int *s*, struct msghdr *msg*, int *flags*); (BSD Version 4.4)

int recvmsg (int *s*, struct omsghdr *msg*, int *flags*); (BSD Version 4.3)

Arguments

s
A socket descriptor created with the `socket( )` function.

msg
See Section 3.2.5 for a description of the `msghdr` structure for BSD Versions 4.3 and 4.4.

flags
A bit mask that can contain one or more of the following flags. The mask is built by using a logical OR operation on the appropriate values.

Flag	Description
MSG_OOB	Allows you to receive out-of-band data. If out-of-band data is available, it is read first. If no out-of-band data is available, the MSG_OOB flag is ignored. Use `send( )`, `sendmsg( )`, and `sendto( )` functions to send out-of-band data.
MSG_PEEK	Allows you to peek at the data that is next in line to be received without actually removing it from the system's buffers.

Description

You can use this function with any socket, whether or not it is in a connected state. It receives data sent by a call to `sendmsg( )`, `send( )`, or `sendto( )`. The message is scattered into several user buffers if such buffers are specified.

To receive data, the socket does not need to be connected to another socket.

When the `ioveciovcnt` array specifies more than one buffer, the input data is scattered into `iovcnt` buffers as specified by the members of the `iovec` array:

```
iov0, iov1, ..., ioviovcnt
```

When a message is received, it is split among the buffers by filling the first buffer in the list, then the second, and so on, until either all of the buffers are full or there is no more data to be placed in the buffers.

You can use the `select( )` function to determine when more data arrives.

Related Functions

See also `read( )`, `send( )`, and `socket( )`.

Return Values

x	The number of bytes returned in the `msg_iov` buffers.
0	Successful completion.
-1	Error; `errno` is set to indicate the error.

Errors

EBADF	The socket descriptor is invalid.
ECONNRESET	A connection was forcibly closed by a peer.
EFAULT	The message argument is not in a readable or writable part of user address space.
EINTR	This function was interrupted by a signal before any data was available.
EINVAL	The MSG_OOB flag is set, and no out-of-band data is available. The value of the `msg_iovlen` member of the `Lmsghdr` structure is less than or equal to zero or is greater than `IOV_MAX`.
ENOBUFS	The system has insufficient resources to complete the call.
ENOMEM	The system did not have sufficient memory to fulfill the request.
ENOTCONN	A receive is attempted on a connection-oriented socket that is not connected.
ENOTSOCK	The socket descriptor is invalid.
EOPNOTSUPP	The specified flags are not supported for this socket type.
EWOULDBLOCK	The socket is marked nonblocking, and no data is ready to be received.

select()

Allows you to poll or check a group of sockets for I/O activity. This function indicates which sockets are ready to be read or written, or which sockets have an exception pending.

Format

```
#include   <time.h>

int select   ( int nfds, int *readfds, int *writefds, int *execptfds, struct timeval *timeout );
```

Arguments

nfds
Specifies the number of open objects that may be ready for reading or writing or that have exceptions pending. The **nfds** argument is normally limited to FD_SETSIZE. Note that a single process can have a maximum of 65535 simultaneous channels (including sockets) on OpenVMS Alpha systems, and a maximum of 2047 on OpenVMS VAX systems.

readfds
A pointer to an array of bits, organized as integers, that should be examined for read readiness. If bit *n* of the longword is set, socket descriptor *n* is checked to see whether it is ready to be read. All bits set in the bit mask must correspond to the file descriptors of sockets. The select( ) function cannot be used on normal files.

On return, the array to which **readfds** points contains a bit mask of the sockets that are ready for reading. Only bits that were set on entry to the select( ) function can be set on exit.

writefds
A pointer to an array of bits, organized as integers, that should be examined for write readiness. If bit *n* of the longword is set, socket descriptor *n* is checked to see whether it is ready to be written to. All bits set in the bit mask must correspond to socket descriptors.

On return, the array to which **writefds** points contains a bit mask of the sockets that are ready for writing. Only bits that were set on entry to the select( ) function are set on exit.

exceptfds
A pointer to an array of bits, organized as integers, that is examined for exceptions. If bit *n* of the longword is set, socket descriptor *n* is checked to see whether it has any pending exceptions. All bits set in the bit mask must correspond to the file descriptors of sockets.

On return, the array **exceptfds** pointer contains a bit mask of the sockets that have exceptions pending. Only bits that were set on entry to the select( ) function can be set on exit.

timeout
The length of time that the select( ) function should examine the sockets before returning. If one of the sockets specified in the **readfds**, **writefds**, and **exceptfds** bit masks is ready for I/O, the select( ) function returns before the timeout period expires.

The **timeout** argument points to a timeval structure. See Section 3.2.9 for a description of the timeval structure.

Description

This function determines the I/O status of the sockets specified in the various mask arguments. It returns when a socket is ready to be read or written, when the timeout period expires, or when exceptions occur. If **timeout** is a non-null pointer, it specifies a maximum interval to wait for the selection to complete.

If the **timeout** argument is null, the select( ) function blocks indefinitely until a selected event occurs. To effect a poll, the value for **timeout** should be non-null, and should point to a zero-value structure.

If a process is blocked on a select( ) function while waiting for input for a socket and the sending process closes the socket, then the select( ) function notes this as an event and unblocks the process. The descriptors are always modified on return if the select( ) function returns because of the timeout.

Note

When the socket option SO_OOBINLINE is set on the device socket, the select( ) function on both read and exception events returns the socket mask that is set on both the read and the exception mask. Otherwise, only the exception mask is set.

Related Functions

See also accept( ), connect( ), read( ), recv( ), recvfrom( ), recvmsg( ), send( ), sendmsg( ), sendto( ), and write( ).

Return Values

n	The number of sockets ready for I/O or pending exceptions. This value matches the number of returned bits that are set in all output masks.
0	The select( ) function timed out before any socket became ready for I/O.
-1	Error; errno is set to indicate the error.

Errors

EBADF	One or more of the I/O descriptor sets specified an invalid file descriptor.
EINTR	A signal was delivered before the time limit specified by the **timeout** argument expired and before any of the selected events occurred.
EINVAL	The time limit specified by the **timeout** argument is invalid. The **nfds** argument is less than zero, or greater than or equal to FD_SETSIZE.

EAGAIN	Allocation of internal data structures failed. A later call to the `select( )` function may complete successfully.
ENETDOWN	TCP/IP Services was not started.
ENOTSOCK	The socket descriptor is invalid.

send()

Sends bytes through a socket to its connected peer.

The $QIO equivalent is the IO$_WRITEVBLK function.

Format

```
#include   <types.h>

#include   <socket.h>

int send   ( int s, char *msg, int len, int flags );
```

Arguments

s
A socket descriptor created with the `socket( )` function that was connected to another socket using the `accept( )` or `connect( )` function.

msg
A pointer to a buffer containing the data to be sent.

len
The length, in bytes, of the data pointed to by *msg*.

flags
Can be either 0 or MSG_OOB. If it is MSG_OOB, the data is sent out of band. Data can be received before other pending data on the receiving socket if the receiver also specifies `MSG_OOB` in the flag argument of its `recv( )` or `recvfrom( )` call.

Description

You can use this function only on connected sockets. To send data on an unconnected socket, use the `sendmsg( )` or `sendto( )` function. The `send( )` function passes data along to its connected peer, which can receive the data by using the `recv( )` or `read( )` function.

If there is no space available to buffer the data being sent on the receiving end of the connection, `send( )` normally blocks until buffer space becomes available. If the socket is defined as nonblocking, however, `send( )` fails with an `errno` indication of `EWOULDBLOCK`. If the message is too large to be sent in one piece, and the socket type requires that messages be sent atomically (`SOCK_DGRAM`), `send( )` fails with an `errno` indication of `EMSGSIZE`.

No indication of failure to deliver is implicit in a `send( )`. All errors (except `EWOULDBLOCK`) are detected locally. You can use the `select( )` function to determine when it is possible to send more data.

Related Functions

See also `read( )`, `recv( )`, `recvmsg( )`, `recvfrom( )`, `getsockopt( )`, and `socket( )`.

Return Values

n	The number of bytes sent. This value normally equals **len**.
-1	Error; `errno` is set to indicate the error.

Errors

EBADF	The socket descriptor is invalid.
ECONNRESET	A connection was forcibly closed by a peer.
EDESTADDRREQ	The socket is not connection-oriented, and no peer address is set.
EFAULT	The message argument is not in a readable or writable part of the user address space.
EINTR	A signal interrupted the `send( )` before any data was transmitted.
EMSGSIZE	The message is too large to be sent all at once, as the socket requires.
ENETDOWN	The local network connection is not operational.
ENETUNREACH	The destination network is unreachable.
ENOBUFS	The system has insufficient resources to complete the call.
ENOTCONN	The socket is not connected or has not had the peer prespecified.
ENOTSOCK	The socket descriptor is invalid.
EOPNOTSUPP	The socket argument is associated with a socket that does not support one or more of the values set in flags.
EWOULDBLOCK	The socket is marked nonblocking, and no space is available for the `send( )` function.

sendmsg()

Sends gathered bytes through a socket to any other socket.

Format

```
#include   <types.h>

#include   <socket.h>

int sendmsg   ( int s, struct msghdr *msg, int flags ); (BSD Version 4.4)

int sendmsg   ( int s, struct omsghdr *msg, int flags ); (BSD Version 4.3)
```

Arguments

s
A socket descriptor created with the socket( ) function.

msg
A pointer to a msghdr structure containing the message to be sent. See Section 3.2.5 for a description of the msghdr structure for BSD Versions 4.3 and 4.4.

The msg_iov field of the msghdr structure is used as a series of buffers from which data is read in order until msg_iovlen bytes have been obtained.

flags
Can be either 0 or MSG_OOB. If it is equal to MSG_OOB, the data is sent out of band. Data can be received before other pending data on the receiving socket if the receiver specifies a flag of MSG_OOB.

Description

You can use this function on any socket to send data to any named socket. The data in the msg_iov field of the msghdr structure is sent to the socket whose address is specified in the msg_name field of the structure. The receiving socket gets the data using the read( ), recv( ), recvfrom( ), or recvmsg( ) function. When the iovec array specifies more than one buffer, the data is gathered from all specified buffers before being sent. See Section 3.2.3 for a description of the iovec structure.

If no space is available to buffer the data on the receiving end of the connection, the sendmsg( ) function blocks until buffer space becomes available. If the socket is defined as nonblocking, the sendmsg( ) function fails with an errno indication of EWOULDBLOCK. If the message is too large to be sent in one piece and the socket type requires that messages be sent atomically (SOCK_DGRAM), the sendmsg( ) fails with an errno indication of EMSGSIZE.

If the address specified is an INADDR_BROADCAST address, the SO_BROADCAST socket option must be set and the process must have a system UIC, OPER, SYSPRV, or BYPASS privilege for the I/O operation to succeed.

No indication of failure to deliver is implicit in the sendmsg( ) function. All errors (except EWOULDBLOCK) are detected locally. You can use the select( ) function to determine when it is possible to send more data.

Related Functions

See also `read( )`, `recv( )`, `recvfrom( )`, `recvmsg( )`, `socket( )`, and `getsockopt( )`.

Return Values

n	The number of bytes sent.
-1	Error; `errno` is set to indicate the error.

Errors

ENOTSOCK	The socket descriptor is invalid.
EFAULT	An invalid user space address is specified for a argument.
EMSGSIZE	The socket requires that messages be sent atomically, but the size of the message to be sent makes this impossible.
EWOULDBLOCK	Blocks if the system does not have enough space for buffering the user data.

sendto()

Sends bytes through a socket to any other socket.

The $QIO equivalent is the IO$_WRITEVBLK function.

Format

```
#include   <types.h>

#include   <socket.h>

int sendto   ( int s, char *msg, int len, int flags, struct sockaddr *to, int tolen );
```

Arguments

s
A socket descriptor created with the `socket( )` function.

msg
A pointer to a buffer containing the data to be sent.

len
The length of the data pointed to by the **msg** argument.

flags
Can be either 0 or `MSG_OOB`. If it is `MSG_OOB`, the data is sent out of band. Data can be received before other pending data on the receiving socket if the receiver specifies `MSG_OOB` in its flag argument of its `recv( )`, `recvfrom( )` or `recvmsg( )` call.

to
Points to the address structure of the socket to which the data is to be sent.

tolen
The length of the address pointed to by the **to** argument.

Description

This function can be used on sockets to send data to named sockets. The data in the **msg** buffer is sent to the socket whose address is specified in the **to** argument, and the address of socket **s** is provided to the receiving socket. The receiving socket gets the data using the `read( )`, `recv( )`, `recvfrom( )`, or `recvmsg( )` function.

If there is no space available to buffer the data being sent on the receiving end of the connection, the `sendto( )` function blocks until buffer space becomes available. If the socket is defined as nonblocking, the `sendto( )` function fails with an `errno` indication of `EWOULDBLOCK`. If the message is too large to be sent in one piece and the socket type requires that messages be sent atomically (SOCK_DGRAM), the `sendto( )` function fails with an `errno` indication of `EMSGSIZE`.

No indication of failure to deliver is implicit in a `sendto( )`. All errors (except `EWOULDBLOCK`) are detected locally. You can use the `select( )` function to determine when it is possible to send more data.

If the address specified is a INADDR_BROADCAST address, then the SO_BROADCAST socket option must have been set and the process must have SYSPRV or BYPASS privilege for the I/O operation to succeed.

Related Functions

See also `read( )`, `recv( )`, `recvfrom( )`, `recvmsg( )`, `socket( )`, and `getsockopt( )`.

Return Values

n	The number of bytes sent. This value normally equals **len**.
-1	Error; `errno` is set to indicate the error.

Errors

EAFNOSUPPORT	Addresses in the specified address family cannot be used with this socket.
EBADF	The socket descriptor is invalid.
ECONNRESET	A connection was forcibly closed by a peer.
EDESTADDRREQ	You did not specify a destination address for the connectionless socket and no peer address is set.
EFAULT	An invalid user space address is specified for a argument.
EHOSTUNREACH	The destination host is unreachable.
EINTR	A signal interrupted `sendto( )` before any data was transmitted.
EINVAL	The **dest_len** argument is not a valid size for the specified address family.
EISCONN	The connection-oriented socket for which a destination address was specified is already connected.
EMSGSIZE	The message is too large to be sent all at once, as the socket requires.
ENETDOWN	The local network connection is not operational.
ENETUNREACH	The destination network is unreachable.
ENOBUFS	The system has insufficient to complete the call.
ENOMEM	The system did not have sufficient memory to fulfill the request.
ENOTCONN	The socket is connection-oriented but is not connected.
ENOTSOCK	The socket descriptor is invalid.
EOPNOTSUPP	The socket argument is associated with a socket that does not support one or more of the values set in flags.

EPIPE	The socket is shut down for writing or is connection oriented, and the peer is closed or shut down for reading. In the latter case, if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling process.
EWOULDBLOCK	The socket is marked nonblocking, and no space is available for the sendto( ) function.

setsockopt()

Sets options on a socket.

The $QIO equivalent is the IO$_SETMODE function.

Format

```
#include   <types.h>

#include   <socket.h>

int setsockopt   ( int s, int level, int optname, char *optval, int optlen );
```

Arguments

s
A socket descriptor created by the `socket( )` function.

level
The protocol level for which the socket options are to be modified. It can have one of the following values:

SOL_SOCKET	Set the options at the socket level.
p	Any protocol number. Set the options for protocol level *p*. See the IN.H header file for the various IPPROTO values.

optname
Interpreted by the protocol specified in **level**. Options at each protocol level are documented with the protocol.

Refer to:

- Table A–1 for a list of socket options
- Table A–2 for a list of TCP options
- Table A–3 for a list of IP options

optval
Points to a buffer containing the arguments of the specified option.

All socket-level options other than `SO_LINGER` should be nonzero if the option is to be enabled, or zero if it is to be disabled.

`SO_LINGER` uses a `linger` structure argument defined in the SOCKET.H header file. This structure specifies the desired state of the option and the linger interval. The option value for the `SO_LINGER` command is the address of a `linger` structure. See Section 3.2.4 for a description of the `linger` structure.

If the socket promises the reliable delivery of data and `l_onoff` is nonzero, the system blocks the process on the `close( )` attempt until it is able to transmit the data or until it decides it is unable to deliver the information. A timeout period, called the linger interval, is specified in `l_linger`.

If `l_onoff` is set to zero and a `close( )` is issued, the system processes the close in a manner that allows the process to continue as soon as possible.

optlen
An integer specifying the size of the buffer pointed to by **optval**.

Description

This function manipulates options associated with a socket. Options can exist at multiple protocol levels. They are always present at the uppermost socket level.

When manipulating socket options, specify the level at which the option resides and the name of the option. To manipulate options at the socket level, specify the value of **level** as SOL_SOCKET. To manipulate options at any other level, supply the protocol number of the appropriate protocol controlling the option. For example, to indicate that an option is to be interpreted by TCP, set the value for **level** argument to the protocol number (IPPROTO_TCP) of TCP. See the IN.H header file for the various IPPROTO values.

Return Values

0	Successful completion.
-1	Error; errno is set to indicate the error.

Errors

EACCES	The calling process does not have appropriate permissions.
EBADF	The descriptor is invalid.
EDOM	The send and receive timeout values are too large to fit in the timeout fields of the socket structure.
EINVAL	The **optlen** argument is invalid.
EISCONN	The socket is already connected; the specified option cannot be set when the socket is in the connected state.
EFAULT	The **optval** argument is not in a readable part of the user address space.
ENOBUFS	The system had insufficient resources to complete the call.
ENOPROTOOPT	The option is unknown.
ENOTSOCK	The socket descriptor is invalid.
EFAULT	The **optname** argument is invalid.

shutdown()

Shuts down all or part of a bidirectional connection on a socket. This function does not allow further receives or sends, or both.

The $QIO equivalent is the IO$_DEACCESS function with the IO$M_ SHUTDOWN function modifier.

Format

```
#include   <socket.h>

int shutdown   ( int s, int how) ;
```

Arguments

s
A socket descriptor that is in a connected state as a result of a previous call to either `connect( )` or `accept( )`.

how
How the socket is to be shut down. Use one of the following values:

0	Do not allow further calls to `recv( )` on the socket.
1	Do not allow further calls to `send( )` on the socket.
2	Do not allow further calls to both `send( )` and `recv( )`.

Description

This function allows communications on a socket to be shut down one piece at a time rather than all at once. Use the `shutdown( )` function to create unidirectional connections rather than the normal bidirectional (full-duplex) connections.

Related Functions
See also `connect( )` and `socket( )`.

Return Values

0	Successful completion.
-1	Error; `errno` is set to indicate the error.

Errors

EBADF	The socket descriptor is invalid.
EINVAL	The **how** argument is invalid.
ENOBUFS	The system has insufficient resources to complete the call.
ENOTCONN	The specified socket is not connected.
ENOTSOCK	The socket descriptor is invalid.

socket()

Creates an endpoint for communication by returning a special kind of file descriptor called a socket descriptor, which is associated with a TCP/IP Services socket device channel.

The $QIO equivalent is the IO$_SETMODE function.

Format

```
#include   <types.h>

#include   <socket.h>

int socket   ( int af, int type, int protocol );
```

Arguments

af
The address family used in later references to the socket. Addresses specified in subsequent operations using the socket are interpreted according to this family. Currently, only the AF_INET (internet) and TCPIP$C_AUXS addresses are supported.

For a network application server with the LISTEN flag enabled, you specify the TCPIP$C_AUXS address family to obtain the connected device socket created by the auxiliary server in response to incoming network traffic. For an example of this situation, refer to the example in Section E.1.3.

type
The socket types are:

- SOCK_STREAM — Provides sequenced, reliable, two-way, connection-based byte streams with an available out-of-band data transmission mechanism.
- SOCK_DGRAM — Supports datagrams (connectionless, unreliable data transmission mechanism).
- SOCK_RAW — Provides access to internal network interfaces. Available only to users with either a system UIC or the SYSPRV privilege.

protocol
The 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, if many protocols exist, a particular protocol must be specified with this argument. Use the protocol number that is specific to the communication domain in which communication takes place.

Description

This function provides the primary mechanism for creating sockets. The type and protocol of the socket affect the way the socket behaves and how it can be used.

The operation of sockets is controlled by socket-level options, which are defined in the SOCKET.H header file and described in the setsockopt( ) function section of this chapter.

Use the setsockopt( ) and getsockopt( ) functions to set and get options. Options take an integer argument that should be nonzero if the option is to be enabled or zero if it is to be disabled. SO_LINGER uses a linger structure argument defined in <socket.h>.

Related Functions

See also accept( ), bind( ), connect( ), listen( ), read( ), recv( ), recvfrom( ), recvmsg( ), select( ), send( ), sendmsg( ), sendto( ), shutdown( ), and write( ).

Related Functions

See also getsockname( ) and getsockopt( ).

Return Values

x	A file descriptor that refers to the socket descriptor.
-1	Error; errno is set to indicate the error.

Errors

EACCES	The process does not have sufficient privileges.
EAFNOSUPPORT	The specified address family is not supported in this version of the system.
EMFILE	The per-process descriptor table is full.
ENETDOWN	TCP/IP Services was not started.
ENFILE	No more file descriptors are available for the system.
ENOBUFS	The system has insufficient resources to complete the call.
ENOMEM	The system was unable to allocate kernel memory to increase the process descriptor table.
EPERM	The process is attempting to open a raw socket and does not have SYSTEM privilege.
EPROTONOSUPPORT	The socket in the specified address family is not supported.
EPROTOTYPE	The socket type is not supported by the protocol.
ESOCKTNOSUPPORT	The specified socket type is not supported in this address family.

write()

Writes bytes from a buffer to a file or socket.

The $QIO equivalent is the IO$_WRITEVBLK function.

Format

```
#include   <unixio.h>

int write   ( int d, void *buffer, int nbytes );
```

Arguments

d
A descriptor that refers to a socket or file.

buffer
The address of a buffer from which the output data is to be taken.

nbytes
The maximum number of bytes involved in the write operation.

Description

This function attempts to write a buffer of data to a socket or file.

Related Functions
See also `socket( )`.

Return Values

x	The number of bytes written to the socket or file.
-1	Error; `errno` is set to indicate the error.

Errors

EPIPE	The socket is shut down for writing or is connection oriented, and the peer is closed or shut down for reading. In the latter case, if the socket is of type `SOCK_STREAM`, the `SIGPIPE` signal is generated to the calling process.
EWOULDBLOCK	The `NBIO` (nonblocking) flag is set for the socket descriptor, and the process is delayed during the write operation.
EINVAL	The **nbytes** argument is a negative value.
EAGAIN	The `O_NONBLOCK` flag is set on this file, and the process is delayed in the write operation.
EBADF	The **d** argument does not specify a valid file descriptor that is open for writing.

EINTR	A write( ) or pwrite( ) function on a pipe is interrupted by a signal, and no bytes have been transferred through the pipe.
EINVAL	On of the following errors occurred: • The STREAM or multiplexer referenced by **d** is linked (directly or indirectly) downstream from a multiplexer. • The **iov_count** argument value was less than or equal to zero or greater than IOV_MAX. • The sum of the **iov_len** values in the **iov** array overflows a ssize_t data type. • The file position pointer associated with the **d** argument was a negative value. • One of the **iov_len** values in the **iov** array was negative, or the sum overflowed a 32-bit integer.
EPERM	An attempt was made to write to a socket of type SOCK_STREAM that is not connected to a peer socket.
EPIPE	An attempt was made to write to a pipe that has only one end open. An attempt was made to write to a pipe or FIFO that is not opened for reading by any process. A SIGPIPE signal is sent to the process.
ERANGE	An attempt was made to write to a STREAM socket where **nbytes** are outside the specified minimum and maximum range, and the minimum value is nonzero.

5
Using the $QIO System Service

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

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

5.1 $QIO System Service Variations

The two variations of the $QIO system service are:

- Queue I/O Request ($QIO) — Completes asynchronously. It returns to the caller immediately after queuing the I/O request, without waiting for the I/O operation to complete.
- Queue I/O Request and Wait ($QIOW) — Completes synchronously. It returns to the caller after the I/O operation completes.

The only difference between the $QIO and $QIOW calling sequences is the service name. The system service arguments are the same.

5.2 $QIO Format

The $QIO calling sequence has the following format:

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

Table 5–1 describes each argument.

Table 5–1 $QIO Arguments

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

5.2.1 Symbol Definition Files

Table 5–2 lists the symbol definition files for the $QIO arguments **p1** through **p6**. Use the standard mechanism for the programming language you are using to include the appropriate symbol definition files in your program.

Table 5–2 Network Symbol Definition Files

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

5.3 $QIO Functions

Table 5–3 lists the $QIO function codes commonly used in a network application.

Note

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

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

Table 5–3 $QIO Function Codes

Function	Description
$QIO(IO$_SETMODE) $QIO(IO$_SETCHAR)	Creates the socket by setting the internet domain, protocol (socket) type, and protocol of the socket.
	Binds a name (local address and port) to the socket.
	Defines a network pseudodevice as a listener on a TCP/IP server.
	Specifies socket options.
$QIO(IO$_ACCESS)	Initiates a connection request from a client to a remote host using TCP.

(continued on next page)

Table 5–3 (Cont.) $QIO Function Codes

Function	Description
	Specifies the peer where you can send datagrams.
	Accepts a connection request from a TCP/IP client when used with the IO$M_ACCEPT function modifier.
$QIO(IO$_WRITEVBLK)	Writes data (virtual block) from the local host to the remote host for stream sockets, datagrams, and raw IP.
$QIO(IO$_READVBLK)	Reads data (virtual block) from the remote host to the local host for stream sockets, datagrams, and raw IP.
$QIO(IO$_DEACCESS)	Disconnects the link established between two communication agents through an IO$_DEACCESS function.
	Shuts down the communication link when used with the IO$M_SHUTDOWN function modifier. You can shut down the receive or transmit portion of the link, or both.
$QIO(IO$_SENSECHAR) $QIO(IO$_SENSEMODE)	Obtains socket information.

5.4 $QIO Arguments

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

5.4.1 $QIO Function-Independent Arguments

Table 5–4 describes the $QIO function-independent arguments.

Table 5–4 $QIO Function-Independent Arguments

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

(continued on next page)

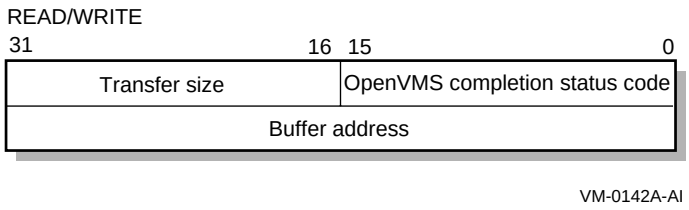
Table 5–4 (Cont.) $QIO Function-Independent Arguments

Argument	Description
iosb	The I/O status block that receives the final status message for the I/O operation. The **iosb** argument is the address of the quadword I/O status block. (For the format of the I/O status block, see Section 5.4.2.)

5.4.2 I/O Status Block

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

Figure 5–1 I/O Status Block for a Successful READ or WRITE Operation



With an unsuccessful IO$_READVBLK or an IO$_WRITEVBLK operation, in most cases, the system returns a Compaq *Tru64* UNIX error code in the second word of the I/O status block.

The OpenVMS completion codes are defined in the SSDEF.H header file. The Compaq *Tru64* UNIX error codes are defined in the ERRNO.H header file and in the TCPIP$INETDEF.H header file.

5.4.3 $QIO Function-Dependent Arguments

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

Table 5–5 $QIO Function-Dependent Arguments

$QIO	p1	p2	p3	p4	p5	p6
IO$_ACCESS	Not used	Not used	Remote socket name[4]	Not used	Not used	Not used
IO$_ACCESS \| IO$M_ACCEPT	Not used	Not used	Remote socket name[5]	Channel number[2]	Not used	Not used
IO$_ACPCONTROL	Subfunction code[3]	Input parameter[3]	Buffer length[2]	Buffer[3]	Not used	Not used
IO$_DEACCESS	Not used	Not used	Not used	Not used	Not used	Not used
IO$_DEACCESS \| IO$M_SHUTDOWN	Not used	Not used	Not used	Shutdown flags [1]	Not used	Not used
IO$_READVBLK	Buffer[2]	Buffer size[1]	Remote socket name[5]	Flags[1]	Not used	Output buffer list[3]
IO$_READVBLK \| IO$M_INTERRUPT	Buffer[2]	Buffer size[1]	Not used	Not used	Not used	Not used
IO$_WRITEVBLK	Buffer[2]	Buffer size[1]	Remote socket name[4]	Flags[1]	Input buffer list[3]	Not used
IO$_WRITEVBLK \| IO$M_INTERRUPT	Buffer[2]	Buffer size[1]	Not used	Not used	Not used	Not used
IO$_SETMODE	Socket char[2]	Not used	Local socket name[4]	Backlog limit[1]	Input parameter list[4]	Not used
IO$_SENSEMODE	Not used	Not used	Local socket name[5]	Remote socket name[5]	Not used	Output parameter list[4]

[1]By value.

[2]By reference.

[3]By descriptor.

[4]By item_list_2 descriptor.

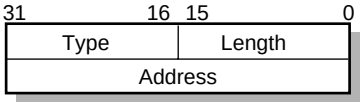
[5]By item_list_3 descriptor.

5.5 Passing Arguments by Descriptor

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

Use of an `item_list_2` or `item_list_3` argument descriptor is indicated when the argument's passing mechanism is specified as an `item_list_2` descriptor or an `item_list_3` descriptor. To determine an argument's passing mechanism, refer to the argument's description in Chapter 6.

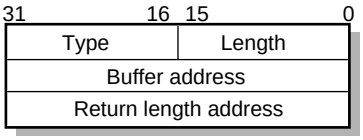
The `item_list_2` argument descriptors describe the size, data type, and starting address of a service parameter. An `item_list_2` argument descriptor contains three fields, as depicted in the following diagram:



VM-0558A-AI

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

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



VM-0559A-AI

The first field is a word containing the length (in bytes) of the buffer in which a service writes information. The length of the buffer needed depends on the data type specified in the type field. If the value of buffer length is too small, the service truncates the data. The second field is a word containing a symbolic code specifying the type of information that a service is to return. The third field is a longword containing the address of the buffer in which a service writes the information. The fourth field is a longword containing the address of a longword in which a service writes the length (in bytes) of the information it actually returned.

5.5.1 Specifying an Input Parameter List

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

To initialize an item_list_2 structure, you need to:

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

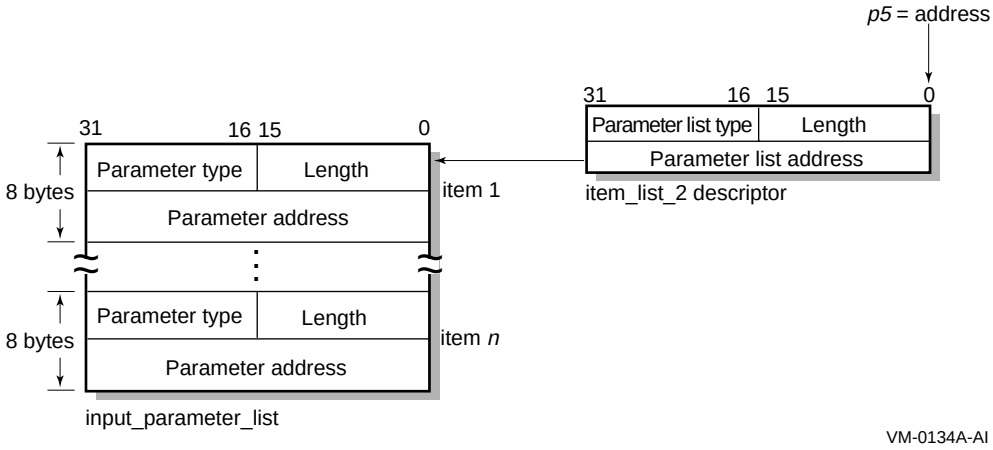
Symbolic Name	Input Parameter List Type
TCPIP$C_SOCKOPT	Socket options
TCPIP$C_TCPOPT	TCP protocol options
TCPIP$C_IPOPT	IP protocol options
TCPIP$C_IOCTL	I/O control commands

2. Set the descriptor's length field to specify the length of the input parameter list.

3. Set the descriptor's address field to specify the starting address of the input parameter list.

Figure 5–2 illustrates how the **p5** argument specifies an input parameter list.

Figure 5–2 Specifying an Input Parameter List



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

Each `item_list_2` structure that appears in an input parameter list describes an individual parameter or item to set. Such items include socket or protocol options as identified by the item's type field.

To initialize an `item_list_2` structure, you need to:

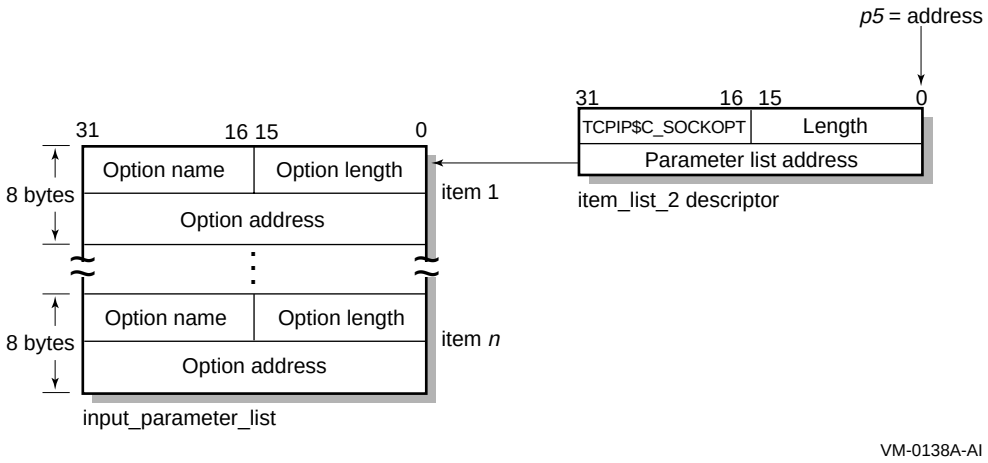
1. Set the item's type field to one of the symbolic codes found in the following tables:

   Table A–1, Socket Options
   Table A–2, TCP Protocol Options
   Table A–3, IP Protocol Options

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

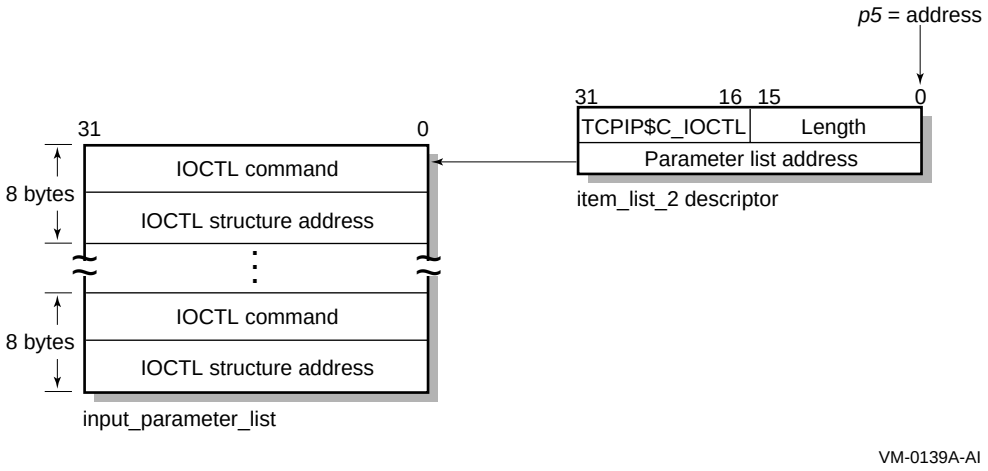
Figure 5–3 illustrates how to specify setting socket options.

Figure 5–3 Setting Socket Options



Each `ioctl_comm` structure appearing in an input parameter list contains an I/O control command—the IOCTL request code (as defined by $SIOCDEF) and its associated IOCTL structure address. Figure 5–4 illustrates how to specify (set) I/O control (IOCTL) commands.

Figure 5–4 Setting IOCTL Parameters



5.5.2 Specifying an Output Parameter List

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

To initialize an `item_list_2` structure, you need to:

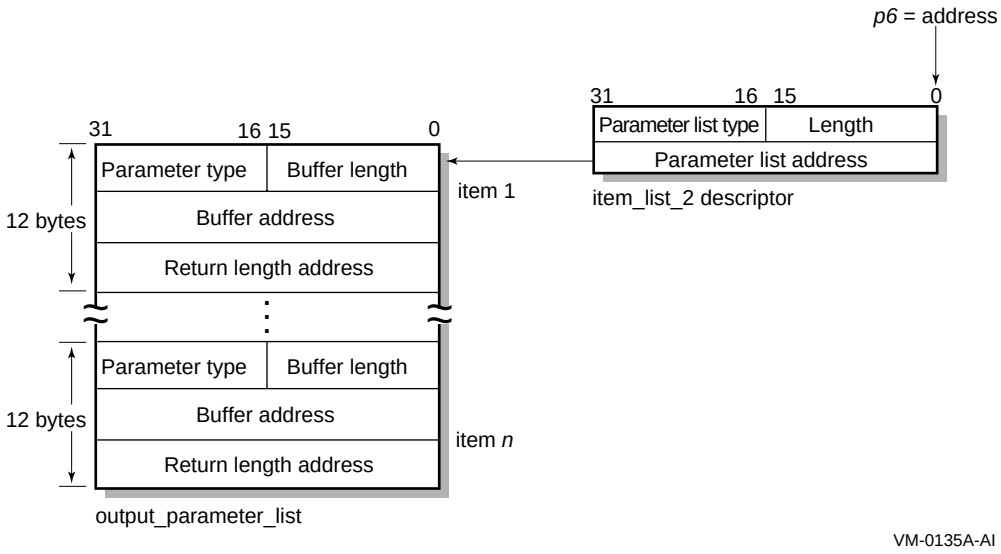
1. Set the descriptor's type field to one of the following symbolic codes to specify the type of output parameter list:

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

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

Figure 5–5 illustrates how the **p6** argument specifies an output parameter list.

Figure 5–5 Specifying an Output Parameter List



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

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

To initialize an `item_list_3` structure, you need to:

1. Set the item's type field to one of symbolic codes found in the following tables:

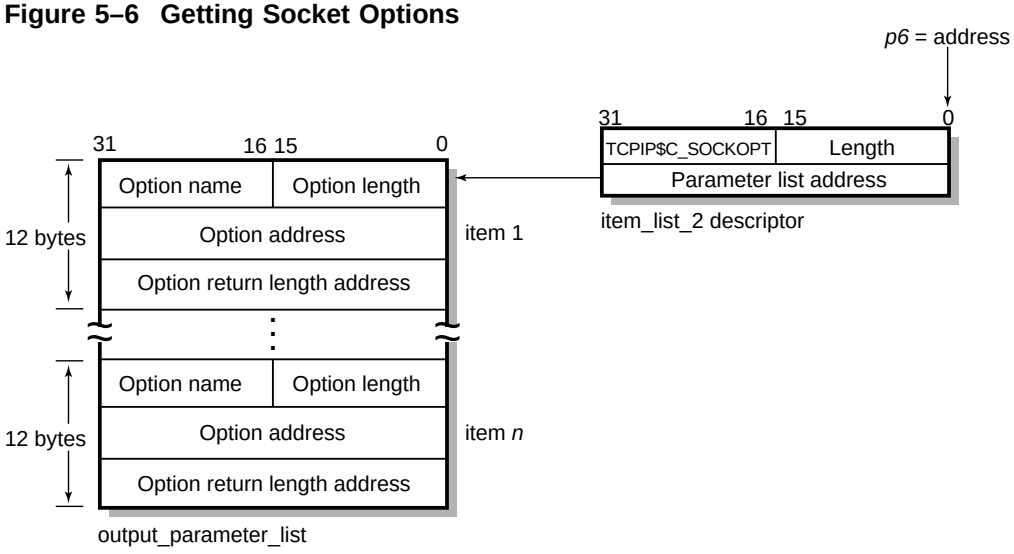
   Table A–1, Socket Options
   Table A–2, TCP Protocol Options
   Table A–3, IP Protocol Options

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

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

Figure 5–6 illustrates how to specify getting socket options.

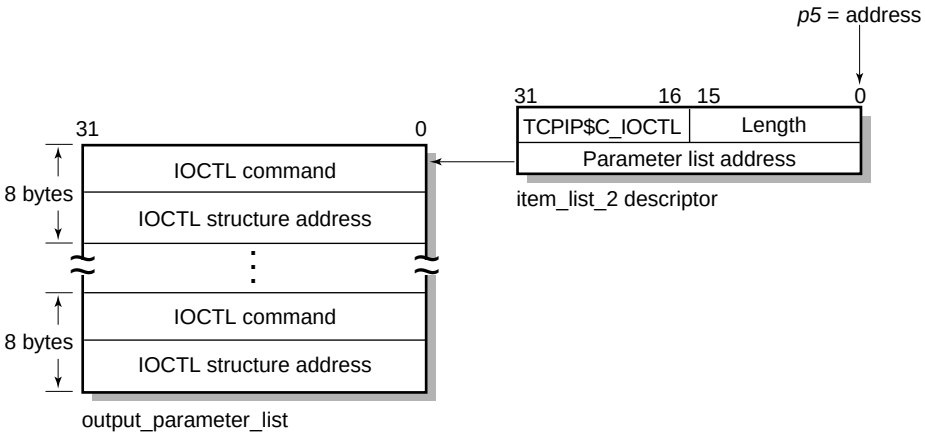
Figure 5–6 Getting Socket Options



VM-0140A-AI

Each `ioctl_comm` structure appearing in a output parameter list contains an I/O control command—the IOCTL request code (as defined by $SIOCDEF) and its associated IOCTL structure address. Figure 5–7 illustrates how to specify (get) I/O control (IOCTL) commands.

Figure 5–7 Getting IOCTL Parameters



VM-0141A-AI

5.6 Specifying a Socket Name

Use the **p3** or **p4** argument with the IO$_ACCESS, IO$_READVBLK, IO$_SENSEMODE, IO$_SETMODE, and IO$_WRITEVBLK functions to specify a socket name. The **p3** and **p4** arguments specify the address of an item_list_2 or item_list_3 descriptor that points to a socket name structure. The socket name structure contains address domain, port number, and host internet address.

Note

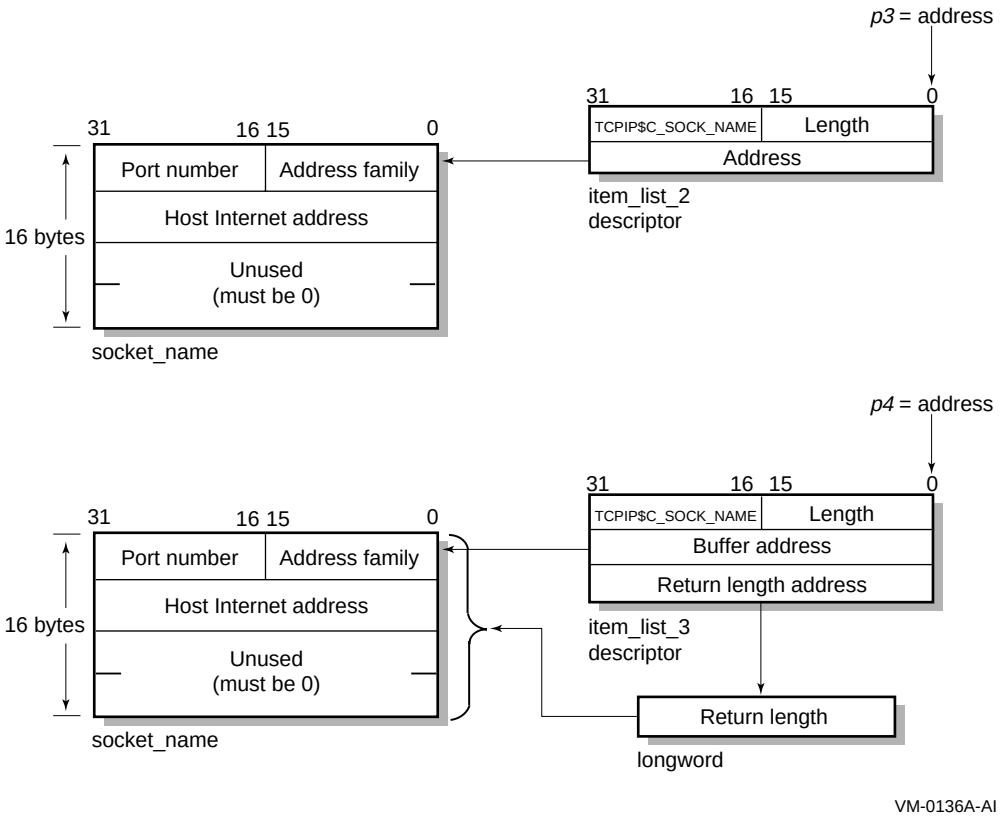
Port numbers 1 to 1023 require a system UIC or a UIC with SYSPRV and BYPASS privileges when assigned. If you specify zero when binding a socket name, the system assigns an available port.

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

Use an item_list_3 argument descriptor with the IO$_ACCESS | IO$M_ACCEPT, IO$_READVBLK, and IO$_SENSEMODE functions to specify (get) a socket name. The descriptor's parameter type is TCPIP$C_SOCK_NAME.

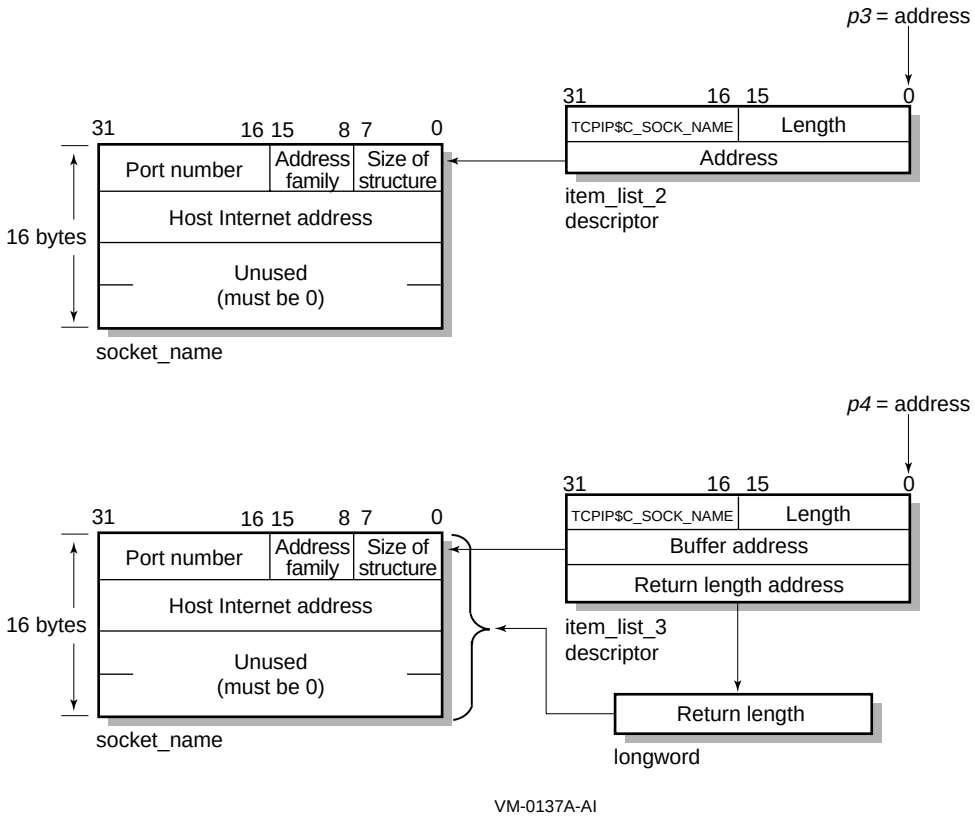
With BSD Version 4.3, specify socket names as illustrated in Figure 5–8.

Figure 5–8 Specifying a Socket Name (BSD Version 4.3)



With BSD Version 4.4, specify socket names as illustrated in Figure 5–9. Note that the first byte in the socket name is the length field. To accommodate this field, use the IO$M_EXTEND function modifier for all I/O functions that take a socket name as an output argument (IO$_ACCESS | IO$M_ACCEPT, IO$_READVBLK, and IO$_SENSEMODE).

Figure 5–9 Specifying a Socket Name (BSD Version 4.4)



5.6.1 Specifying a Buffer List

Use the **p5** argument with the IO$_WRITEVBLK function to specify input buffer lists. The **p5** argument specifies the address of a 32- or 64-bit fixed-length descriptor (on Alpha systems) or a 32-bit fixed-length descriptor (on VAX systems) pointing to an input buffer list.

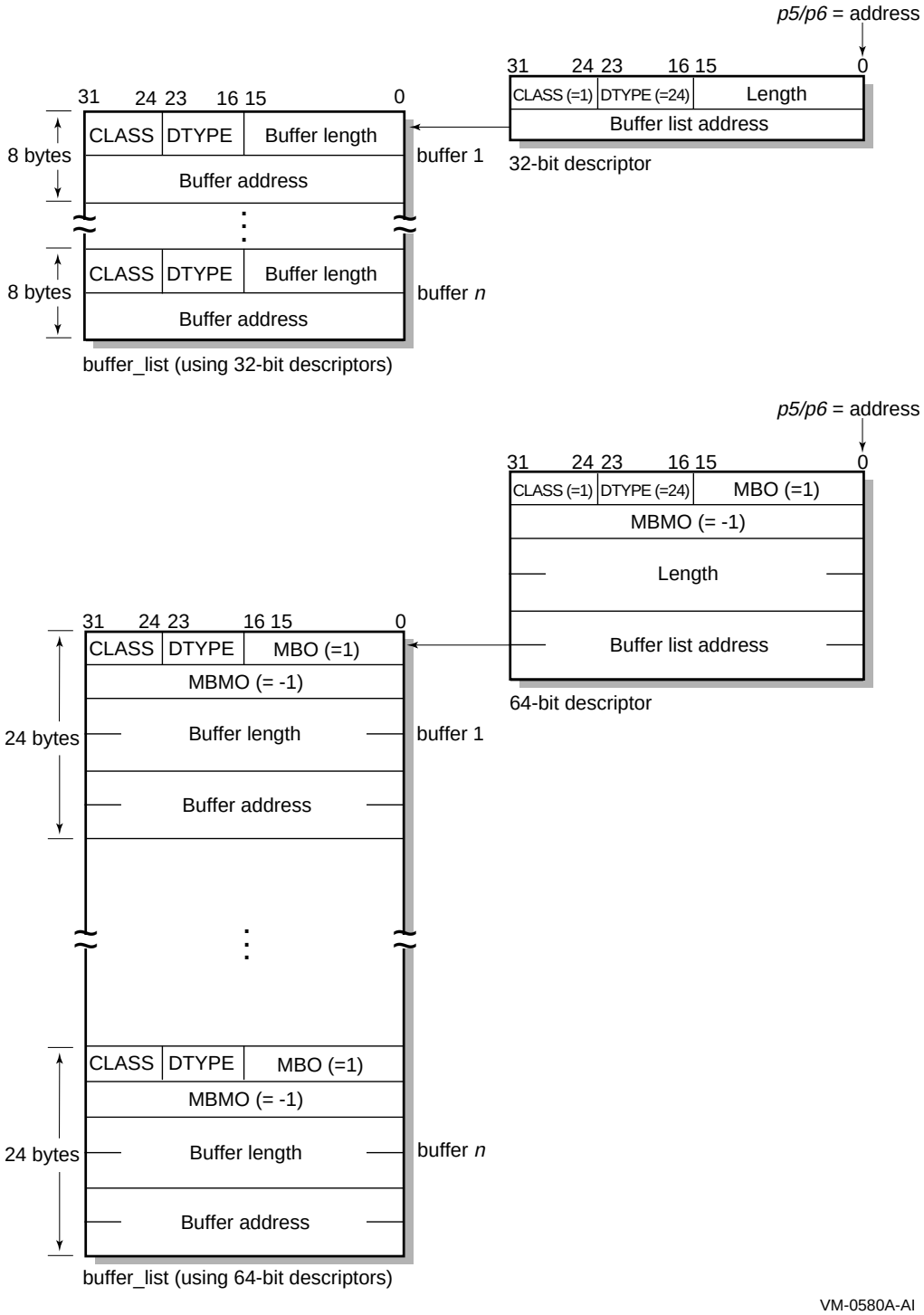
Use the **p6** argument with the IO$_READVBLK function to specify output buffer lists. The **p6** argument specifies the address of a 32- or 64-bit fixed-length descriptor (on Alpha systems) or a 32-bit fixed-length descriptor (on VAX systems) pointing to an output buffer list.

To initialize the **p5** or **p6** arugment descriptor, you need to:

1. Set the descriptor's data-type code (the DTYPE field) to DSC$K_DTYPE_DSC to specify a buffer list containing one or more descriptors defining the length and starting address of user buffers.
2. Set the descriptor's class code (the CLASS field) to DSC$K_CLASS_S.
3. Set the descriptor's length field to specify the length of the buffer list.
4. Set the descriptor's MBO field to 1 and the MBMO field to all 1s if this is a 64-bit argument descriptor.

Figure 5–10 illustrates how to specify a buffer list.

Figure 5–10 Specifying a Buffer List



Buffer lists, as the name implies, consist of one or more contiguous 32- or 64-bit fixed-length descriptors (on Alpha systems) or 32-bit fixed-length descriptors (on VAX systems).

Each 32- or 64-bit descriptor that appears in a buffer list describes one user buffer. Initialize each descriptor by setting its data type, class, length, and address fields as appropriate for 32- and 64-bit descriptors.

For more information about using 32-bit and 64-bit descriptors, refer to the OpenVMS Calling Standard.

6

OpenVMS System Services Reference

This chapter provides detailed information about the OpenVMS system services for writing network applications. The chapter also describes the network pseudodevice driver and TELNET port driver I/O functions used with the $QIO system service.

The descriptions of the system services and I/O function codes are targeted specifically for network application programmers. For a general description of these system services and I/O function codes, see the *OpenVMS System Services Reference* manuals.

Table 6–1 lists the equivalent Sockets API function for each system service and $QIO I/O function code in this chapter. See Chapter 4 for descriptions of the Sockets API functions.

Table 6–1 OpenVMS System Service and Equivalent Sockets API Function

OpenVMS System Service	Sockets API Function or Description
$ASSIGN	socket( )
$CANCEL	close( )
$DASSGN	close( )
$QIO	
Network Pseudodevice I/O Function Codes:	
IO$_ACCESS	connect( )
IO$_ACCESS \| IO$M_ACCEPT	accept( )
IO$_ACPCONTROL	gethostbyname( ), gethostbyaddr( ), getnetbyname( ), getnetbyaddr( )
IO$_DEACCESS	close( )
IO$_DEACCESS \| IO$M_SHUTDOWN	shutdown( )
IO$_READVBLK	read( ), recv( ), recvfrom( ), recvmsg( )
IO$_SENSEMODE	getsockopt( ), ioctl( ), getpeername( ), getsockname( )
IO$_SENSECHAR	getsockopt( ), ioctl( ), getpeername( ), getsockname( )
IO$_SETMODE	socket( ), bind( ), listen( ), setsockopt( ), ioctl( )

(continued on next page)

Table 6–1 (Cont.) OpenVMS System Service and Equivalent Sockets API Function

OpenVMS System Service	Sockets API Function or Description
IO$_SETCHAR	socket( ), bind( ), listen( ), setsockopt( ), ioctl( )
IO$_WRITEVBLK	send( ), sendto( ), sendmsg( ), write( )
TELNET Port Driver I/O Function Codes:	
IO$_TTY_PORT	
IO$M_TN_STARTUP	Binds a socket to a TELNET terminal device.
IO$M_TN_SHUTDOWN	Breaks a previously bound socket terminal connection.
IO$_TTY_PORT_BUFIO	
IO$M_TN_SENSEMODE	Reads parameters associated with the device.
IO$M_TN_SETMODE	Writes parameters associated with the device.

6.1 System Service Descriptions

This section describes the OpenVMS system service routines used to write network applications.

Detailed information about each argument is listed for each I/O function. The following format is used to describe each argument:

argument-name

OpenVMS usage:	OpenVMS data type
type:	argument data type
access:	argument access
mechanism:	argument passing mechanism

The purpose of the OpenVMS usage entry is to facilitate the coding of source-language data type declarations in application programs. Ordinarily, the standard data type is sufficient to describe the type of data passed by an argument. However, within the OpenVMS operating system environment, many system routines contain arguments whose conceptual nature or complexity requires additional explanation.

See Appendix C for a list of the possible OpenVMS usage entries and their definitions. Refer to the appropriate language implementation table in Appendix C to determine the correct syntax of the type declaration in the language you are using.

Note that the OpenVMS usage entry is not a traditional data type (such as the standard data types of byte, word, longword, and so on). It is significant only within the context of the OpenVMS operating system and is intended solely to expedite data declarations within application programs.

$ASSIGN—Assign I/O Channel

Provides a calling process with an I/O channel, thereby allowing the calling process to perform I/O operations on the network pseudodevice.

On Alpha systems, this service accepts 64-bit addresses.

Format

SYS$ASSIGN *devnam*, *chan*, [*acmode*], [*mbxnam*], [flags]

C Prototype

int sys$assign (void *devnam, unsigned short int *chan, unsigned int acmode, void *mbxnam,...);

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under Condition Values Returned.

Arguments

devnam

OpenVMS usage:	device_name
type:	character-coded text string
access:	read only
mechanism:	by 32- or 64-bit descriptor-fixed-length string descriptor (Alpha) by 32-bit descriptor-fixed-length string descriptor (VAX)

Name of the device to which $ASSIGN is to assign a channel. The **devnam** argument is the 32- or 64-bit address (on Alpha systems) or the 32-bit address (on VAX systems) of a character string descriptor pointing to the network pseudodevice name string (either TCPIP$DEVICE: or SYS$NET:).

chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	write only
mechanism:	by 32- or 64-bit reference (Alpha) by 32-bit reference (VAX)

Number of the channel that is assigned. The **chan** argument is the 32- or 64-bit address (on Alpha systems) or the 32-bit address (on VAX systems) of a word into which $ASSIGN writes the channel number.

acmode

OpenVMS usage:	access_mode
type:	longword (unsigned)
access:	read only
mechanism:	by value

Access mode to be associated with the channel. I/O operations on the channel can be performed only from equal and more privileged access modes. The $PSLDEF macro defines the following symbols for the four access modes:

Symbol	Access Mode	Numeric Value
PSL$C_KERNEL	Kernel	0
PSL$C_EXEC	Executive	1
PSL$C_SUPER	Supervisor	2
PSL$C_USER	User	3

mbxnam

OpenVMS usage: device_name
type: character-coded text string
access: read only
mechanism: by 32- or 64-bit descriptor-fixed-length string descriptor (Alpha)
by 32-bit descriptor-fixed-length string descriptor (VAX)

This argument is not used.

flags

OpenVMS usage: mask_longword
type: longword (unsigned)
access: read only
mechanism: by value

An optional device-specific argument. The **flags** argument is a longword bit mask. For more information about the applicability of the **flags** argument for a particular device, refer to the *OpenVMS I/O User's Reference Manual*.

Description

The $ASSIGN system service establishes a path to a device but does not check whether the calling process has the capability to do I/O operations to the device. The device drivers may apply privilege and protection restrictions. The calling process must have NETMBX privilege to assign a channel.

System dynamic memory is required for the target device, and the I/O byte limit quota from the process buffer is used.

When a channel is assigned to the TCPIP$DEVICE: template network device, the network software creates a new device called BG*n*, where *n* is a unique unit number. The corresponding channel number is used in any subsequent operation requests for that device.

When the auxiliary server creates a process for a service with the LISTEN flag set, the server creates a device socket. In order for your application to receive the device socket, assign a channel to SYS$NET, which is the logical name of a network pseudodevice, and perform an appropriate $QIO(IO$_SETMODE) operation.

Channels remain assigned either until they are explicitly deassigned with the Deassign I/O Channel ($DASSGN) service or, if they are user-mode channels, until the image that assigned the channel exits.

Condition Values Returned

SS$_NORMAL	The service completed successfully.
SS$_ACCVIO	The caller cannot read the device string or string descriptor, or the caller cannot write the channel number.
SS$_DEVALLOC	The device is allocated to another process.
SS$_DEVLSTFULL	The system maximum number of BG: device units has been reached.
SS$_EXQUOTA	The process has exceeded its buffered I/O byte limit (BIOLM) quota.
SS$_IVDEVNAM	No device name was specified, the logical name translation failed, or the device name string contains invalid characters.
SS$_IVLOGNAM	The device name string has a length of zero or has more than 63 characters.
SS$_NOIOCHAN	No I/O channel is available for assignment.
SS$_NOSUCHDEV	The specified device does not exist.

$CANCEL—Cancel I/O on Channel

Cancels all pending I/O requests on a specified channel.

Related Functions

The equivalent Sockets API function is `close( )`.

Format

SYS$CANCEL *chan*

C Prototype

int sys$cancel (unsigned short int chan);

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under Condition Values Returned.

Arguments

chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

I/O channel on which I/O is to be canceled. The **chan** argument is a word containing the channel number.

Description

To cancel I/O on a channel, the access mode of the calling process must be equal to or more privileged than the access mode of the process that made the original channel assignment.

The $CANCEL service requires system dynamic memory and uses the process's buffered I/O limit (BIOLM) quota.

When a request currently in progress is canceled, the driver is notified immediately. Actual cancellation may or may not occur immediately, depending on the logical state of the driver. When cancellation does occur, the action taken for I/O in progress is similar to that taken for queued requests. For example:

- The specified event flag is set.
- The first word of the I/O status block, if specified, is set to SS$_CANCEL if the I/O request is queued, or to SS$_ABORT if the I/O operation is in progress.
- If the asynchronous system trap (AST) is specified, it is queued.

For proper synchronization between this service and the actual canceling of I/O requests to take place, the issuing process must wait for the I/O process to complete normally. Note that the I/O has been canceled. Outstanding I/O requests are canceled automatically at image exit.

Condition Values Returned

SS$_NORMAL	The service completed successfully.
SS$_ABORT	A physical line went down during a network connect operation.
SS$_CANCEL	The I/O operation was canceled by executing a $CANCEL system service.
SS$_EXQUOTA	The process has exceeded its buffered I/O limit (BIOLM) quota.
SS$_INSFMEM	Insufficient system dynamic memory to cancel the I/O.
SS$_IVCHAN	An invalid channel was specified (that is, a channel number of 0 or a number larger than the number of channels available).
SS$_NOPRIV	The specified channel is not assigned or was assigned from a more privileged access mode.

$DASSGN—Deassign I/O Channel

Deassigns (releases) an I/O channel previously acquired using the Assign I/O Channel ($ASSIGN) service.

Related Functions

The equivalent Sockets API function is `close( )`.

Format

SYS$DASSGN *chan*

C Prototype

int sys$dassgn (unsigned short int chan);

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under Condition Values Returned.

Arguments

chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

Number of the I/O channel to be deassigned. The **chan** argument is a word containing this number.

Description

After all communication is completed, use the $DASSGN system service to free an I/O channel. A $DASSGN operation executed on a channel associated with a network pseudodevice does the following:

- Ends all pending operations to send or receive data at $QIO level ($CANCEL system service).
- Clears the port associated with the channel. When executing the $DASSGN system service for TCP sockets, the socket remains until the connection is closed on both the local and remote sides.
- Ends all communications with the network pseudodevice that the I/O channel identifies.
- Frees the channel associated with the network pseudodevice. An I/O channel can be deassigned only from an access mode equal to or more privileged than the access mode from which the original channel assignment was made.

I/O channels assigned from user mode are automatically deassigned at image exit.

Note

Even after a $DASSGN has been issued, a TCP socket may remain until the TCP close timeout interval expires. The default and maximum timeout interval is either 10 minutes if the peer host is not responding or 30 seconds after acknowledging the socket close. Although the TCP socket is open, you cannot make a reference to that socket after issuing a $DASSGN.

Condition Values Returned

SS$_NORMAL	The service completed successfully.
SS$_IVCHAN	An invalid channel number was specified (that is, a channel number of zero or a number larger than the number of channels available).
SS$_NOPRIV	The specified channel is not assigned or is assigned from a more privileged access mode.

$QIO—Queue I/O Request

Queues an I/O request to a channel associated with a network pseudodevice.

The $QIO service is completed asynchronously; that is, it returns to the caller immediately after queuing the I/O request, without waiting for the I/O operation to be completed.)

For synchronous completion, use the Queue I/O Request and Wait ($QIOW) service. The $QIOW service is identical to the $QIO service, except the $QIOW returns to the caller after the I/O operation has completed.

On Alpha systems, this service accepts 64-bit addresses.

Format

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

C Prototype

int sys$qio (unsigned int efn, unsigned short int chan, unsigned int func, struct _iosb *iosb, void (*astadr)(__unknown_params), __int64 astprm, void *p1, __int64 p2, __int64 p3, __int64 p4, __int64 p5, __int64 p6);

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All system services return (by immediate value) a condition value in R0. Condition values that can be returned by this service are listed under Condition Values Returned.

Arguments

efn

OpenVMS usage:	ef_number
type:	longword (unsigned)
access:	read only
mechanism:	by value

Event flag that $QIO sets when the I/O operation completes. The **efn** argument is a longword value containing the number of the event flag, however, $QIO uses only the low-order byte.

If **efn** is not specified, event flag 0 is set.

The specified event flag is set if the service terminates without queuing an I/O request.

chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

I/O channel that is assigned to the device to which the request is directed. The **chan** argument is a word value containing the number of the I/O channel.

func

OpenVMS usage:	function_code
type:	longword (unsigned)
access:	read only
mechanism:	by value

Function codes and function modifiers specifying the operation to be performed. The **func** argument is a longword containing the function code.

For information about the network pseudodevice and TELNET device function codes and modifiers, see Section 6.2 and Section 6.3.

iosb

OpenVMS usage:	io_status_block
type:	quadword (unsigned)
access:	write only
mechanism:	by 32-bit reference or 64-bit reference (Alpha) by 32-bit reference (VAX)

I/O status block to receive the final completion status of the I/O operation. The **iosb** is the address of the quadword I/O status block. See Figure 5–1 for a description of the general structure of the I/O status block.

When the $QIO begins executing, it clears the event flag. The $QIO also clears the quadword I/O status block if the **iosb** argument is specified.

Although the **iosb** argument is optional, Compaq strongly recommends that you specify it, for the following reasons:

- If you are using an event flag to signal the completion of the service, you can test the I/O status block for a condition value to be sure that the event flag was not set by an event other than service completion.
- If you are using the $SYNCH service to synchronize completion of the service, the I/O status block is a required argument for $SYNCH.
- The condition value returned in R0 and the condition value returned in the I/O status block provide information about different aspects of the call to the $QIO service. The condition value returned in R0 provides information about the success or failure of the service call itself; the condition values returned in the I/O status block give information on the success or failure of the service operation. Therefore, to access the success or failure of the $QIO call, check the condition values returned in both the R0 and the I/O status block.

astadr

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by 32- or 64-bit reference (Alpha) by 32-bit reference (VAX)

AST service routine to be executed when the I/O completes. The **astadr** argument is the address of the AST routine.

The AST routine executes at the access mode of the caller of $QIO.

astprm

OpenVMS usage:	user_arg
type:	quadword unsigned (Alpha); longword unsigned (VAX)
access:	read only
mechanism:	by 32- or 64-bit value (Alpha) by 32-bit value (VAX)

AST parameter to be passed to the AST service routine. On Alpha systems, the **astprm** argument is a quadword value containing the AST parameter. On VAX systems, the **astprm** argument is a longword value containing the AST parameter.

p1 to p6

OpenVMS usage:	varying_arg
type:	quadword unsigned (Alpha); longword unsigned (VAX)
access:	read only
mechanism:	by 32- or 64-bit reference or by 64-bit value depending on the I/O function (Alpha) by 32-bit reference or by 32-bit value depending on the I/O function (VAX)

Optional device- and function-specific I/O request arguments. The parameter values contained in these arguments vary according to the function for which they are used. See Table 6–2 for descriptions of the network pseudodevice driver I/O function codes; see Table 6–7 through Table 6–10 for related TELNET device driver I/O function codes.

Description

The Queue I/O Request service operates only on assigned I/O channels and only from access modes that are equal to or more privileged than the access mode from which the original channel assignment was made.

For TCP/IP Services, $QIO uses the following system resources:

- The process's AST limit (ASTLM) quota, if an AST service routine is specified.
- System dynamic memory, which is required to queue the I/O request. System dynamic memory requirements are protocol specific.
- Additional memory, on a device-dependent basis.

For $QIO, completion can be synchronized as follows:

- By specifying the **astadr** argument to have an AST routine execute when the I/O is completed.
- By calling the $SYNCH synchronize service to await completion of the I/O operation. (If you want your I/O operation to complete synchronously, use the $QIOW system service instead.)

Condition Values Returned

Each function used with $QIO has its own error codes. See the error codes listed under the individual descriptions of the device driver I/O function code in the remainder of this chapter.

6.2 Network Pseudodevice Driver I/O Functions

The network pseudodevice allows physical, logical, and virtual I/O functions. The physical and logical I/O functions are used only with the IP layer. Table 6–2 lists the basic I/O functions and their modifiers. The sections that follow describe in greater detail the operation of these I/O functions.

Table 6–2 Network Pseudodevice Driver I/O Functions

Function Code and Arguments	Function Modifier	Description
IO$_ACCESS **p3,p4**	IO$M_ACCEPT IO$M_EXTEND IO$M_NOW	Opens a connection.
IO$_ACPCONTROL **p1, p2, p3, p4**		Performs an ACP (ancillary control process) operation.
IO$_DEACCESS **p4**	IO$M_NOW IO$M_SHUTDOWN	Aborts or closes a connection.
IO$_READVBLK **p1,p2,p3,p4,p6**	IO$M_EXTEND IO$M_INTERRUPT	Reads a virtual block.
	IO$M_LOCKBUF IO$M_PURGE	Controls the buffer operations.
IO$_SENSEMODE **p2**,**p3,p4,p6**		Reads the network pseudodevice characteristics.
IO$_SENSECHAR **p2**,**p3,p4,p6**		Reads the network pseudodevice characteristics.
IO$_SETMODE **p1**,**p2**, **p3,p4,p5**	IO$M_OUTBAND IO$M_READATTN IO$M_WRTATTN	Sets the network pseudodevice characteristics for subsequent operations.
IO$_SETCHAR **p1**,**p2**, **p3,p4,p5**	IO$M_OUTBAND IO$M_READATTN IO$M_WRTEATTN	Sets the network pseudodevice characteristics for subsequent operations.
IO$_WRITEVBLK **p1,p2,p3,p4,p5**	IO$M_INTERRUPT	Writes a virtual block.

Table 5–2 lists the file names of the symbol definition files. These files specify $QIO arguments (**p1,p2,...p6**) for applications written in the corresponding programming languages. You must invoke the symbol definition by using the appropriate `include` statement in your application.

IO$_ACCESS

When using a connection-oriented protocol, such as TCP, the IO$_ACCESS function initiates a connection and specifies a remote port number and internet address. When using a connectionless protocol, such as UDP, the IO$_ACCESS function sets the remote port number and internet address.

For TCP, a connection request times out at a specified interval (75 seconds is the default). This interval can be changed by the system manager. The program can also set a specific timeout interval for a socket that it has created.

If a connection fails, you must deallocate the socket and then create a new socket before trying to reconnect.

Related Functions

The equivalent Sockets API function is `connect( )`.

Arguments

p3

OpenVMS usage:	socket_name
type:	vector byte (unsigned)
access:	read only
mechanism:	by `item_list_2` descriptor

The remote port number and internet address of the host to connect. The **p3** argument is the address of an `item_list_2` descriptor that points to the socket address structure containing the remote port number and internet address.

Function Modifiers

IO$M_NOW	Regardless of a $QIO or $QIOW, if the system detects a condition that would cause the operation to block, the system completes the I/O operation and returns the SS$_SUSPENDED status code.

Condition Values Returned

SS$_NORMAL	The service completed successfully.
SS$_BADPARAM	Programming error that occurred for one of the following reasons: • $QIO system service was specified without a socket. • An IO$_ACCESS function was specified without the address of a remote socket name (**p3** was null).
SS$_BUGCHECK	Inconsistent state. Report the problem to your Compaq support representative.
SS$_CANCEL	The I/O operation was canceled by a $CANCEL system service.

SS$_CONNECFAIL	The connection to a network object timed out or failed.
SS$_DEVINTACT	The network driver was not started.
SS$_DEVNOTMOUNT	The network driver is loaded, but the INETACP is not currently available for use.
SS$_DUPLNAM	A network configuration error. No ports were available for new connections.
SS$_EXQUOTA	The process has exceeded its socket quota or some other process quota.
SS$_FILALRACC	The specified socket name is already in use by one of the following: • On a raw socket, the remote internet address was already specified on a previous IO$_ACCESS call. • On a datagram, the remote internet address was already specified on a previous IO$_ACCESS call. • On a stream socket, the IO$_ACCESS function targeted a stream socket that was already connected.
SS$_ILLCNTRFUNC	Illegal function.
SS$_INSFMEM	Insufficient system dynamic memory to complete the service.
SS$_IVADDR	The specified internet address was not found, or an invalid port number and internet address combination was specified with the IO$_ACCESS function. Port 0 is not allowed with the IO$_ACCESS function.
SS$_IVBUFLEN	The size of the socket name structure specified with the IO$_ACCESS function was invalid.
SS$_LINKABORT	The remote socket closed the connection.
SS$_NOLICENSE	The Compaq TCP/IP Services for OpenVMS license is not present.
SS$_PROTOCOL	A network protocol error occurred. The address family specified in the socket address structure is not supported.
SS$_REJECT	The network connection is rejected for one of the following reasons: • An attempt was made to connect to a remote socket that is already connected. • An error was encountered while establishing the connection • The peer socket refused the connection request or is closing the connection.

SS$_SHUT	The local or remote node is no longer accepting connections.
SS$_SUSPENDED	The system detected a condition that might cause the operation to block.
SS$_TIMEOUT	A TCP connection timed out before the connection could be established.
SS$_UNREACHABLE	The remote node is currently unreachable.

IO$_ACCESS|IO$M_ACCEPT

This function is used with a connection-based protocol, such as TCP, to accept a new connection on a passive socket.

This function completes the first connection on the queue of pending connections.

Related Functions

The equivalent Sockets API function is `accept()`.

Arguments

p3

OpenVMS usage:	socket_name
type:	vector byte (unsigned)
access:	read only
mechanism:	by `item_list_3` descriptor

The remote port number and internet address of a new connection. The **p3** argument is the address of an `item_list_3` descriptor that points to the socket address structure into which the remote port number and internet address of the new connection is written.

Use the IO$_ACCESS function with the IO$M_EXTEND modifier to specify a BSD Version 4.4 formatted socket address structure.

p4

OpenVMS usage:	channel
type:	word (unsigned)
access:	write only
mechanism:	by reference

The I/O channel number assigned to a new connection. The **p4** argument is the address of a word into which the new connection's channel number is written.

Function Modifiers

IO$M_EXTEND	Allows the usage of BSD Version 4.4 formatted socket address structures.
IO$M_NOW	Regardless of a $QIO or $QIOW, if the system detects a condition that would cause the operation to block, the system completes the I/O operation and returns the SS$_SUSPENDED status code.

Condition Values Returned

SS$_NORMAL	The service completed successfully.

SS$_BADPARAM	Programming error that occurred for one of the following reasons: • $QIO system service was specified without a socket. • A IO$_ACCESS | IO$M_ACCEPT function was specified without the address of the channel for the new connection (**p4** was null or invalid).
SS$_BUGCHECK	Inconsistent state. Report the problem to your Compaq support representative.
SS$_CANCEL	The I/O operation was canceled by a $CANCEL system service.
SS$_DEVINTACT	The network driver was not started.
SS$_DEVNOTMOUNT	The network driver is loaded, but the INETACP is not currently available for use.
SS$_EXQUOTA	The process has exceeded its socket quota or some other process quota.
SS$_FILALRACC	The specified socket name is already in use by one of the following: • On a raw socket, the remote internet address was already specified on a previous IO$_ACCESS call. • On a datagram, the remote internet address was already specified on a previous IO$_ACCESS call. • On a stream socket, the IO$_ACCESS function targeted a stream socket that was already connected.
SS$_ILLCNTRFUNC	Illegal function.
SS$_INSFMEM	Insufficient system dynamic memory to complete the service.
SS$_IVADDR	The specified internet address was not found, or an invalid port number and internet address combination was specified with the IO$_ACCESS function. Port 0 is not allowed with the IO$_ACCESS function.
SS$_IVBUFLEN	The size of the socket name structure specified with the IO$_ACCESS function was invalid.
SS$_LINKABORT	The remote socket closed the connection.
SS$_NOLICENSE	The TCP/IP Services license is not present.
SS$_PROTOCOL	A network protocol error occurred. The address family specified in the socket address structure is not supported.

SS$_REJECT	The network connection is rejected for one of the following reasons: • An attempt was made to connect to a remote socket that is already connected. • An error was encountered while establishing the connection • The peer socket refused the connection request or is closing the connection.
SS$_SHUT	The local or remote node is no longer accepting connections.
SS$_SUSPENDED	The system detected a condition that might cause the operation to block.
SS$_TIMEOUT	A TCP connection timed out before the connection could be established.
SS$_UNREACHABLE	The remote node is currently unreachable.

IO$_ACPCONTROL

The IO$_ACPCONTROL function accesses the network ACP to retrieve information from the host and the network database files.

Related Functions

The equivalent Sockets API functions are `gethostbyaddr( )`, `gethostbyname( )`, `getnetbyaddr( )`, and `getnetbyname( )`.

Arguments

p1

OpenVMS usage:	subfunction_code
type:	longword (unsigned)
access:	read only
mechanism:	by descriptor-fixed-length descriptor

A longword identifying the network ACP operation to perform. The **p1** argument is the address of a descriptor pointing to this longword.

To specify the network ACP operation to perform, select a subfunction code from Table 6–3 and a call code from Table 6–4.

Table 6–3 defines subfunction codes for network ACP operations.

Table 6–3 Subfunction Codes

Subfunction Code	Description
INETACP_FUNC$C_GETHOSTBYADDR	Get the host name of the specified internet address from the host database.
INETACP_FUNC$C_GETHOSTBYNAME	Get the internet address of the specified host from the host database.
INETACP_FUNC$C_GETNETBYADDR	Get the network name of the specified internet address from the network database.
INETACP_FUNC$C_GETNETBYNAME	Get the internet address of the specified network from the network database.

Table 6–4 defines call codes for network ACP operations.

Table 6–4 Call Codes

Call Code	Description
INETACP$C_ALIASES	Returns the list of alias names associated with the specified host or network from the internet hosts or network database.

(continued on next page)

Table 6–4 (Cont.) Call Codes

Call Code	Description
INETACP$C_TRANS	Returns the internet address associated with the specified host or network as a 32-bit value in network byte order.
INETACPC$C_HOSTENT_OFFSET	Returns full host information in a modified `hostent` structure. In the modified structure, pointers are replaced with offsets from the beginning of the structure.
INETACP$C_NETENT_OFFSET	Returns full network information in a modified `netent` structure. In the modified structure, pointers are replaced with offsets from the beginning of the structure.

IO$_ACPCONTROL searches the local host database for the host's name. If a matching host name is not found in the local host database, IO$_ACPCONTROL then searches the BIND database if the BIND resolver is enabled.

p2

OpenVMS usage: char_string
type: character-coded text string
access: read only
mechanism: by descriptor-fixed-length string descriptor

Input string for the network ACP operation containing one of the following: host internet address, host name, network internet address, or network name. The **p2** argument is the address of a string descriptor pointing to the input string.

All internet addresses are specified in dotted-decimal notation.

p3

OpenVMS usage: word_unsigned
type: word (unsigned)
access: write only
mechanism: by reference

Length in bytes of the output buffer returned by IO$_ACPCONTROL. The **p3** argument is the address of a word in which the length of the output buffer is written.

p4

OpenVMS usage: buffer
type: vector byte (unsigned)
access: write only
mechanism: by descriptor-fixed-length descriptor

Buffer into which IO$_ACPCONTROL writes its output data. The **p4** argument is the address of a descriptor pointing to the output buffer.

The format of the data returned in the output buffer is dictated by the call code specified by the **p1** argument.

- Strings returned by IO$_ACPCONTROL with a call code of INETACP$C_ ALIASES consist one of the following: host internet address, host name, network internet address, or network name. All internet addresses are formatted using dotted-decimal notation. Alias names are separated by a null character (0). The length of the returned string includes all null characters that separate alias names.
- Internet addresses returned by IO$_ACPCONTROL with a call code of INETACP$C_TRANS are 32-bit value and in network byte order.
- All `hostent` and `netent` structures returned by IO$_ACPCONTROL with a call code of INETACP$C_HOSTENT_OFFSET or INETACP$C_NETENT_ OFFSET are modified; pointers are replaced with offsets from the beginning of the structure.

Condition Values Returned

SS$_NORMAL	The service completed successfully
SS$_ABORT	An error was detected while performing an ACP function.
SS$_BADPARAM	Programming or internal error. A bad parameter (name or address) was specified in a GET{HOST,NET}BY{NAME,ADDRESS} ACP call.
SS$_BUFFEROVF	Programming error. There was not enough space for returning all alias names in a GET{HOST,NET}BY{NAME,ADDRESS} ACP call.
SS$_ENDOFFILE	The information requested is not in the database.
SS$_ILLCNTRFUNC	Illegal function.
SS$_NOPRIV	No privilege for the execution of an ACP function.
SS$_RESULTOVF	The ACP overflowed the buffer in returning a parameter.
SS$_SHUT	The local or remote node is no longer accepting connections.

IO$_DEACCESS

The IO$_DEACCESS function closes a connection and deletes a socket. Any pending messages queued for transmission are sent before tearing down the connection.

When used with the IO$M_SHUTDOWN function modifier, the IO$_DEACCESS function shuts down all or part of a bidirectional connection on a socket. Use the **p4** argument to specify the disposition of pending I/O operations on the socket.

You can specify a wait time or time-to-linger socket parameter (TCPIP$C_LINGER option) for transmission completion before disconnecting the connection. Use the IO$_SETMODE or IO$_SETCHAR function to set and clear the TCPIP$C_LINGER option.

If you set the TCPIP$C_LINGER option, a $QIO call that uses the IO$_DEACCESS function allows data queued to the socket to arrive at the destination. The system is blocked until data arrives at the remote socket. The socket data structure remains open for the duration of the TCP idle time interval.

If you do not set the TCPIP$C_LINGER option (option is set to 0), a $QIO call that uses the IO$_DEACCESS function discards any data queued to the socket and deallocates the socket data structure.

Note

For compatibility with Compaq *Tru64* UNIX, the TCP/IP Services forces a time to linger of 2 minutes on TCP stream sockets.

Related Functions

The equivalent Sockets API functions are `close( )` and `shutdown( )`.

Arguments

p4

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by value

Longword of shutdown flags to specify the disposition of pending I/O operations on the socket. The **p4** argument is used only with the IO$M_SHUTDOWN function modifier. The following table lists available shutdown flags.

Shutdown Flag	Description
TCPIP$C_DSC_RCV	Discards messages from the receive queue and disallows further receiving. Pending messages in the receive queue for this connection are discarded.
TCPIP$C_DSC_SND	Discards messages from the send queue and disallows sending new messages. Pending messages in the transmit queue for this connection are discarded.

Shutdown Flag	Description
TCPIP$C_DSC_ALL	Discards all messages and disallows both sending and receiving. All pending messages are discarded. Specifying this flag has the same effect as issuing a $CANCEL QIO followed by an IO$_DEACCESS QIO without specifying any flags.

Function Modifiers

IO$M_SHUTDOWN	Causes all or part of a full-duplex connection on a socket to be shut down.
IO$M_NOW	Regardless of a $QIO or $QIOW, if the system detects a condition that would cause the operation to block, the system completes the I/O operation and returns the SS$_SUSPENDED status code.

Condition Values Returned

SS$_NORMAL	The service completed successfully.
SS$_BADPARAM	The IO$_DEACCESS operation failed to specify a socket.
SS$_CANCEL	The I/O operation was canceled by a $CANCEL system service.
SS$_DEVINTACT	The network driver was not started.
SS$_DEVNOTMOUNT	The network driver is loaded, but the INETACP is not currently available for use.
SS$_NOLINKS	The specified socket was not connected.
SS$_SHUT	The local or remote node is no longer accepting connections.
SS$_SUSPENDED	The system detected a condition that might cause the operation to block.

IO$_READVBLK

The IO$_READVBLK function transfers data received from an internet host to the specified user buffers. Use both **p1** and **p2** arguments to specify a single user buffer. Use the **p6** argument to specify multiple buffers.

For connection-oriented protocols, such as TCP, data is buffered in system space as a stream of bytes. The IO$_READVBLK function completes (1) when there is no more data buffered in system space for this socket, or (2) when there is no more available space in the user buffer. Data that is buffered in system space but did not fit in the user buffer is available to the user in subsequent $QIOs.

For connectionless protocols, datagram and raw socket data is buffered in system space as a chain of records. The user buffer specified with a IO$_READVBLK function is filled with data that is buffered in one record. Each IO$_READVBLK reads data from one record. The IO$_READVBLK function completes (1) when all data from a record is transferred to the user buffer, or (2) when there is no more available space in the user buffer. Any data remaining in the current record that did not fit in the user buffer is discarded. A subsequent $QIO reads data from the next record buffered in system space.

Use the management command SHOW DEVICE_SOCKET/FULL to display counters related to read operations.

Related Functions

The equivalent Sockets API functions are `read( )`, `recv( )`, `recvfrom( )`, and `recvmsg( )`.

Arguments

p1

OpenVMS usage:	buffer
type:	vector byte (unsigned)
access:	read only
mechanism:	by 32- or 64-bit reference (Alpha) by 32-bit reference (VAX)

The 32- or 64-bit address (on Alpha systems) or the 32-bit address (on VAX systems) of the buffer to receive the incoming data. The length of this buffer is specified by the **p2** argument.

p2

OpenVMS usage:	buffer_length
type:	quadword unsigned (Alpha); longword unsigned (VAX)
access:	read only
mechanism:	by 64-bit value (Alpha) by 32-bit value (VAX)

The length (in bytes) of the buffer available to hold the incoming data. The address of this buffer is specified by the **p1** argument.

p3

OpenVMS usage:	socket_name
type:	vector byte (unsigned)
access:	read only
mechanism:	by `item_list_3` descriptor

The remote port number and internet address of the source of the datagram or raw IP message (not TCP). The **p3** argument is the address of an item_list_3 descriptor that points to the socket address structure into which the remote port number and internet address of the message source is written.

Use the IO$_READVBLK function with the IO$M_EXTEND modifier to specify a BSD Version 4.4 formatted socket address structure. If the IO$M_EXTEND modifier is not specified, the IO$_READVBLK function returns a BSD Version 4.3 formatted socket address structure.

p4

OpenVMS usage: mask_longword
type: longword (unsigned)
access: read only
mechanism: by value

Longword of flags to specify attributes for the read operations. Table 6–5 lists the available read flags.

Table 6–5 Read Flags

Read Flag	Description
TCPIP$C_MSG_OOB	Reads an out-of-band byte.
TCPIP$C_MSG_PEEK	Reads a message but leaves the message in the queue.
TCPIP$C_MSG_NBIO	Does not block the I/O operation if the receive queue is empty (similar to using IO$M_NOWAIT).
TCPIP$C_MSG_PURGE	Flushes data from the queue (similar to using IO$M_PURGE).
TCPIP$C_MSG_BLOCKALL	Blocks the completion of the operation until the buffer is filled completely or until the connection is closed (similar to using IO$M_LOCKBUF).

p6

OpenVMS usage: buffer_list
type: vector byte (unsigned)
access: read only
mechanism: by 32- or 64-bit descriptor-fixed-length descriptor (Alpha)
by 32-bit descriptor-fixed-length descriptor (VAX)

Output buffer list describing one or more buffers to hold the incoming data. The **p6** argument is the 32- or 64-bit address (on Alpha systems) or the 32-bit address (on VAX systems) of a descriptor that points to a output buffer list. Buffers are filled in the order specified by the output buffer list. The transfer-length value returned in the I/O status block is the total number of bytes transferred to all buffers.

If you use the **p1** and **p2** arguments, do not use the **p6** argument; they are mutually exclusive.

Function Modifiers

IO$M_EXTEND	Specifies the format of the socket address structure to return when used with the **p3** argument. When specified, a BSD Version 4.4 formatted socket address structure is returned that identifies the source of the received UDP datagram or raw IP message.
IO$M_INTERRUPT	Reads an out-of-band (OOB) message. This has the same effect as specifying the TCPIP$C_MSG_OOB flag in the **p4** argument. On receiving a TCP/IP OOB character, TCP/IP stores the pointer in the received stream with the character that precedes the OOB character. A read operation with a user-buffer size larger than the size of the received stream up to the OOB character completes and returns to the user the received stream up to, but not including, the OOB character. To determine whether the socket must issue more read $QIOs before getting all the characters from the stream preceding an OOB character, poll the socket. To do this, issue a $QIO with the $IO_SENSEMODE function, and the TCPIP$C_IOCTL subfunction that specifies the SIOCATMARK command. The SIOCATMARK values are as follows: • 0 = Issue more read QIOs to read more data before reading the OOB. • 1 = The next read QIO will return the OOB. Polling a socket is particularly useful when the OOBINLINE socket option is set. When the OOBINLINE is set, TCP/IP reads the OOB character with the characters in the stream (IO$_READVBLK), but not before reading the preceding characters. Use this polling mechanism to determine whether the first character in the user buffer on the next read is an OOB character.

	On a socket without the OOBINLINE option set, a received OOB will always be read by issuing a $QIO with either the IO$_READVBLK | IO$M_INTERRUPT or IO$_READVBLK and the TCPIP$C_MSG_OOB flag set. This can occur regardless of how many preceding characters in the stream have been returned to the user.
IO$M_LOCKBUF	Blocks the completion of the I/O operation until the user buffer is completely filled or until the connection is closed. This is particularly useful when you want to minimize the number of $QIO service calls issued to read a data stream of a set size. This function modifier supports only stream protocols.
IO$M_NOWAIT	Regardless of a $QIO or $QIOW, if the system detects a condition that would cause the operation to block, the system completes the I/O operation and returns the SS$_SUSPENDED status code.
IO$M_PURGE	Flushes data from the socket receive queue (discards data). If the user buffer is larger than the amount of data in the queue, all data is flushed.

Condition Values Returned

SS$_NORMAL	The service completed successfully.
SS$_ABORT	Programming error, INET management error, or hardware error. The execution of the I/O was aborted.
SS$_ACCVIO	Access to an invalid memory location or buffer occurred.

SS$_BADPARAM	One of the following methods was used to specify a $QIO function with an invalid parameter: • An I/O function executed without specifying a device socket. First issue a $QIO with the IO$_SETMODE function and the proper parameters to create the device socket. • An IO$_READVBLK function that does not specify a correct buffer address (**p1** or **p6** is null). • An IO$_READVBLK function specified an invalid vectored buffer (**p6** is an invalid descriptor). • The socket has the OOBINLINE option set, or there is no OOB character in the socket's OOB queue because the character was either already read or never received. This condition happens only if you use the IO$M_INTERRUPT modifier or set the TCPIP$C_MSG_OOB flag with IO$_READVBLK.
SS$_CANCEL	The I/O operation was canceled by a $CANCEL system service.
SS$_DEVINTACT	The network driver was not started.
SS$_DEVNOTMOUNT	The network driver is loaded, but the INETACP is not currently available for use.
SS$_INSFMEM	INET management or programming error. There is not enough buffer space for allocation. The INET software needs more buffer space. You should set a higher quota for the dynamic buffer space, or shut down and restart your internet with a larger static buffer space.
SS$_IVBUFLEN	Programming error occurred for one of the following reasons: • The size of the buffer for an I/O function is insufficient. • An IO$_READVBLK specified a correct buffer address (**p1** valid), but does not specify a buffer length (**p2** is null).
SS$_LINKDISCON	A virtual circuit (TCP/IP) was closed at the initiative of the peer.
SS$_NOLINKS	Programming error. Read attempt on unconnected TCP socket.
SS$_SHUT	The network is being shut down.

SS$_SUSPENDED	The operation is blocked for one of the following reasons: • No messages were received, so the receive operation cannot complete. The socket is marked as nonblocking. • The socket has the OOBINLINE option clear, and the OOB character has already been read.
SS$_TIMEOUT	This applies to a socket that has KEEPALIVE set. The connection was idle for longer than the timeout interval (10 minutes is the default).
SS$_UNREACHABLE	Communication status. The remote host or network is unreachable.

IO$_SENSEMODE/IO$_SENSECHAR

The IO$_SENSEMODE and IO$_SENSECHAR functions return one or more parameters (characteristics) pertaining to the network driver.

Socket names (local and remote peer) are returned by using IO$_SENSEMODE's **p3** and **p4** arguments. Other parameters such as socket and protocol options, are specified in an output parameter list using the IO$_SENSEMODE **p6** argument.

IO$_SENSEMODE **p3** and **p4** arguments can be used with the **p6** argument in a single $QIO system service to return socket names as well as socket and protocol options. IO$_SENSEMODE processes arguments in this order: **p3**, **p4**, **p6**. If IO$_SENSEMODE detects an error, the IOSB contains the error and argument address or the value that was at fault.

Refer to individual argument descriptions for details about specifying the type and format of output parameters.

Arguments

p3

OpenVMS usage: socket_name
type: vector byte (unsigned)
access: read only
mechanism: by `item_list_3` descriptor

The port number and internet address of the local name associated with the socket. The **p3** argument is the address of an `item_list_3` descriptor that points to the socket address structure into which the local name is written.

Use the IO$_SENSEMODE function with the IO$M_EXTEND modifier to specify a BSD Version 4.4 formatted socket address structure.

Related Functions

The Sockets API equivalent for this function is `getsockname( )`.

p4

OpenVMS usage: socket_name
type: vector byte (unsigned)
access: read only
mechanism: by `item_list_3` descriptor

The port number and internet address of the remote name associated with the socket's peer. The **p3** argument is the address of an `item_list_3` descriptor that points to the socket address structure into which the peer name is written.

Use the IO$_SENSEMODE function with the IO$M_EXTEND modifier to specify a BSD Version 4.4 formatted socket address structure.

Related Functions

The equivalent Sockets API function is `getpeername( )`.

p6

OpenVMS usage: output_parameter_list
type: vector byte (unsigned)
access: read only
mechanism: by `item_list_2` descriptor

Output parameter list describing one or more parameters to return. The **p6** argument is the address of an item_list_2 descriptor that points to and identifies the type of output parameter list.

The following are the types of output parameter lists:

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

Each item_list_3 structure appearing in an output parameter list describes an individual parameter or item to return. See Table A–1 for details about socket option parameters; see Table A–2 for TCP protocol option parameters; and see Table A–3 for IP protocol option parameters. Unsupported socket or protocol options are ignored.

Each ioctl_com structure that appears in an output parameter list contains an I/O control command — the get IOCTL request code and its associated IOCTL structure address. See Table B–1 for details about IOCTL command parameters.

Unsupported socket options are ignored.

The equivalent Sockets API functions are getsockopt( ) and ioctl( ).

Function Modifiers

IO$M_EXTEND	Specifies the format of the socket address structure to return when used with the **p3** or **p4** arguments. When specified, a BSD Version 4.4 formatted socket address structure is returned.

Condition Values Returned

SS$_NORMAL	The service completed successfully.
SS$_ACCVIO	The service cannot access a buffer specified by one or more arguments.
SS$_BADPARAM	Programming error occurred for one of the following reasons: • $QIO system service was specified without a socket. • Error occurred processing a socket or protocol option.
SS$_DEVINTACT	The network driver was not started.
SS$_DEVNOTMOUNT	The network driver is loaded, but the INETACP is not currently available for use.

SS$_ILLCNTRFUNC	Programming error. The operation is unsupported for one of the following reasons: • An invalid IO$_SENSEMODE function for the interface was specified. The interface does not have an IOCTL routine. • An IO$_SENSEMODE function that requires a socket was specified, but the device did not have one. Create a socket and then issue the function. • An unsupported operation was performed on at least one of the following protocols: raw IP, datagram, or stream sockets.
SS$_INSFMEM	Insufficient system dynamic memory to complete the service.
SS$_IVBUFLEN	The size of a socket option buffer specified with the IO$_SENSEMODE function was invalid.
SS$_NOSUCHDEV	Programming error or INET management error. An INET address is not in the Address Resolution Protocol (ARP) table. An attempt to show or delete an ARP table entry failed.
SS$_NOLINKS	The specified socket was not connected.
SS$_NOOPER	Programming error. An attempt to get ARP information occurred without OPER privilege.
SS$_PROTOCOL	A network protocol error occurred. The address family specified in the socket address structure is not supported.
SS$_SHUT	The local or remote node is no longer accepting connections.
SS$_UNREACHABLE	The remote node is currently unreachable.

IO$_SETMODE/IO$_SETCHAR

The IO$_SETMODE and IO$_SETCHAR functions set one or more parameters (characteristics) pertaining to the network driver.

Sockets are created using the IO$_SETMODE **p1** argument. Names are assigned to sockets using the IO$_SETMODE **p3** argument. Active sockets are converted to passive sockets using the IO$_SETMODE **p4** argument. Other parameters, such as socket and protocol options, are specified in an input parameter list using the IO$_SETMODE **p5** argument.

The IO$_SETMODE **p1**, **p3**, and **p4** arguments can be used with the **p5** argument in a single $QIO system service to set socket names as well as socket and protocol options. IO$_SETMODE processes arguments in this order: **p1**, **p3**, **p4**, **p5**. If IO$_SETMODE detects an error, the IOSB contains the error and argument address or the value that was at fault.

Refer to individual argument descriptions for details about specifying the type and format of input parameters.

Arguments

p1

OpenVMS usage:	socket_characteristics
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Longword specifying the protocol, socket type, and address family, of a new socket. The **p1** argument is the address of the longword containing the socket characteristics.

The newly created socket is marked privileged if the image that creates a socket runs in a process that has a privileged UIC or has BYPASS, OPER, or SYSPRV privilege.

The following table shows protocol codes:

Protocol	Description
TCPIP$C_TCP	TCP/IP protocol
TCPIP$C_UDP	UDP/IP protocol
TCPIP$C_RAW_IP	IP protocol

Table 6–6 lists the valid socket types.

Table 6–6 Socket Types

Socket Type	Description
TCPIP$C_STREAM	Permits bidirectional, reliable, sequenced, and unduplicated data flow without record boundaries.

(continued on next page)

Table 6–6 (Cont.) Socket Types

Socket Type	Description
TCPIP$C_DGRAM	Permits bidirectional data flow with record boundaries. No provisions for sequencing, reliability, or unduplicated messages.
TCPIP$C_RAW	Permits access to the IP layer; used to develop new protocols that are layered upon the IP layer.

The following table shows address family codes:

Address Family	Description
TCPIP$C_AF_INET	Internet domain (default).
TCPIP$C_AUXS	Accept hand-off of a socket already created and initialized by the auxiliary server.

Related Functions

The equivalent Sockets API function is `socket( )`.

p3

OpenVMS usage: socket_name
type: vector byte (unsigned)
access: read only
mechanism: by `item_list_2` descriptor

The local name (that is, port number and internet address) to assign to the socket. The **p3** argument is the address of an `item_list_2` descriptor that points to the socket address structure containing the local name.

Related Functions

The equivalent Sockets API function is `bind( )` .

p4

OpenVMS usage: connection_backlog
type: byte (unsigned)
access: read only
mechanism: by value

Maximum limit of outstanding connection requests for a socket that is connection oriented. If more connection requests are received than are specified, the additional requests are ignored so that TCP retries can succeed.

Related Functions

The equivalent Sockets API function is `listen( )`.

p5

OpenVMS usage: input_parameter_list
type: vector byte (unsigned)
access: read only
mechanism: by `item_list_2` descriptor

Input parameter list describing one or more parameters to set. The **p5** argument is the address of an `item_list_2` descriptor that points to and identifies the type of input parameter list.

The following are the types of input parameter lists:

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

Each `item_list_2` structure appearing in an input parameter list describes an individual parameter or item to set. See Table A–1 for details about socket option parameters; see Table A–2 for TCP protocol option parameters; and see Table A–3 for details about IP protocol option parameters. Unsupported socket or protocol options are ignored.

Each `ioctl_com` structure that appears in an input parameter list contains an I/O control command — the set IOCTL request code and its associated IOCTL structure address. See Table B–1 for details about IOCTL command parameters.

You can use one $QIO system call to set up several socket options at once.

Unsupported socket options are ignored.

To execute set IOCTL operations, you need a system UIC or SYSPRV, BYPASS, or OPER privilege.

Related Functions

The equivalent Sockets API functions are `setsockopt( )` and `ioctl( )`.

Condition Values Returned

SS$_NORMAL	The service completed successfully.
SS$_ACCVIO	The service cannot access a buffer specified by one or more arguments.
SS$_BADPARAM	Programming error that occurred for one of the following reasons: • $QIO system service was specified without a socket. • Error occurred processing a socket or protocol option.
SS$_DEVINTACT	The network driver was not started.
SS$_DEVNOTMOUNT	The network driver is loaded, but the INETACP is not currently available for use.
SS$_DUPLNAM	Programming error. The port being bound is already in use. An attempt to bind the socket to an address and port failed.

SS$_EXQUOTA	Programming or INET management error occurred for one of the following reasons: • An attempt to create a new socket with the IO$_SETMODE function occurred, but the maximum number of sockets was exceeded. Increase the maximum number of sockets (INET parameter). • The number of sockets specified with the IO$_SETMODE (listen) exceeds the maximum number of sockets. Increase the maximum number of sockets (INET parameter), or reduce the *listen* parameter (the number of sockets the listener socket can create).
SS$_FILALRACC	Programming error. The INET address is already in use. An attempt to bind the socket to an address and port failed.
SS$_ILLCNTRFUNC	Programming error. The operation is not supported for one of the following reasons: • An invalid IO$_SETMODE function for the interface occurred that does not have an IOCTL routine. • An attempt to perform an IO$_SETMODE function required a socket, but the device did not have one. Create a socket before issuing the function.
SS$_IVADDR	Programming error. The INET address you specified using the IO$_SETMODE function was not placed into the system. This resulted in an invalid port number or INET address combination. The INET address was invalid for one of the following reasons: • Port zero and INET address zero are not allowed, or port zero is not allowed when using an IO$_ACCESS or IO$_WRITEVBLK function. • An attempt to exceed the limit of allowable permanent entries in the ARP table occurred. • An attempt to bind a raw IP socket when there are no interfaces defined in the system occurred. • An attempt to bind a raw IP socket to a null INET address occurred.
SS$_INSFMEM	Insufficient system dynamic memory to complete the service.

SS$_IVBUFLEN	The size of a socket option buffer specified with the IO$_SETMODE function was invalid.
SS$_NOLICENSE	Programming or system management error. A TCP/IP Services license is not present.
SS$_NOOPER	Programming or INET management error. An attempt to execute an I/O function that needs the OPER privilege occurred.
SS$_NOPRIV	Programming or INET management error. There are not enough privileges for the attempted operation for one of the following reasons: • An attempt to broadcast an IP datagram on a process without a system UIC or SYSPRV, BYPASS, or OPER privilege occurred. • An attempt to use a reserved port number lower than 1024 occurred. • An attempt to access a process that requires a system UIC or SYSPRV, or BYPASS privilege occurred. • An attempt to use raw IP on a privileged socket that requires the SYSPRV or BYPASS privilege occurred.
SS$_NOSUCHDEV	Programming error or INET management error. An attempt to show or delete an ARP table entry failed because the INET address is not found.
SS$_NOSUCHNODE	Programming error or INET management error. An attempt to delete a route from the routing table failed because the entry was not found.

SS$_PROTOCOL	Programming error. A specified protocol or address family caused an error for one of the following reasons: • An invalid protocol type was specified at socket creation. • An unsupported protocol was specified. • A protocol type that was not found in the internal tables was specified. • The address family is unsupported for one of the following reasons: – An unsupported address family with the IO$_SETMODE subfunction was specified. Instead, specify the TCPIP$C_AF_INET or TCPIP$C_UNSPEC address family. – An unsupported address family for a remote INET address with the IO$_ACCESS or IO$_WRITEVBLK function was specified. Instead, specify the TCPIP$C_AF_INET address family. – An unsupported address family for the local INET address with the IO$_SETMODE function was specified. Instead, specify the TCPIP$C_AF_INET address family. – An unsupported address family for the INET address of the routing module was specified. Instead, specify the TCPIP$C_AF_INET address family.
SS$_SHUT	The local or remote node is no longer accepting connections.

IO$_SETMODE | IO$M_OUTBAND

The IO$_SETMODE | IO$M_OUTBAND function/modifier combination requests that the asynchronous system trap (AST) for an out-of-band (OOB) character be delivered to the requesting process. This is to be done only when an OOB character is received on the socket and there is no waiting read request. The socket must be a TCP (stream) socket.

The Enable OOB character AST function allows an Attention AST to be delivered to the requesting process only once. After the AST occurs, the function must explicitly reenable AST delivery before a new AST can be delivered. This function is subject to AST quotas.

Arguments

p1

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

To enable the AST, the **p1** argument is the address of the OOB character AST routine. To disable the AST, **p1** equals 0.

p2

OpenVMS usage:	user_arg
type:	longword (unsigned)
access:	read only
mechanism:	by value

AST parameter to be delivered to the AST routine specified by the **p1** argument.

p3

OpenVMS usage:	access_mode
type:	longword (unsigned)
access:	read only
mechanism:	by value

Access mode to deliver the AST.

Condition Values Returned

SS$_NORMAL	The service completed successfully.
SS$_ABORT	Programming, INET management, or hardware error.
SS$_ACCVIO	Programming error. An attempt to access an invalid memory location or buffer occurred.

SS$_BADPARAM	Programming error. A $QIO service with an invalid parameter occurred for one of the following reasons: • An attempt to execute an IO$_SETMODE function (all subfunctions, except socket creation) without specifying a device socket. Instead, create a device socket by issuing a $QIO with the IO$_SETMODE function and the proper parameters. • A socket option was specified incorrectly.
SS$_DEVACTIVE	INET management error. An attempt to change the static INET parameters occurred. If new parameters are needed, shut down the internet, reset the static parameters, and issue the START COMMUNICATION command.
SS$_DEVINTACT	INET management error. The driver was not started. Issue a START COMMUNICATION command before issuing $QIO functions.
SS$_DEVNOTMOUNT	INET management error. The INET startup procedure executed incorrectly. The driver was loaded, but the INET_ACP was not activated. Execute the INET startup procedure again.
SS$_DUPLNAM	Programming error. An attempt to bind a port that is already in use occurred. An attempt to bind the socket to an address and port failed.
SS$_EXQUOTA	Programming or INET management error occurred because of one of the following reasons: • An attempt to create a new socket with the IO$_SETMODE function failed because the maximum number of sockets was exceeded. Increase the maximum number of sockets (INET parameter). • The number of sockets specified with the IO$_SETMODE (listen) exceeds the maximum number of sockets. Increase the maximum number of sockets (INET parameter), or reduce the *listen* parameter (the number of sockets that the listener socket can create).
SS$_FILALRACC	Programming error. INET address is already in use. An attempt to bind the socket to an address and port failed.
SS$_INSFMEM	Programming or system management error: Not enough resources to allocate new socket.

SS$_ILLCNTRFUNC	Programming error. Operation is not supported because of one of the following reasons: • Invalid IO$_SETMODE (IOCTL) function was used for the interface. The interface does not have an IOCTL routine. • An attempt to perform an IO$_SETMODE (IOCTL) function that required a socket, but the device did not have one. Create a socket and issue the IOCTL function.
SS$_IVADDR	Programming error. The specified INET address is not in the system, and an invalid port number or an invalid INET address combination was specified with an IO$_SETMODE function (a bind) for one of the following reasons: • An attempt to bind the address failed because the INET address is not in the system and port zero and INET address zero are not allowed. • An attempt to make a permanent entry in an ARP table that was full failed. • An attempt was made to bind an IP socket (raw IP) when there are no interfaces defined in the system. • An attempt was made to bind an IP socket (raw IP) to a null INET address.
SS$_IVBUFLEN	Programming error. The socket option buffer has an invalid size.
SS$_NOLICENSE	Programming or system management error. TCP/IP Services not present.
SS$_NOOPER	Programming or INET management error. An attempt was made to execute an I/O function that needs the OPER privilege.

SS$_NOPRIV	Programming or INET management error. Not enough privileges for the attempted operation for one of the following reasons: • Broadcasting an IP datagram was denied because the process does not have a system UIC or SYSPRV, BYPASS, or OPER privilege. • An attempt was made to use a reserved port number lower than 1024. • An operation accesses only processes that have a system UIC or SYSPRV or BYPASS privilege. • Raw IP protocol can be used only on privileged sockets. The process must have a SYSPRV or BYPASS privilege.
SS$_NOSUCHDEV	Programming error or INET management error. An INET address is not in the ARP table. An attempt to show or delete an ARP table entry failed.
SS$_NOSUCHNODE	Programming or INET management error. An attempt to delete a route from the routing table failed because a route entry was not found.

SS$_PROTOCOL	Programming error because of one of the following reasons: • The protocol type specified at socket creation is not valid. • The protocol is not supported. • The protocol type specified is not found in the internal tables and therefore is an invalid type. • The address family is not supported for one of the following reasons: – The address family specified with an IO$_SETMODE function (IOCTL subfunction) is not supported. The address family should be the TCPIP$C_AF_INET or TCPIP$C_UNSPEC address family. – The address family of the local INET address specified with an IO$_SETMODE (bind) function is not supported. The address family should be the TCPIP$C_AF_INET address family. – The address family of the INET address specified in a request to the routing module is not supported. The address family should be the TCPIP$C_AF_INET address family.
SS$_SHUT	The local or remote node is no longer accepting connections.

IO$_SETMODE | IO$M_READATTN

The IO$_SETMODE | IO$M_READATTN function/modifier combination requests that an Attention AST be delivered to the requesting process when a data packet is received on the socket and there is no waiting read request.

The Enable Read Attention AST function enables an Attention AST to be delivered to the requesting process only once. After the AST occurs, the function must explicitly reenable AST delivery before the AST can occur again. The function is subject to AST quotas.

Consider the following when using IO$M_READATTN:

- There is a one-to-one correspondence between the number of times you enable an Attention AST and the number of times the AST is delivered. For example, for each enabled AST, one AST is delivered. If you enable an Attention AST several times, several ASTs are delivered for one event when an event occurs.
- If an out-of-band (OOB) Attention AST is enabled, the OOB AST is delivered, regardless of the following:
  - An enabled Read Attention AST
  - The TCPIP$C_OOBINLINE socket option
  - A READ $QIO waiting for completion on the socket

If the TCPIP$C_OOBINLINE option is set, then a waiting READ $QIO is completed and the OOB character is returned in the data stream.

- If both an OOB AST and a Read Attention AST are enabled, only the OOB AST is delivered when an OOB character is received.
- If a Read Attention AST is enabled and the TCPIP$C_OOBINLINE socket option is set, a waiting READ $QIO completes and the OOB character is returned in the data stream.
- If a Read Attention AST is enabled and the TCPIP$C_OOBINLINE socket option is not set (clear), the Read Attention AST is delivered when an OOB character is received, regardless of whether a READ $QIO is waiting for completion. In this case, the OOB character is not returned in the data stream. Therefore, if the OOB character is the only character received, the READ $QIO does not complete.

Arguments

p1

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

To enable the AST, the **p1** argument is the address of the Read Attention AST routine. To disable the AST, set **p1** to 0.

p2

OpenVMS usage:	user_arg
type:	longword (unsigned)
access:	read only
mechanism:	by value

AST parameter to be delivered to the AST routine.

p3

OpenVMS usage:	access_mode
type:	longword (unsigned)
access:	read only
mechanism:	by value

Access mode to deliver the AST.

Condition Values Returned

SS$_ABORT	Programming, INET management, or hardware error. The route entry already exists, so the attempt to add a route entry using the IO$_SETMODE function failed.
SS$_ACCVIO	Programming error. An attempt to access an invalid memory location or buffer occurred.
SS$_BADPARAM	Programming error. The parameter specified for a $QIO function was invalid for one of the following reasons: • An attempt to execute the IO$_SETMODE subfunctions without specifying a device socket occurred. Instead, create a device socket by issuing a $QIO with the IO$_ SETMODE function and the proper parameters. • A socket option was specified incorrectly.
SS$_DEVACTIVE	INET management error. An attempt to change the static INET parameter was unsuccessful. If you need new parameters, shut down the internet, reset the static parameters, and issue the START COMMUNICATION command.
SS$_DEVINTACT	INET management error. The driver was not started. Issue a START COMMUNICATION command before issuing $QIO functions.
SS$_DEVNOTMOUNT	INET management error. TCP/IP Services improperly executed the startup procedure. The driver was loaded, but the INET_ACP was not activated. Execute the INET startup procedure again.
SS$_DUPLNAM	Programming error. An attempt to bind a port already in use occurred so the operation to bind the socket to the address and port failed.

SS$_EXQUOTA	Programming or INET management error. The quota for the valid number of sockets caused an error for one of the following reasons: • An attempt to exceed the maximum number of sockets by creating new socket with the IO$_SETMODE function occurred. Increase the maximum number of allowable sockets (INET parameter) before creating more sockets. • The number of sockets specified with the IO$_SETMODE function exceeds the maximum number of sockets allowed. Increase the maximum number of sockets (INET parameter) or reduce the number of sockets that the listener socket can create (*listen* parameter).
SS$_FILALRACC	Programming error. An attempt to bind the socket to an address that is already in use occurred and the operation failed.
SS$_INSFMEM	Programming or system management error. The system does not have enough resources to allocate new socket.
SS$_ILLCNTRFUNC	Programming error. Operation is not supported. • Invalid IO$_SETMODE (IOCTL) function was used for the interface. The interface does not have an IOCTL routine. • An attempt was made to perform an IO$_ SETMODE (IOCTL) function that required a socket, but the device did not have one. Create a socket and issue the IOCTL function.

SS$_IVADDR	Programming error. The specified INET address is not in the system, and an invalid port number or an invalid INET address combination was specified with an IO$_SETMODE function (a bind). • An attempt to bind the address failed because the INET address is not in the system, port zero and INET address zero are not allowed, or port zero is not allowed when using an IO$_ACCESS or IO$_WRITEVBLK function. • An attempt to make a permanent entry in the ARP table failed because of lack of space. Too many permanent entries. • An attempt was made to bind an IP socket (raw IP) when there are no interfaces defined in the system. • An attempt was made to bind an IP socket (raw IP) to a null INET address.
SS$_IVBUFLEN	Programming error. The socket option buffer has an invalid size.
SS$_NOLICENSE	Programming or system management error. TCP/IP Services not present.
SS$_NOOPER	Programming or INET management error. An attempt was made to execute an I/O function that needs the OPER privilege.
SS$_NOPRIV	Programming or INET management error. Not enough privileges for the attempted operation. • Broadcasting an IP datagram was denied because the process does not have a system UIC or SYSPRV, BYPASS, or OPER privilege. • An attempt was made to use a reserved port number lower than 1024. • An operation accesses only processes that have a system UIC or SYSPRV, or BYPASS privilege. • Raw IP protocol can be used only on privileged sockets. The process must have a SYSPRV or BYPASS privilege.
SS$_NOSUCHDEV	Programming error or INET management error. An INET address is not in the ARP table. An attempt to show or delete an ARP table entry failed.
SS$_NOSUCHNODE	Programming error or INET management error. An attempt to delete a route from the routing table failed because a route entry was not found.

SS$_PROTOCOL	Programming error. • The protocol type specified at socket creation is not valid. • The protocol is not supported. • The protocol type specified is not found in the internal tables. It is an invalid type. • The address family is not supported: – The address family specified with an IO$_SETMODE function (IOCTL subfunction) is not supported. The address family should be the TCPIP$C_AF_INET or TCPIP$C_UNSPEC address family. – The address family of the remote INET address specified with an IO$_ACCESS or IO$_WRITEVBLK function is not supported (UDP/IP or TCP/IP). The address family should be the TCPIP$C_AF_INET address family. – The address family of the local INET address specified with an IO$_SETMODE (bind) function is not supported. The address family should be the TCPIP$C_AF_INET address family. – The address family of the INET address that is specified in a request to the routing module is not supported. The address family should be the TCPIP$C_AF_INET address family.
SS$_SHUT	The local or remote node is no longer accepting connections.

IO$_SETMODE | IO$M_WRTATTN

The IO$_SETMODE | IO$M_WRTATTN function/modifier combination (IO$M_WRTATTN is Enable Write Attention AST) requests that an Attention AST be delivered to the requesting process when a data packet can be queued to the socket. For TCP sockets, this occurs when space becomes available in the TCP transmit queue.

The Enable Write Attention AST function enables an Attention AST to be delivered to the requesting process only once. After the AST occurs, the function must explicitly reenable AST delivery before the AST can occur again. The function is subject to AST quotas.

There is a one-to-one correspondence between the number of times you enable an Attention AST and the number of times the AST is delivered. For example, for each enabled AST, one AST is delivered. If you enable an Attention AST several times, several ASTs are delivered for one event when the event occurs.

You can use the management command SHOW DEVICE_SOCKET to display information about the socket's characteristics, options, and state.

Arguments

p1

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

To enable the AST, the **p1** argument is the address of the Write Attention AST routine. To disable the AST, **p1** is set to 0.

p2

OpenVMS usage:	user_arg
type:	longword (unsigned)
access:	read only
mechanism:	by value

AST parameter to be delivered to the AST routine.

p3

OpenVMS usage:	access_mode
type:	longword (unsigned)
access:	read only
mechanism:	by value

Access mode to deliver the AST.

Condition Values Returned

SS$_ABORT	Programming error, INET management error, or hardware error. The route specified with the IO$_SETMODE function already exists. Therefore, the operation failed.

SS$_ACCVIO	Programming error. An attempt to access an invalid memory location or buffer occurred.
SS$_BADPARAM	Programming error. The parameter specified for the $QIO I/O function was invalid for one of the following reasons: • An attempt to execute the IO$_SETMODE functions without specifying a device socket occurred. Instead, create a device socket by issuing a $QIO with the IO$_SETMODE function and the proper parameters. • A socket option was specified incorrectly.
SS$_DEVACTIVE	INET management error. You attempted to change the static INET parameters. If you need new parameters, shut down the internet, reset the static parameters, and issue the START COMMUNICATION command.
SS$_DEVINTACT	INET management error. The driver is not started. Issue a START COMMUNICATION command before issuing $QIO functions.
SS$_DEVNOTMOUNT	INET management error. The INET startup procedure was improperly executed. The driver was loaded, but the INET_ACP was not activated. Execute the INET startup procedure again.
SS$_DUPLNAM	Programming error. Port that is being bound is already in use. An attempt to bind the socket to an address and port failed.
SS$_EXQUOTA	Programming or INET management error. • An attempt to create a new socket with the IO$_SETMODE function and it failed because the maximum number of sockets was exceeded. Increase the maximum number of sockets (INET parameter), and then create a new socket. • The number of sockets specified with the IO$_SETMODE function exceeds the allowable maximum number of sockets. Increase the maximum number of sockets (INET parameter), or reduce the number of sockets that the listener socket can create (*listen* parameter).
SS$_FILALRACC	Programming error. Because the INET address is already in use, an attempt to bind the socket to an address and port failed.
SS$_INSFMEM	Programming or system management error. There are not enough resources to allocate a new socket.

SS$_ILLCNTRFUNC	Programming error. The operation is unsupported for one of the following reasons: • An invalid IO$_SETMODE function for the interface was specified. The interface does not have an IOCTL routine. • An attempt to execute an IO$_SETMODE function that required a socket, but the device did not have one. Instead, create a socket and issue the function.
SS$_IVADDR	Programming error. An invalid port number and INET address combination was specified with the IO$_SETMODE bind function. This caused the operation to fail for one of the following reasons: • An illegal combination of port zero and INET address zero was specified. • An attempt to make a permanent entry in the ARP table occurred, and the operation failed because of lack of space. There are too many permanent entries. • An attempt to bind a raw IP socket occurred when there were no interfaces defined in the system. • An attempt to bind a raw IP socket to a null INET address occurred.
SS$_IVBUFLEN	Programming error. An invalid size was specified for the socket option buffer.
SS$_NOLICENSE	Programming or system management error. There is no TCP/IP Services license present.
SS$_NOOPER	Programming or INET management error. An attempt to execute an I/O function that needs the OPER privilege occurred.
SS$_NOPRIV	Programming or INET management error. The operation failed for one of the following reasons: • An attempt to broadcast an IP datagram for a process without having a system UIC or SYSPRV, BYPASS, or OPER privilege. • An attempt to use a reserved port number lower than 1024 occurred. • An attempt to access a process without having a system UIC or SYSPRV, or BYPASS privilege occurred. • An attempt to use raw IP on a socket that is not a privileged socket occurred. To do this, the process must have SYSPRV or BYPASS privilege.

SS$_NOSUCHDEV	Programming error or INET management error. An attempt to show or delete an entry in the ARP table occurred. However, because the INET address was not in the ARP table, the operation failed.
SS$_NOSUCHNODE	Programming error or INET management error. An attempt to delete a route from the routing information table (RIT) occurred. However, because the route was not found in the RIT, the operation failed.
SS$_PROTOCOL	Programming error. • An invalid protocol type was specified when creating a socket. • An unsupported protocol was specified. • An unsupported protocol type was specified because it is not found in the internal tables. • An unsupported address family was specified for one of the following reasons:

	– An invalid address family was specified with an IO$_SETMODE subfunction. Instead, specify the TCPIP$C_ AF_INET or TCPIP$C_ UNSPEC address family. – An address family of the remote INET address for a datagram or stream socket was specified with an IO$_ACCESS or IO$_WRITEVBLK function. Instead, specify the TCPIP$C_AF_INET address family. – An invalid address family of the local INET address was specified with an IO$_SETMODE bind function. Instead, specify the TCPIP$C_AF_INET address family. – You made a request to the routing module by specifying the address family of the INET address. Instead, specify the TCPIP$C_AF_INET address family.
SS$_SHUT	The local or remote node is no longer accepting connections.

IO$_WRITEVBLK

The IO$_WRITEVBLK function transmits data from the specified user buffers to an internet host. Use both **p1** and **p2** arguments to specify a single user buffer. Use the **p5** argument to specify multiple buffers.

For connection-oriented protocols, such as TCP, if the socket transmit buffer is full, the IO$_WRITEVBLK function is blocked until the socket transmit buffer has room for the user data.

For connectless-oriented protocols, such as UDP and raw IP, the user data is transmitted in one datagram. If the user data is greater than the socket's transmit quota, the error code (SS$_TOOMUCHDATA) is returned.

Related Functions

The equivalent Sockets API functions are `send( )`, `sendto( )`, `sendmsg( )`, and `write( )`.

Arguments

p1

OpenVMS usage: buffer
type: vector byte (unsigned)
access: read only
mechanism: by 32- or 64-bit reference (Alpha)
by 32-bit reference (VAX)

The 32- or 64-bit address (on Alpha systems) or the 32-bit address (on VAX systems) of the buffer containing the data to be transmitted. The length of this buffer is specified by the **p2** argument.

p2

OpenVMS usage: buffer_length
type: quadword unsigned (Alpha); longword unsigned (VAX)
access: read only
mechanism: by 64-bit value (Alpha)
by 32-bit value (VAX)

The length (in bytes) of the buffer containing data to be transmitted. The address of this buffer is specified by the **p1** argument.

p3

OpenVMS usage: socket_name
type: vector byte (unsigned)
access: read only
mechanism: by `item_list_2` descriptor

The remote port number and internet address of the message destination. The **p3** argument is the address of an `item_list_2` descriptor pointing to the socket address structure containing the remote port number and internet address.

p4

OpenVMS usage: mask_longword
type: longword (unsigned)
access: read only
mechanism: by value

Longword of flags to specify attributes for this write operation. The following table lists the available write flags:

Write Flag	Description
TCPIP$C_MSG_OOB	Writes an out-of-band (OOB) byte.
TCPIP$C_MSG_DONTROUTE	Sends message directly without routing.
TCPIP$C_MSG_NBIO	Completes the I/O operation and returns an error if a condition arises that would cause the I/O operation to be blocked. (Similar to using IO$M_NOWAIT.)

p5

OpenVMS usage:	buffer_list
type:	vector byte (unsigned)
access:	read only
mechanism:	by 32- or 64-bit descriptor-fixed-length descriptor (Alpha) by 32-bit descriptor-fixed-length descriptor (VAX)

Input buffer list describing one or more buffers containing the data to be transmitted. The **p5** argument is the 32- or 64-bit address (on Alpha systems) or the 32-bit address (on VAX systems) of a descriptor pointing to a input buffer list. Buffers are transmitted in the order specified by the input buffer list. The transfer-length value returned in the I/O status block is the total number of bytes transferred from all buffers.

If you use the **p1** and **p2** arguments, do not use the **p5** argument; they are mutually exclusive.

Function Modifiers

IO$M_EXTEND	Allows the use of extended modifiers with BSD Version 4.4. Valid only for datagram sockets (UDP or raw IP); ignored for TCP.
IO$M_INTERRUPT	Sends an OOB message.
IO$M_NOWAIT	Regardless of a $QIO or $QIOW, if the system detects a condition that would cause the operation to block, the system completes the I/O operation and returns the SS$_SUSPENDED status code. When using this function modified, always check the message length in the IOSB to ensure that all data is transferred. IO$_WRITEVBLK returns a success status even if data is only partially transferred.

Condition Values Returned

SS$_ABORT	Programming error, INET management error, or hardware error. The execution of the I/O was aborted.
SS$_ACCVIO	Programming error. An attempt was made to access an invalid memory location or buffer.
SS$_BADPARAM	Programming error. A $QIO I/O function was specified using an invalid parameter. • An attempt was made to execute an IO$_WRITEVBLK function without specifying a device socket. First create a device socket by issuing an IO$_SETMODE function and the proper arguments. • An attempt was made to issue an IO$_WRITEVBLK function that did not specify a correct buffer address (**p1** or **p5** is null). • An attempt was made to issue an IO$_ WRITEVBLK that specifies an invalid vectored buffer (**p5** specifies an invalid address descriptor).
SS$_CANCEL	The I/O operation was canceled by the $CANCEL system service.
SS$_DEVINTACT	The network driver was not started.
SS$_DEVNOTMOUNT	The network driver is loaded, but the INETACP in not currently available for use.
SS$_EXQUOTA	Returned when process resource mode wait is disabled. There is no internet request packet (IRP) available for completing the request. Increase the buffered I/O quota.
SS$_FILALRACC	Programming error. • INET address is already in use. An attempt was made to bind the socket to an address but the port failed. • IP protocol (raw socket). An attempt was made to specify a remote INET socket address with an IO$_WRITEVBLK function, while an INET address was already specified with an IO$_ACCESS function. • UDP/IP protocol. An attempt was made to specify a remote INET socket address with an IO$_WRITEVBLK function, while an INET address was already specified with the IO$_ACCESS function.

SS$_ILLCNTRFUNC	Programming error. Unsupported operation on the protocol (IP, UDP/IP, TCP/IP).
SS$_INSFMEM	INET management or programming error returned when process resource mode wait is disabled. Not enough system space for buffering user data. A higher quota for socket buffer space needs to be set, or the internet needs more dynamic buffer space (number of dynamic clusters should be increased).
SS$_IVADDR	Programming error. The specified INET address is not in the system, and an invalid port number or an INET address combination was specified with an IO$_WRITEVBLK operation. • An attempt to bind the socket failed because the INET address is not in the system, port number zero and INET address zero are not allowed, or port zero is not allowed with an IO$_ACCESS or IO$_WRITEVBLK function. • An attempt to get an interface INET address, broadcast mask, or network mask failed. • A send request was made on a datagram-oriented protocol, but the destination address is unknown or not specified.
SS$_IVBUFLEN	Programming error. • The size of the buffer for an I/O function is insufficient. • An attempt was made to issue an IO$_WRITEVBLK function that specifies a correct buffer address (**p1** valid) but does not specify a buffer length (**p2** is null).
SS$_LINKDISCON	Notification. Connection completion return code. The virtual circuit (TCP/IP) was closed at the initiative of the peer. The application must stop sending data and must either shut down or close the socket.
SS$_PROTOCOL	Programming error. The address family of the remote INET address specified with an IO$_WRITEVBLK function is not supported (UDP/IP or TCP/IP). The address family should be the TCPIP$C_AF_INET address family.

SS$_NOLINKS	Programming error. The socket was not connected (TCP/IP), or an INET port and address were not specified with an IO$_ACCESS (UDP/IP or IP). • An IO$_WRITEVBLK with no remote INET socket address was issued on a socket that was not the object of an IO$_ACCESS function (raw IP). • An IO$_WRITEVBLK with no remote INET socket address was issued on a socket that was not the object of an IO$_ACCESS function (UDP/IP). • An attempt was made to disconnect a socket that is not connected, or an attempt was made to issue an IO$_WRITEVBLK function on an unconnected socket (TCP/IP).
SS$_SHUT	The local or remote node is no longer accepting connections.
SS$_SUSPENDED	The system detected a condition that might cause the operation to block.
SS$_TIMEOUT	Programming error, INET management error, or hardware error. • A TCP/IP connection timed out after several unsuccessful retransmissions. • On a TCP socket where KEEPALIVE is set, the connection was idle for longer than the timeout interval (The default is 10 minutes).
SS$_TOOMUCHDATA	Programming or INET management error. The message size was too large. • An IP packet that is broadcast cannot be fragmented. • The Not Fragment IP flag was set and the IP datagram was too large to be sent without being fragmented. • Internal error. The length of the Ethernet datagram does not allow enough space for the minimum IP header. • The message to be sent on a UDP/IP or raw IP socket is larger than the socket buffer high water allows. • An attempt was made to send or receive more than 16 buffers specified with the **p5** argument.

SS$_UNREACHABLE	Communication status. The remote host is currently unreachable. Hardware error. The data link adapter detected an error and shut itself off. The TCP/IP Services software is waiting for the adapter to come back on line.

6.3 TELNET Port Driver I/O Function Codes

The TELNET port driver (TNDRIVER) provides terminal session support for TCP stream connections using the RAW, NVT, RLOGIN, and TELNET protocols. Either a remote device or an application can be present at the remote endpoint of the connection.

A user program can manage a TELNET connection with the standard OpenVMS $QIO system service by using the IO$_TTY_PORT and IO$_TTY_PORT_BUFIO I/O function codes. This section describes these I/O function codes and their associated arguments.

6.3.1 Interface Definition

The following definitions are used by the interface. The symbols are defined in SYS$LIBRARY:TNIODEF.H.

6.3.1.1 Item List Codes

Table 6–7 describes the symbols used with the **p5** parameter.

Table 6–7 List Codes for the p5 Item

Item Code	Maximum Size	Description
TN$_ACCPORNAM	64	Access port name string. When written, the string's length is determined by the item_length field. The value of `item_length` should not be more than 63 bytes. When read, the string is returned in ASCIC format (the first byte contains the string's length), so a size of 64 is appropriate.
TN$_CHARACTERISTICS	4	Characteristics mask. This longword contains a bit mask of the device's characteristics read or to be written. (See Table 6–8.)
TN$_CONNECTION_ATTEMPTS	4	Reconnection attempts. This item is the number of unsuccessful reconnection attempts which have been made on a reconnectable device. The value will be reinitialized when a successful connection is made. This item is read only.
TN$_CONNECTION_INTERVAL	4	Minimum time (in seconds) before reconnection attempts.
TN$_CONNECTION_TIMEOUT	4	Current time (in seconds) since the last reconnection attempt. This item is read only.
TN$_DATA_HIGH	4	Maximum amount of output data (in bytes) buffered at the network port. This number does not affect the amount of data buffered within the socket.

(continued on next page)

Table 6–7 (Cont.) List Codes for the p5 Item

Item Code	Maximum Size	Description
TN$_DEVICE_UNIT	4	Terminal device unit number. When written, this value must be between 1 and 9999.
TN$_IDLE_INTERVAL	4	Maximum idle time (in seconds) allowed before a connection is to be broken. Connections are not broken if the device is stalled.
TN$_IDLE_TIMEOUT	4	Current time (in seconds) since last output on the terminal. This item is read only.
TN$_LOCAL_ADDRESS	32	Local `sockaddr` of the active connection. When written, the value of `item_length` determines the size of the `sockaddr`. Note that the `sockaddr` is in BSD Version 4.4 format, which includes a `sockaddr` size field. (C programs should be compiled with the _SOCKADDR_LEN symbol defined.) This item is read only.
TN$_NETWORK_DEVICE_NAME	64	Name of the network pseudodevice currently bound to the terminal. When read, the data is returned in ASCIC format (the first byte contains the string's length). This item is read only.
TN$_PROTOCOL	4	Session protocol. (See Table 6–9.)
TN$_REMOTE_ADDRESS	32	Remote peer's `sockaddr` of the active connection. Note that the `sockaddr` is in BSD Version 4.4 format, which includes a `sockaddr` size field. The size of the `sockaddr` should be determined from this field. This item is read only.
TN$_SERVICE_TYPE	4	Class of terminal service. (See Table 6–10.)
TN$_STATUS	4	Current device and session status. This item is read only.

6.3.1.2 Characteristic Mask Bits

Table 6–8 describes the characteristic mask bits used with the **p5** parameter.

Table 6–8 Characteristic Mask Bits

Characteristic	Description
TN$M_AUTOCONNECT	The device supports automatic connect/reconnect.
TN$M_LOGIN_ON_DASSGN	Initiate a login when the TELNET device is deassigned. This characteristic requires the BYPASS or SYSNAM privilege or executive or kernel mode calls.
TN$M_LOGIN_TIMER	Used in conjunction with TN$M_LOGIN_ON_ DASSGN, this bit indicates that the login completion timer applies. If the TN device fails to login within 60 seconds, the connection will be broken and the device deallocated. This characteristic requires the BYPASS or SYSNAM privileges or executive or kernel mode calls.
TN$M_PERMANENT_UCB	The TELNET device is to remain until explicitly deleted.
TN$M_RETAIN_ON_DASSGN	The TELNET device is not to be deleted upon the deassignment of the last channel to this device. This condition is cleared on this last deassignment, so that a subsequent assign and deassign will result in the device being deleted.
TN$M_VIRTUAL_TERMINAL	When logging in under this device, a virtual terminal is to be created by TTDRIVER.

6.3.1.3 Protocol Types

Table 6–9 describes the protocol types used with the **p5** parameter.

Table 6–9 Protocol Type Codes

Protocol Type	Description
TN$K_PROTOCOL_UNDEFINED	There is no explicit protocol for this session. Data is transmitted and received on the socket without any interpretation. This is a raw connection.
TN$K_PROTOCOL_NVT	Network Virtual Terminal (NVT) protocol. The protocol understands basic session control but does not include the options negotiation present in the TELNET protocol.
TN$K_PROTOCOL_RLOGIN	BSD Remote Login protocol. This simple protocol provides some special control character support but lacks the architecture independence of the NVT and TELNET protocols.
TN$K_PROTOCOL_TELNET	TELNET protocol. Including the basic NVT protocol, TELNET adds support for options negotiation. This can provide an enhanced terminal session depending upon the client and server involved.

6.3.1.4 Service Types

Table 6–10 describes the service type codes used with the **p5** parameter.

Table 6–10 Service Type Codes

Service Type	Description
TN$K_SERVICE_NONE	The service type is not currently known.
TN$K_SERVICE_INCOMING	The service is an incoming connection.
TN$K_SERVICE_OUTGOING	The service is an outgoing connection.

6.3.2 Passing Parameters to the TELNET Port Driver

The IO$_TTY_PORT function is used to pass $QIO parameters through the terminal driver to the TELNET port driver. The actual subfunction is encoded as an option mask and may be:

- IO$M_TN_STARTUP — Bind socket to a TELNET terminal.
- IO$M_TN_SHUTDOWN — Unbind socket from a TELNET terminal.

IO$_TTY_PORT | IO$M_TN_STARTUP

Bind socket to a TELNET terminal.

This subfunction will bind a created (connected) socket to a TELNET terminal device.

Arguments

p1
OpenVMS usage: channel
type: word (unsigned)
access: read only
mechanism: by value

The **p1** argument contains the channel number of the socket over which the TELNET session is to be established.

p2
OpenVMS usage: protocol_number
type: longword (unsigned)
access: read only
mechanism: by value

The **p2** argument contains the protocol selection.

p3
OpenVMS usage: characteristics_mask
type: longword (unsigned)
access: read only
mechanism: by value

The **p3** argument specifies a mask of characteristics to apply against the connection. See Table 6–8 for possible values.

Description

The IO$M_TN_STARTUP subfunction allows the application to communicate over a socket using the terminal driver QIO interface. Note that incoming and outgoing data is processed by the terminal driver, and that the terminal's characteristics may affect the format of the data. Be aware that by default, the terminal will echo incoming data back to the sender.

Once the subfunction completes, the application is free to perform all terminal QIO functions on the connection. While the socket is bound to a terminal device, it will process neither the IO$_READ*x*BLK nor the IO$_WRITE*x*BLK function, and will return the error SS$_DEVINUSE.

Condition Values Returned

SS$_IVCHAN	Programming error. The specified channel is not valid.

SS$_IVMODE	Programming error. The access mode of the channel is more privileged than the access mode of the terminal's channel.
SS$_NOPRIV	Programming error. The TN$M_LOGIN_ON_ DASSGN characteristic was specified in a characteristics mask from a QIO in USER or SUPERVISOR mode without either the BYPASS or SYSPRV privilege.
SS$_NOTNETDEV	Programming error. The specified channel is an assignment to a non-BG device.
SS$_PROTOCOL	Programming error. The specified protocol number is not valid, or the internet network is not available.

IO$_TTY_PORT | IO$M_TN_SHUTDOWN

Unbind socket from a TELNET terminal.

This subfunction will unbind a previously bound socket-terminal connection.

Arguments

p1
OpenVMS usage: channel
type: word (unsigned)
access: read only
mechanism: by value

The **p1** argument contains the channel number of the socket to establish the TELNET session.

Description

The IO$M_TN_SHUTDOWN subfunction allows the application to break a previously bound socket-terminal connection (created with IO$M_TN_STARTUP). The channel must be from an assignment to the same network pseudodevice in the socket-terminal connection.

Upon completion, the application retains the assignments to the connection and the TELNET terminal, but they are no longer related. Any subsequent IO$_READ*x*BLK or IO$_WRITE*x*BLK function on the socket channel will no longer return the error SS$_DEVINUSE.

Condition Values Returned

SS$_IVCHAN	Programming error. The specified channel is not valid.
SS$_IVMODE	Programming error. The access mode of the channel is more privileged than the access mode of the terminal's channel.
SS$_NOTNETDEV	Programming error. The specified channel is an assignment to a non-BG device.
SS$_DEVREQERR	Programming error. The device on the channel does not match the device in the socket-terminal connection.

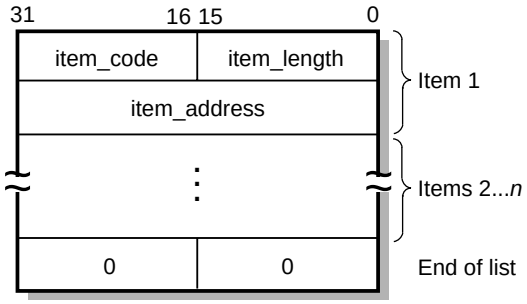
6.3.3 Buffered Reading and Writing of Item Lists

The IO$_TTY_PORT_BUFIO function is used to pass $QIO parameters through the terminal driver to the TELNET port driver. IO$_TTY_PORT_BUFIO differs from IO$_TTY_PORT in that certain subfunctions accept buffered item lists for reading or writing parameters to the terminal device.

- IO$M_TN_SENSEMODE — Read device parameters.
- IO$M_TN_SETMODE — Write device parameters.

The subfunctions of IO$_TTY_PORT_BUFIO accept an item list for input or output. Figure 6–1 shows the format of this item list.

Figure 6–1 Subfunction Item List



VM-0449A-AI

The item list is terminated with an item_code and item_length, both of which are zero.

The subfunctions of IO$_TTY_PORT_BUFIO can be combined into a single QIO. For example, the IO$M_TN_SETMODE and IO$M_TN_CONNECT can be combined to set the device's parameters and then to attempt to make a connection.

The subfunctions are performed in the following order:

1. IO$M_TN_SETMODE
2. IO$M_TN_CONNECT
3. IO$M_TN_SENSEMODE
4. IO$M_TN_DISCON

Note

Certain items are read only (IO$M_TN_SENSEMODE) and cannot be written (IO$M_TN_SETMODE). Normally, attempting to write such items would result in the error SS$_BADATTRIB. However, if a combination operation (IO$M_TN_SENSEMODE | IO$M_TN_SETMODE) is being performed, these items will *not* result in an error. Rather, the items will be ignored in the IO$M_TN_SETMODE processing, and the QIO will continue with IO$M_TN_SENSEMODE processing, returning the information that the item specifies.

IO$_TTY_PORT_BUFIO | IO$M_TN_SENSEMODE

Read device parameters.

Arguments

p5

OpenVMS usage:	item_list_2
type:	vector byte (unsigned)
access:	read only
mechanism:	by reference

The **p5** argument is the address of an item list that contains a summary of information to be read from the device.

Description

The IO$M_TN_SENSEMODE subfunction of IO$_TTY_PORT_BUFIO is used to read the parameters associated with a device.

Condition Values Returned

SS$_BADATTRIB	Programming error. The item code within the list is not valid. This could be because of its code, an attempt to write a read-only parameter, or inappropriate size. The address of the item's buffer is returned in the second longword of the I/O status block.
SS$_IVBUFLEN	Programming error. The length of the specified item is not acceptable. The address of the item's buffer is returned in the second longword of the I/O status block.
SS$_NOPRIV	Programming error. An item that requires a privilege which the requestor does not have is present in the item list. The address of the item's buffer is returned in the second longword of the I/O status block.

IO$_TTY_PORT_BUFIO | IO$M_TN_SETMODE

Write device parameters.

Arguments

p5

OpenVMS usage:	item_list_2
type:	vector (byte unsigned)
access:	read only
mechanism:	by reference

The **p5** argument is the address of an item list that contains a summary of information to be written to the device.

Description

The IO$M_TN_SETMODE subfunction of IO$_TTY_PORT_BUFIO is used to write the parameters associated with a device.

Condition Values Returned

SS$_BADATTRIB	Programming error. The item code within the list is not valid. This could be because of its code, an attempt to write a read-only parameter, or inappropriate size. The address of the item's buffer is returned in the second longword of the I/O status block.
SS$_DUPLNAM	Programming error. An attempt to set the device's unit number via the TN$_DEVICE_ UNIT item has failed because that specified unit number was already present.
SS$_IVBUFLEN	Programming error. The length of the specified item is not acceptable. The address of the item's buffer is returned in the second longword of the I/O status block.
SS$_NOPRIV	Programming error. An item that requires a privilege which the requester does not have is present in the item list. The address of the item's buffer is returned in the second longword of the I/O status block.

A

Socket Options

This appendix describes the socket options that you can set with the Sockets API `setsockopt( )` function and the $QIO system service IO$_SETMODE and IO$_SETCHAR I/O function codes. You can query the value of these socket options using the Sockets API `getstockopt( )` function or the $QIO system service IO$_SENSEMODE or IO$_SENSECHAR I/O function code.

The following tables list:

- Socket Options
- TCP Protocol Options
- IP Protocol Options

Table A–1 lists the socket options that are set at the SOL_SOCKET level and their Sockets API and system service symbol names.

Table A–1 Socket Options

Sockets API Symbol	System Service Symbol	Description
SO_BROADCAST	TCPIP$C_BROADCAST	Permits the sending of broadcast messages. Takes an integer parameter and requires a system user identification code (UIC) or SYSPRV, BYPASS, or OPER privilege. Optional for a connectionless datagram.
SO_DONTROUTE	TCPIP$C_DONTROUTE	Indicates that outgoing messages should bypass the standard routing facilities. Instead, the messages are directed to the appropriate network interface according to the network portion of the destination address.
SO_ERROR	TCPIP$C_ERROR	Obtains the socket error status and clears the error on the socket.
SO_FULL_DUPLEX_CLOSE	TCPIP$C_FULL_DUPLEX_CLOSE	When set before a close operation, the receive and transmit sides of the communications are closed.
SO_KEEPALIVE	TCPIP$C_KEEPALIVE	Keeps connections active. Enables the periodic transmission of keepalive probes to the remote system. If the remote system fails to respond to the keepalive probes, the connection is broken. If the SO_KEEPALIVE option is enabled, the values of TCP_KEEPCNT, TCP_KEEPINTVL and TCP_KEEPIDLE affect TCP behavior on the socket.

(continued on next page)

Table A–1 (Cont.) Socket Options

Sockets API Symbol	System Service Symbol	Description
SO_LINGER	TCPIP$C_LINGER	Lingers on a `close( )` function if data is present. Controls the action taken when unsent messages queue on a socket and a `close( )` function is performed. Uses a `linger` structure parameter defined in SOCKET.H to specify the state of the option and the linger interval. If SO_LINGER is specified, the system blocks the process during the `close( )` function until it can transmit the data or until the time expires. If the option is not specified and a `close( )` function is issued, the system allows the process to resume as soon as possible.
SO_OOBINLINE	TCPIP$C_OOBINLINE	When this option is set, out-of-band data is placed in the normal input queue. When `SO_OOBINLINE` is set, the `MSG_OOB` flag to the receive functions cannot be used to read the out-of-band data. A value of 0 disables the option, and a nonzero value enables the option.
SO_RCVBUF	TCPIP$C_RCVBUF	Sets the receive buffer size, in bytes. Takes an integer parameter and requires a system UIC or SYSPRV, BYPASS, or OPER privilege.
SO_RCVTIMEO	TCPIP$C_RCVTIMEO	For Compaq use only. Sets the timeout value for a `recv( )` operation. The argument to the two `sockopt` functions is a pointer to a `timeval` structure containing an integer value specified in seconds.
SO_REUSEADDR	TCPIP$C_REUSEADDR	Specifies that the rules used in validating addresses supplied by a `bind( )` function should allow reuse of local addresses. A value of 0 disables the option, and a non-zero value enables the option. The SO_REUSEPORT option is automatically set when an application sets SO_REUSEADDR
SO_REUSEPORT	TCPIP$C_REUSEPORT	Allows more than one process to receive UDP datagrams destined for the same port. The `bind( )` call that binds a process to the port must be preceded by a `setsockopt( )` call specifying this option. SO_REUSEPORT is automatically set when an application sets the SO_REUSEADDR option.
SO_SHARE	TCPIP$C_SHARE	Allows multiple processes to share the socket.

(continued on next page)

Table A–1 (Cont.) Socket Options

Sockets API Symbol	System Service Symbol	Description
SO_SNDBUF	TCPIP$C_SNDBUF	Sets the send buffer size in bytes. Takes an integer parameter and requires a system UIC or SYSPRV, BYPASS, or OPER privilege. Optional for a connectionless datagram.
SO_SNDLOWAT	TCPIP$C_SNDLOWAT	Sets the low-water mark for a `send( )` operation. The send low-water mark is the amount of space that must exist in the socket send buffer for `select( )` to return writeable. Takes an integer value specified in bytes.
SO_SNDTIMEO	TCPIP$C_SNDTIMEO	For Compaq use only. Sets the timeout value for a `send( )` operation. The argument to the two `sockopt( )` functions is a pointer to a `timeval` structure containing an integer value specified in seconds.
SO_TYPE	TCPIP$C_TYPE	Obtains the socket type.
SO_USELOOPBACK	TCPIP$C_USELOOPBACK	For Compaq use only. This option applies only to sockets in the routing domain (AF_ROUTE), When you enable this option, the socket receives a copy of everything sent on the socket.

Table A–2 lists the TCP protocol options that are set at the IPPROTO_TCP level and their Sockets API and system service symbol names.

Table A–2 TCP Protocol Options

Sockets API Symbol	System Service Symbol	Description
TCP_KEEPCNT	TCPIP$C_TCP_KEEPCNT	When the SO_KEEPALIVE option is enabled, TCP sends a keepalive probe to the remote system of a connection that has been idle for a period of time. If the remote system does not respond to the keepalive probe, TCP retransmits a keepalive probe for a certain number of times before a connection is considered to be broken. The TCP_KEEPCNT option specifies the maximum number of keepalive probes to be sent. The value of TCP_KEEPCNT is an integer value between 1 and *n*, where *n* is the value of the systemwide `tcp_keepcnt` parameter. The default value for for the systemwide parameter, `tcp_keepcnt`, is 8. To display the values of the systemwide parameters, enter the following command at the system prompt: `$ sysconfig -q inet` The default value for TCP_KEEPCNT is 8.

(continued on next page)

Table A–2 (Cont.) TCP Protocol Options

Sockets API Symbol	System Service Symbol	Description
TCP_KEEPIDLE	TCPIP$C_TCP_KEEPIDLE	When the SO_KEEPALIVE option is enabled, TCP sends a keepalive probe to the remote system of a connection that has been idle for a period of time. If the remote system does not respond to the keepalive probe, TCP retransmits a keepalive probe for a certain number of times before a connection is considered to be broken. TCP_KEEPIDLE specifies the number of seconds before TCP will send the initial keepalive probe. The default value for TCP_KEEPIDLE is an integer value between 1 and *n*, where *n* is the value for the systemwide parameter `tcp_keepidle`. The default value for `tcp_keepidle`, specified in half-second units, is 150 (75 seconds). To display the values of the systemwide parameters, enter the following command at the system prompt: `$ sysconfig -q inet` The default value for TCP_KEEPIDLE is 75 seconds.
TCP_KEEPINIT	TCPIP$C_TCP_KEEPINIT	If a TCP connection cannot be established within a period of time, TCP will time out the connection attempt. The default timeout value for this initial connection establishment is 75 seconds. The TCP_KEEPINIT option specifies the number of seconds to wait before the connection attempt times out. For passive connections, the TCP_KEEPINIT option value is inherited from the listening socket. The value of TCP_KEEPINIT is an integer between 1 and *n*, where *n* is the value for the systemwide parameter `tcp_keepinit`. The default value of the systemwide parameter `tcp_keepinit`, specified in half-second units, is 150 (75 seconds). To display the values of the systemwide parameters, enter the following command at the system prompt: `$ sysconfig -q inet` The TCP_KEEPINIT option does not require the SO_KEEPALIVE option to be enabled.
TCP_KEEPINTVL	TCPIP$C_TCP_KEEPINTVL	When the SO_KEEPALIVE option is enabled, TCP sends a keepalive probe to the remote system on a connection that has been idle for a period of time. If the remote system does not respond to a keepalive probe, TCP retransmits the keepalive probe after a period of time. The default value for this retransmit interval is 75 seconds. The TCP_KEEPINTVL option specifies the number of seconds to wait before retransmitting a keepalive probe. The value of the TCP_KEEPINTVL option is an integer between 1 and *n*, where *n* is the value of the systemwide parameter `tcp_keepintvl` which is specified in half-second units. The default value for the systemwide parameter `tcp_keepintvl` is 150 (75 seconds). To display the values of the systemwide parameters, enter the following command at the system prompt: `$ sysconfig -q inet` (continued on next page)

Table A–2 (Cont.) TCP Protocol Options

Sockets API Symbol	System Service Symbol	Description
TCP_NODELAY	TCPIP$C_TCP_NODELAY	Specifies that the send( ) operation not be delayed to merge packets. Under most circumstances, TCP sends data when it is presented. When outstanding data has not yet been acknowledged, TCP gathers small amounts of the data into a single packet and sends it when an acknowledgment is received. This functionality can cause significant delays for some clients that do not expect replies (such as windowing systems that send a stream of events from the mouse). The TCP_NODELAY disables the Nagle algorithm, which reduces the number of small packets on a wide area network.
TCP_MAXSEG	TCPIP$C_TCP_MAXSEG	Sets the maximum transmission unit (MTU) of a TCP segment to a specified integer value from 1 to 65535. The default is 576 bytes. Can only be set before a listen( ) or connect( ) operation on the socket. For passive connections, the value is obtained from the listening socket. Note that TCP does not use an MTU value that is less than 32 or greater than the local network's MTU. Setting the option to zero results in the default behavior.
TCP_NODELACK	TCPIP$C_TCP_NODELACK	When specified, disables the algorithm that gathers outstanding data that has not been acknowledged and sends it in a single packet when acknowledgment is received. Takes an integer value.
TCP protocol options that are obsolete but provided for backward compatibility		
TCP_DROP_IDLE	TCPIP$C_TCP_DROP_IDLE	When the TCP_KEEPALIVE option is enabled, the TCP_DROP_IDLE option specifies the time interval after which a connection is dropped. The value of TCP_ DROP_IDLE is an integer specified in seconds. The default value is 600 seconds. When the TCP_DROP_IDLE option is set, the value of the TCP_KEEPCNT option is calculated as the value of TCP_DROP_IDLE divided by the value of TCP_KEEPINTVL. A call to getsockopt( ) function specifying the TCP_DROP_IDLE option returns the result of multiplying the values of TCP_KEEPCNT and TCP_KEEPINTVL.
TCP_PROBE_IDLE	TCPIP$C_TCP_PROBE_IDLE	When the TCP_KEEPALIVE option is enabled, the TCP_PROBE_IDLE option specifies the time interval between the keepalive probes and for the connections establishing the timeout. The default value for TCP_PROBE_IDLE is 75 seconds. The value of TCP_PROBE_IDLE is an integer specified in seconds. When this option is set, TCP_KEEPINTVL, TCP_KEEPIDLE and TCP_KEEPINIT are set to the value specified for TCP_PROBE_IDLE. A call to the getsockopt( ) function specifying the TCP_PROBE_IDLE option returns the value of TCP_KEEPINTVL.

Table A–3 lists options that are set at the IPPROTO_IP level and their Sockets API and system service symbol names.

Table A–3 IP Protocol Options

Sockets API Symbol	System Service Symbol	Description
IP_ADD_MEMBERSHIP	TCPIP$C_IP_ADD_MEMBERSHIP	Adds the host to the membership of a multicast group. A host must become a member of a multicast group before it can receive datagrams sent to the group. Membership is associated with a single interface; programs running on multihomed hosts may need to join the same group on more than one interface. Up to IP_MAX_MEMBERSHIPS (currently 20) memberships may be added on a single socket.
IP_DROP_MEMBERSHIP	TCPIP$C_IP_DROP_MEMBERSHIP	Removes the host from the membership of a multicast group.
IP_HDRINCL	TCPIP$C_IP_HDRINCL	If specified for a raw IP socket, you must build the IP header for all datagrams sent on the raw socket.
IP_MULTICAST_IF	TCPIP$C_IP_MULTICAST_IF	Specifies the interface for outgoing multicast datagrams sent on this socket. The interface is specified as an `in_addr` structure.
IP_MULTICAST_LOOP	TCPIP$C_IP_MULTICAST_LOOP	Disables loopback of local delivery. If a multicast datagram is sent to a group which the sending host is a member, a copy of the datagram is looped back by the IP layer for local delivery (the default). To disable the loopback delivery, specify a value of 0.
IP_MULTICAST_TTL	TCPIP$C_IP_MULTICAST_TTL	Specifies the time-to-live (TTL) value for outgoing multicast datagrams. Takes an integer value between 0 and 255:

Value	Action
0	Restricts distribution to applications running on the local host.
1	Forwards the multicast datagram to hosts on the local subnet.
2 - 255	With a multicast router attached to the sending host's network, forwards multicast datagrams beyond the local subnet. Multicast routers forward the datagram to known networks that have hosts belonging to the specified multicast group. The TTL value is decremented by each multicast router in the path. When the TTL value is decremented to zero, the datagram is no longer forwarded.

IP_OPTIONS	TCPIP$C_IP_OPTIONS	Provides IP options to be transmitted in the IP header of each outgoing packet.
IP_RECVDSTADDR	TCPIP$C_IP_RECVDSTADDR	Enables a SOCK_DGRAM socket to receive the destination IP address for a UDP datagram.

(continued on next page)

Table A–3 (Cont.) IP Protocol Options

Sockets API Symbol	System Service Symbol	Description
IP_RECVOPTS	TCPIP$C_IP_RECVOPTS	Enables a SOCK_DGRAM socket to receive IP options.
IP_TTL	TCPIP$C_IP_TTL	Time to live (TTL) for a datagram.
IP_TOS	TCPIP$C_IP_TOS	Type of service (1-byte value).

B

IOCTL Requests

The `ioctl( )` Sockets API function and the IO$_SENSEMODE/IO$_SENSECHAR and IO$_SETMODE/IO$_SETCHAR I/O function codes used with the $QIO system service perform I/O control functions on a network device (BG:).

Table B–1 lists the IOCTL requests supported by TCP/IP Services, their data types, the $QIO function code to use if using system services, and a description of the operation.

Table B–1 IOCTL Requests

IOCTL Request	Data Type	$QIO Function Code	Description
Socket Operations			
SIOCSHIWAT	int	IO$_SETMODE	Set high watermark.
SIOCGHIWAT	int	IO$_SENSEMODE	Get high watermark.
SIOCSLOWAT	int	IO$_SETMODE	Set low watermark.
SIOCGLOWAT	int	IO$_SENSEMODE	Get low watermark.
SIOCATMARK	int	IO$_SENSEMODE	Determines whether you are at the out-of-band character mark. The operation returns a nonzero value if the socket's read pointer is currently at the end-of-band mark or a zero value if the read pointer is not at the out-of-band mark. The value is returned in the integer pointed to by the third argument of the `ioct( )` call.
Interface Operations			
SIOCSIFADDR	struct ifreq[1]	IO$_SETMODE	Sets the interface address from the `ifr_addr` member. The initialization function for the interface is also called.
SIOCSIFDSTADDR	struct ifreq	IO$_SETMODE	Sets the point-to-point address from the `ifr_dstaddr` member.
SIOCSIFFLAGS	struct ifreq	IO$_SETMODE	Sets the interface flags from the `ifr_flags` member.

[1]Defined in the IF.H header file.

(continued on next page)

Table B–1 (Cont.) IOCTL Requests

IOCTL Request	Data Type	$QIO Function Code	Description
Interface Operations			
SIOCGIFFLAGS	struct ifreq	IO$_SENSEMODE	Returns the interface flags in the ifr_flags member. The flags indicate whether the interface is up (IFF_UP), is a point-to-point interface (IFF_POINTOPOINT), supports broadcasts (IFF_BROADCAST), and other flags.
SIOCSIFBRDADDR	struct ifreq	IO$_SETMODE	Sets the broadcast address from the ifr_broadaddr member.
SIOCSIFNETMASK	struct ifreq	IO$_SETMODE	Sets the subnet address mask from the ifr_addr member.
SIOCGIFMETRIC	struct ifreq	IO$_SENSEMODE	Returns the interface routing metric in the ifr_metric member. The interface metric is maintained by the kernel for each interface but is used by the routing daemon (routd). The interface metric is added to the hop count (to make an interface less favorable).
SIOCSIFMETRIC	struct ifreq	IO$_SETMODE	Sets the interface routing metric from the ifr_metric member.
SIOCDIFADDR	struct ifreq	IO$_SETMODE	Deletes an interface address
SIOCAIFADDR	struct ifaliasreq[1]	IO$_SETMODE	Adds or changes an interface alias.
SIOCPIFADDR	struct ifaliasreq	IO$_SETMODE	Sets the primary interface address.
SIOCADDMULTI	struct ifreq	IO$_SETMODE	Adds a multicast address.
SIOCDELMULTI	struct ifreq	IO$_SETMODE	Deletes a multicast address.
SIOCENABLBACK	struct ifreq	IO$_SETMODE	Enables the loopback interface.
SIOCDISABLBACK	struct ifreq	IO$_SETMODE	Disables the loopback interface.
SIOCSIPMTU	struct ifreq	IO$_SETMODE	Sets the interface IP MTU value.
SIOCRIPMTU	struct ifreq	IO$_SENSEMODE	Returns the interface IP MTU value.
SIOCGIFINDEX	struct ifreq	IO$_SENSEMODE	Returns the IF index value.
SIOCGMEDIAMTU	struct ifreq	IO$_SENSEMODE	Returns the value of the media MTU.
SIOCGIFTYPE	struct ifreq	IO$_SENSEMODE	Returns the interface type.
SIOCGIFADDR	struct ifreq	IO$_SENSEMODE	Returns the interface address.
SIOCGIFDSTADDR	struct ifreq	IO$_SENSEMODE	Returns the point-to-point interface address.
SIOCGIFBRDADDR	struct ifreq	IO$_SENSEMODE	Returns the interface broadcast address.

[1]Defined in the IF.H header file.

(continued on next page)

Table B–1 (Cont.) IOCTL Requests

IOCTL Request	Data Type	$QIO Function Code	Description
Interface Operations			
SIOCGIFCONF	struct ifconf[1]	IO$_SENSEMODE	Returns the interface list.
SIOCGIFNETMASK	struct ifreq	IO$_SENSEMODE	Returns the interface subnet address mask.
Routing Table Operations			
SIOCADDRT	struct ortentry[2]	IO$_SETMODE	Adds an entry to the routing table.
SIOCDELRT	struct ortentry	IO$_SETMODE	Deletes an entry from the routing table.
ARP Cache Operations			
SIOCSARP	struct arpreq[3]	IO$_SETMODE	Adds a new entry to or modifies an existing entry in the ARP table.
SIOCDARP	struct arpreq	IO$_SETMODE	Deletes an entry from the ARP table.
SIOCGARP	struct arpreq	IO$_SENSEMODE	Returns an ARP table entry.

[1]Defined in the IF.H header file.
[2]Defined in the ROUTE.H header file.
[3]Defined in the IF_ARP.H header file.

C
Data Types

As part of the OpenVMS common language environment, the TCP/IP system services data types provide compatibility between procedure calls that support many different high-level languages. Specifically, the OpenVMS data types apply to both Alpha and VAX architectures as the mechanism for passing argument data between procedures. This appendix describes the context and structure of the TCP/IP system services data types and identifies the associated declarations to each of the specific high-level language implementations.

C.1 OpenVMS Data Types

In Chapter 6, the OpenVMS usage entry in the TCP/IP Services documentation format for system services indicates the OpenVMS data type of the argument. Most data types can be considered conceptual types; that is, their meaning is unique in the context of the OpenVMS operating system. The OpenVMS data type `access_mode` is one example. The storage representation of this OpenVMS type is an unsigned byte, and the conceptual content of this unsigned byte is the fact that it designates a hardware access mode and therefore has only four valid values: 0, kernel mode; 1, executive mode; 2, supervisor mode; and 3, user mode. However, some OpenVMS data types are not conceptual types; that is, they specify a storage representation but carry no other semantic content in the OpenVMS context. For example, the data type `byte_signed` is not a conceptual type.

Note

The OpenVMS usage entry is not a traditional data type such as the OpenVMS standard data types—byte, word, longword, and so on. It is significant only within the OpenVMS operating system environment and is intended solely to expedite data declarations within application programs.

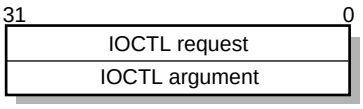
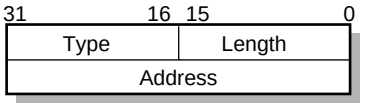
To use the OpenVMS usage entry, perform the following steps:

1. Find the data type in Table C–1 and read its definition.
2. Find the same OpenVMS data type in the C and C++ language implementation table (Table C–2) and its corresponding source language type declaration.
3. Use this code as your type declaration in your application program. Note that, in some instances, you might have to modify the declaration.
4. For all other OpenVMS data types not listed in Table C–2, refer to Appendix F of the *OpenVMS Programming Concepts, Volume 2* manual.

Data Types
C.1 OpenVMS Data Types

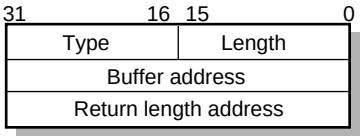
For both Alpha and VAX architectures, Table C–1 lists and describes OpenVMS data type declarations for the OpenVMS usage entry of system services unique to TCP/IP Services.

Table C–1 TCP/IP Services Usage Data Type Entries

Data Type	Definition
`buffer_list`	Structure that consists of one or more descriptors defining the length and starting address of user buffers. On VAX systems, each descriptor is a 32-bit fixed-length descriptor. On Alpha systems, each descriptor can be a 32- or 64-bit fixed-length descriptor. For more information concerning descriptors, see the OpenVMS Calling Standard.
`input_parameter_list`	Structure that consists of one or more `item_list_2` or `ioctl_comm` structures. Each `item_list_2` structure describes an individual parameter that can be set by a service. Such parameters include socket or protocol options as identified by the item's type field. Each `ioctl_comm` structure describes an IOCTL command; its encoded request code and address of its associated argument.
`ioctl_comm`	Quadword structure that describes an IOCTL command's encoded request code and address of its associated argument. It contains two longword fields, as depicted in the following diagram:  VM-0563A-AI The first field is a longword containing the IOCTL encoded request code specifying the type of I/O control operation to be performed. The second field is a longword containing the address of a variable or a data structure targeted by this IOCTL command.
`item_list_2`	Quadword structure that describes the size, data type, and starting address of a user-supplied data item. It contains three fields, as depicted in the following diagram:  VM-0558A-AI The first field is a word containing the length (in bytes) of the user-supplied data item being described. The second field is a word containing a symbolic code specifying the data type of the user-supplied data item. The third field is a longword containing the starting address of the user-supplied data item.

(continued on next page)

Table C–1 (Cont.) TCP/IP Services Usage Data Type Entries

Data Type	Definition
item_list_2 descriptor	An item_list_2 structure, used as an argument descriptor and containing structural information about the argument's type and the address of a data item. This data item is associated with the argument. The format of this descriptor is unique to TCP/IP Services and supplements argument descriptors defined in the OpenVMS Calling Standard.
item_list_3	A 3-longword structure that describes the size, data type, and address of a buffer in which a service writes information. It contains four fields, as depicted in the following diagram:  VM-0559A-AI The first field is a word containing the length (in bytes) of the buffer in which a service writes information. The length of the buffer needed depends on the data type specified in the type field. If the value of buffer length is too small, the service truncates the data. The second field is a word containing a symbolic code and specifies the type of information that a service is to return. The third field is a longword containing the address of the buffer in which a service writes the information. The fourth field is a longword containing the address of a longword in which a service writes the length in bytes of the information it actually returned.
item_list_3 descriptor	An item_list_3 structure, used as an argument descriptor and containing structural information about the argument's type and the address of a buffer used to return service information. This buffer is associated with the argument. The format of this descriptor is unique to TCP/IP Services and supplements argument descriptors defined in the OpenVMS Calling Standard.
output_parameter_list	Structure that consists of one or more item_list_3 or ioctl_comm structures. Each item_list_3 structure describes an individual parameter that can be returned by a service. Such parameters include socket or protocol options as identified by the item's type field. Each ioctl_comm structure describes an IOCTL command, its encoded request code, and the address of its associated argument.

(continued on next page)

Table C–1 (Cont.) TCP/IP Services Usage Data Type Entries

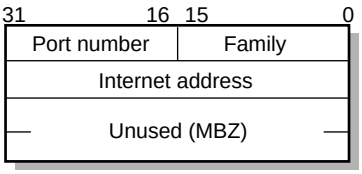
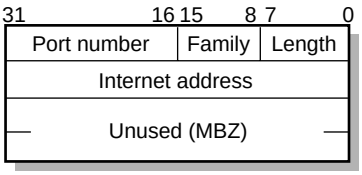
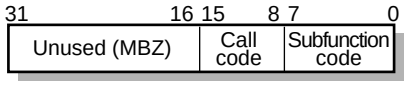
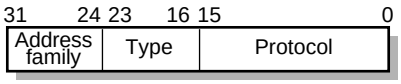
Data Type	Definition
socket_name	Internet domain socket address structure that consists of a internet address and a port number. The layout of socket address structures varies between BSD Version 4.3 and BSD Version 4.4. BSD Version 4.3 specifies a 16-byte socket address structure. It contains four fields, as depicted in the following diagram:  VM-0566A-AI The first field is a word identifying a socket address structure as belonging to the internet domain (always a value of 2). The second field is a word containing a 16-bit port number (stored in network byte order) used to demultiplex transport-level messages. The third field is a longword containing a 32-bit internet address (stored in network byte order). The fourth field is a quadword. It is unused but must be initialized to all zeros. BSD Version 4.4 specifies a 16-byte socket address structure. It contains five fields, as depicted in the following diagram:  VM-0567A-AI The first field is a byte containing the size of this socket address structure (always a value of 16). The second field is a byte identifying a socket address structure as belonging to the internet domain (always a value of 2). The third field is a word containing a 16-bit port number (stored in network byte order) used to demultiplex transport-level messages. The fourth field is a longword containing a 32-bit internet address (stored in network byte order). The fifth field is a quadword. It is unused but must be initialized to all zeros. (continued on next page)

Table C–1 (Cont.) TCP/IP Services Usage Data Type Entries

Data Type	Definition
subfunction_code	Longword structure specifying the exact operation an IO$_ ACPCONTROL function is to perform. This structure has three fields, as depicted in the following diagram:  VM-0568A-AI The first field is a byte specifying the network ACP operation. The second field is a byte specifying the network ACP suboperation. The third field is word that is unused but must be initialized to all zeros (MBZ).
socket_characteristics	Longword structure specifying the address family, socket type, and protocol of a new socket. This structure has three fields, as depicted in the following diagram:  VM-0569A-AI The first field is a word specifying the protocol to be used with the socket. The second field is a byte specifying the socket type. The third field is a byte specifying the address family.

C.2 C and C++ Implementations

Table C–2 lists the OpenVMS data types and their corresponding C and C++ data type declarations.

Table C–2 C and C++ Implementations

OpenVMS Data Types	C an C++ Implementations
buffer_list	User defined[1]
input_parameter_list	User defined[1]

ioctl_comm	`struct ioctl_comm`
	`{`
	`int ioctl_req;  /* ioctl request code */`
	`void *ioctl_arg;  /* ioctl argument */`
	`}`

[1]The declaration of a user-defined data structure depends on how the data will be used. Such data structures can be declared in a variety of ways, each of which is suitable only to specific applications.

(continued on next page)

Table C–2 (Cont.) C and C++ Implementations

OpenVMS Data Types	C an C++ Implementations

item_list_2	struct item_list_2
	{
	unsigned short length; /* item length */
	unsigned short type; /* item type */
	void *address; /* item address */
	}

item_list_2 descriptor	struct item_list_2
	{
	unsigned short length; /* argument length */
	unsigned short type; /* argument type */
	void *address; /* argument address */
	}

item_list_3	struct item_list_3
	{
	unsigned short length; /* buffer length */
	unsigned short type; /* buffer type */
	void *address; /* buffer address */
	unsigned int *retlen; /* buffer returned */
	/* length address */
	}

item_list_3 descriptor	struct item_list_3
	{
	unsigned short length; /* argument length */
	unsigned short type; /* argument type */
	void *address; /* argument address */
	unsigned int *retlen; /* argument returned */
	/* length address */
	}

output_parameter_list	User defined[1]
socket_name	#include <in.h> struct sockaddr_in

subfunction_code	struct acpfunc
	{
	unsigned char code; /* subfunction code */
	unsigned char type; /* call code */
	unsigned short reserved; /* reserved */
	/* (must be zero) */
	}

socket_characteristics	struct sockchar
	{
	unsigned short prot; /* protocol */
	unsigned char type; /* type */
	unsigned char af; /* address format */
	}

[1]The declaration of a user-defined data structure depends on how the data will be used. Such data structures can be declared in a variety of ways, each of which is suitable only to specific applications.

D
Error Codes

This appendix contains a table of Sockets API error codes and their equivalent OpenVMS system service status codes (Table D–1). The Sockets API functions use the Compaq C compiler.

Table D–1 Translation of Socket Error Codes to OpenVMS Status Codes

Sockets (Compaq C) Error Code	OpenVMS System Service Status Code	Meaning
0	SS$_NORMAL	Success
1 EPERM	SS$_ABORT	Not owner
2 ENOENT	SS$_ABORT	No such file or directory
3 ESRCH	SS$_NOSUCHNODE	No such process
4 EINTR	SS$_ABORT	Interrupted system call
5 EIO	SS$_ABORT	I/O error
6 ENXIO	SS$_NOSUCHDEV	No such device or address
7 E2BIG	SS$_ABORT	Argument list too long
8 ENOEXEC	SS$_ABORT	Execution format error
9 EBADF	SS$_BADPARAM	Bad file number
10 ECHILD	SS$_ABORT	No children
11 EAGAIN	SS$_ABORT	No more processes
12 ENOMEM	SS$_INSFMEM	Not enough core
13 EACCES	SS$_ABORT	Permission denied
14 EFAULT	SS$_ACCVIO	Bad address
15 ENOTBLK	SS$_ABORT	Block device required
16 EBUSY	SS$_ABORT	Mount device busy
17 EEXIST	SS$_FILALRACC	File exists
18 EXDEV	SS$_ABORT	Cross-device link
19 ENODEV	SS$_ABORT	No such device
20 ENOTDIR	SS$_ABORT	Not a directory
21 EISDIR	SS$_ABORT	Is a directory
22 EINVAL	SS$_BADPARAM	Invalid argument
23 ENFILE	SS$_ABORT	File table overflow
24 EMFILE	SS$_ABORT	Too many open files

(continued on next page)

Table D–1 (Cont.) Translation of Socket Error Codes to OpenVMS Status Codes

Sockets (Compaq C) Error Code	OpenVMS System Service Status Code	Meaning
25 ENOTTY	SS$_ABORT	Not a typewriter
26 ETXTBSY	SS$_ABORT	Text file busy
27 EFBIG	SS$_ABORT	File too large
28 ENOSPC	SS$_ABORT	No space left on device
29 ESPIPE	SS$_ABORT	Illegal seek
30 EROFS	SS$_ABORT	Read-only file system
31 EMLINK	SS$_ABORT	Too many links
32 EPIPE	SS$_LINKDISCON	Broken pipe
33 EDOM	SS$_BADPARAM	Argument too large
34 ERANGE	SS$_TOOMUCHDATA	Result too large
35 EWOULDBLOCK	SS$_SUSPENDED	Operation would block
36 EINPROGRESS	SS$_ABORT	Operation now in progress
37 EALREADY	SS$_ABORT	Operation already in progress
38 ENOTSOCK	SS$_NOTNETDEV	Socket operation on nonsocket
39 EDESTADDRREQ	SS$_NOSUCHNODE	Destination address required
40 EMSGSIZE	SS$_TOOMUCHDATA	Message too long
41 EPROTOTYPE	SS$_PROTOCOL	Protocol wrong type for socket
42 ENOPROTOOPT	SS$_PROTOCOL	Protocol not available
43 EPROTONOSUPPORT	SS$_PROTOCOL	Protocol not supported
44 ESOCKTNOSUPPORT	SS$_PROTOCOL	Socket type not supported
45 EOPNOTSUPP	SS$_ILLCNTRFUNC	Operation not supported on socket
46 EPFNOSUPPORT	SS$_PROTOCOL	Protocol family not supported
47 EAFNOSUPPORT	SS$_PROTOCOL	Address family not supported
48 EADDRINUSE	SS$_DUPLNAM	Address already in use
49 EADDRNOTAVAIL	SS$_IVADDR	Requested address cannot be assigned
50 ENETDOWN	SS$_UNREACHABLE	Network is down
51 ENETUNREACH	SS$_UNREACHABLE	Network is unreachable
52 ENETRESET	SS$_RESET	Network dropped connection on reset
53 ECONNABORTED	SS$_LINKABORT	Software caused connection abort
54 ECONNRESET	SS$_CONNECFAIL	Connection reset by peer
55 ENOBUFS	SS$_INSFMEM	No buffer space available
56 EISCONN	SS$_FILALRACC	Socket is already connected
57 ENOTCONN	SS$_NOLINKS	Socket is not connected
58 ESHUTDOWN	SS$_SHUT	Cannot send after socket shutdown
59 ETOOMANYREFS	SS$_ABORT	Too many references, cannot splice

(continued on next page)

Table D–1 (Cont.) Translation of Socket Error Codes to OpenVMS Status Codes

Sockets (Compaq C) Error Code	OpenVMS System Service Status Code	Meaning
60 ETIMEDOUT	SS$_TIMEOUT	Connection timed out
61 ECONNREFUSED	SS$_REJECT	Connection refused
62 ELOOP	SS$_ABORT	Too many levels of symbolic links
63 ENAMETOOLONG	SS$_ABORT	File name too long
64 EHOSTDOWN	SS$_SHUT	Host is down
65 EHOSTUNREACH	SS$_UNREACHABLE	No route to host

E
Programming Examples

Table E–1 lists the sample programs contained in this appendix and on line in the directory specified by the TCPIP$EXAMPLES system logical. See Table 1–4 and Table 1–5 for a complete list of all the sample programs provided on line in the TCPIP$EXAMPLES directory.

Table E–1 Client/Server Programming Examples

File	Refer to...
TCPIP$TCP_CLIENT_SOCK.C	Section E.1.1
TCPIP$TCP_SERVER_SOCK.C	Section E.1.2
TCPIP$TCP_SERVER_SOCK_AUXS.C	Section E.1.3
TCPIP$TCP_CLIENT_QIO.C	Section E.2.1
TCPIP$TCP_SERVER_QIO.C	Section E.2.2
TCPIP$TCP_SERVER_QIO_AUXS.C	Section E.2.3
TCPIP$UDP_CLIENT_SOCK.C	Section E.3.1
TCPIP$UDP_SERVER_SOCK.C	Section E.3.2
TCPIP$UDP_CLIENT_QIO.C	Section E.4.1
TCPIP$UDP_SERVER_QIO.C	Section E.4.2

E.1 TCP Client/Server Examples (Sockets API)

This section contains examples that show the following:

- A TCP/IP IPv4 client using BSD Version 4.*x* Sockets API to handle network I/O operations.
- A TCP/IP IPv4 server using BSD Version 4.*x* Sockets API to handle network I/O operations.
- A TCP/IP IPv4 server using BSD Version 4.*x* Sockets API to handle network I/O operations, and how the server accepts connections from the auxiliary server.

E.1.1 TCP Client

Example E–1 shows how a typical TCP IPv4 client uses the Sockets API to handle the tasks of creating a socket, initiating server connections, reading service connection data, and then terminating the server connections.

Example E–1 TCP Client (Sockets API)

```
#pragma module  tcpip$tcp_client_sock                  \
                "V5.1-00"
/*
 * Copyright 2000 Compaq Computer Corporation
 *
 * COMPAQ Registered in U.S. Patent and Trademark Office.
 *
 * Confidential computer software.  Valid license from  Compaq
 * or  authorized sublicensor required for possession, use  or
 * copying.  Consistent with FAR 12.211 and 12.212, Commercial
 * Computer Software,   Computer Software Documentation,   and
 * Technical Data for Commercial Items  are  licensed  to  the
 * U.S. Government under vendor's standard commercial license.
 *
 * ++
 * FACILITY:
 *
 *    EXAMPLES
 *
 * ABSTRACT:
 *
 *    This is an example of a TCP/IP IPv4 client using  4.x BSD
 *    socket  Application Programming Interface (API) to handle
 *    network I/O operations.
 *
 *    Refer to 'Build, Configuration, and Run Instructions' for
 *    details on how to build, configure, and run this program.
 *
 * ENVIRONMENT:
 *
 *    OpenVMS Alpha/VAX V7.1
 *    TCP/IP Services V5.0 or higher
 *
 * AUTHOR:
 *
 *    TCPIP Development Group,  CREATION DATE: 23-May-1989
 *
 * --
```

(continued on next page)

Example E–1 (Cont.) TCP Client (Sockets API)

```
 */
       /* Build, Configuration, and Run Instructions */

/*
 * BUILD INSTRUCTIONS:
 *
 *    To build this example program use commands of the form,
 *
 *    using the DEC "C" compiler:
 *
 *      $ cc/prefix=all TCPIP$TCP_CLIENT_SOCK.C
 *      $ link TCPIP$TCP_CLIENT_SOCK
 *
 *    using the DEC "C++" compiler:
 *
 *      $ cxx/prefix=all/define=VMS TCPIP$TCP_CLIENT_SOCK.C
 *      $ link TCPIP$TCP_CLIENT_SOCK
 *
 *
 * CONFIGURATION INSTRUCTIONS:
 *
 *    No special configuration required.
 *
 *
 * RUN INSTRUCTIONS:
 *
 *    To run this example program:
 *
 *    1) Start the client's server program as shown below:
 *
 *       $ run tcpip$tcp_server_sock
 *       Waiting for a client connection on port: m
 *
 *    2) After the server program blocks, start this client program,
 *       entering the server host as shown below:
 *
 *       $ run tcpip$tcp_client_sock
 *       Enter remote host:
 *
 *       Note: You can specify a server host by using either an IPv4
 *             address in dotted-decimal notation (e.g. 16.20.10.56)
 *             or a host domain name (e.g. serverhost.compaq.com).
 *
 *    3) The client program then displays server connection information
 *       and server data as shown below:
 *
 *       Initiated connection to host: a.b.c.d, port: n
 *       Data received: Hello, world!
 *
 *    You can enter "ctrl/z" at any user prompt to terminate program
 *    execution.
 *
 */

/*
 * INCLUDE FILES:
 */
```

(continued on next page)

Example E–1 (Cont.) TCP Client (Sockets API)

```
#include <in.h>                    /* define internet related constants,   */
                                   /* functions, and structures            */
#include <inet.h>                  /* define network address info          */

#include <netdb.h>                 /* define network database library info */

#include <socket.h>                /* define BSD 4.x socket api            */
#include <stdio.h>                 /* define standard i/o functions        */
#include <stdlib.h>                /* define standard library functions    */
#include <string.h>                /* define string handling functions     */

#include <unixio.h>                /* define unix i/o                      */

/*
 * NAMED CONSTANTS:
 */

#define BUFSZ           1024            /* user input buffer size           */
#define SERV_PORTNUM    12345           /* server port number               */


/*
 * FORWARD REFERENCES:
 */

int  main( void );                      /* client main                      */
void get_serv_addr( void * );           /* get server host address          */

        /* Client Main */

/*
 * FUNCTIONAL DESCRIPTION:
 *
 *    This is the client's main-line code. It handles all the tasks of the
 *    client including:  socket creation,  initiating server  connections,
 *    reading server connection data, and terminating server connections.
 *
 *    This example program implements a typical TCP IPv4 client using  the
 *    BSD socket API to handle network i/o operations as shown below:
 *
 *      1) To create a socket:
 *
 *         socket()
 *
 *      2) To initiate a connection:
 *
 *         connect()
 *
 *      3) To transfer data:
 *
 *         recv()
 *
 *      4) To shutdown a socket:
 *
 *         shutdown()
 *
 *      5) To close a socket:
 *
 *         close()
 *
 *    This function is invoked by the DCL "RUN" command (see below);  the
 *    function's  completion  status is interpreted by DCL and if needed,
 *    an error message is displayed.
 *
```

(continued on next page)

Example E–1 (Cont.) TCP Client (Sockets API)

```
 * SYNOPSIS:
 *
 *    int main( void )
 *
 * FORMAL PARAMETERS:
 *
 *    ** None **
 *
 * IMPLICIT INPUTS:
 *
 *    ** None **
 *
 * IMPLICIT OUTPUTS:
 *
 *    ** None **
 *
 * FUNCTION VALUE:
 *
 *    completion status
 *
 * SIDE EFFECTS:
 *
 *    ** None **
 *
 */
int
main( void )
{
    int sockfd;                         /* connection socket descriptor     */
    char buf[512];                      /* client data buffer               */
    struct sockaddr_in serv_addr;       /* server socket address structure  */
    /*
     * init server's socket address structure
     */
    memset( &serv_addr, 0, sizeof(serv_addr) );
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port   = htons( SERV_PORTNUM );
    get_serv_addr( &serv_addr.sin_addr );
    /*
     * create connection socket
     */
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }
    /*
     * connect to specified host and port number
     */
    printf( "Initiated connection to host: %s, port: %d\n",
            inet_ntoa(serv_addr.sin_addr), ntohs(serv_addr.sin_port)
          );
```

(continued on next page)

Example E–1 (Cont.) TCP Client (Sockets API)

```
    if ( connect(sockfd,
                (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0 )
        {
        perror( "Failed to connect to server" );
        exit( EXIT_FAILURE );
        }

    /*
     * connection established with a server;
     * now attempt to read on this connection
     */

    if ( recv(sockfd, buf, sizeof(buf), 0) < 0 )
        {
        perror( "Failed to read data from server connection" );
        exit( EXIT_FAILURE );
        }

    printf( "Data received: %s\n", buf );  /* output client's data buffer  */

    /*
     * shutdown connection socket (both directions)
     */
    if ( shutdown(sockfd, 2) < 0 )
        {
        perror( "Failed to shutdown server connection" );
        exit( EXIT_FAILURE );
        }

    /*
     * close connection socket
     */

    if ( close(sockfd) < 0 )
        {
        perror( "Failed to close socket" );
        exit( EXIT_FAILURE );
        }

    exit( EXIT_SUCCESS );
}

        /* Get Server Host Address */
```

(continued on next page)

Example E–1 (Cont.) TCP Client (Sockets API)

```
/*
 * FUNCTIONAL DESCRIPTION:
 *
 *    This function gets the server host's address from the user and then
 *    stores it in the server's socket address structure.  Note that  the
 *    user can specify a server host by using either an IPv4  address  in
 *    dotted-decimal notation (e.g. 16.20.10.126) or a host  domain  name
 *    (e.g. serverhost.compaq.com).
 *
 *    Enter "ctrl/z" to terminate program execution.
 *
 * SYNOPSIS:
 *
 *    void get_serv_addr( void *addrptr )
 *
 * FORMAL PARAMETERS:
 *
 *    addrptr  -  pointer to socket address structure's 'sin_addr' field
 *                to store the specified network address
 *
 * IMPLICIT INPUTS:
 *
 *    ** None **
 *
 * IMPLICIT OUTPUTS:
 *
 *    ** None **
 *
 * FUNCTION VALUE:
 *
 *    ** None **
 *
 * SIDE EFFECTS:
 *
 *    Program execution is terminated if unable to read user's input
 *
 */
void
get_serv_addr( void *addrptr )
{
    char buf[BUFSZ];
    struct in_addr val;
    struct hostent *host;

    while ( TRUE )
        {
        printf( "Enter remote host: " );

        if ( fgets(buf, sizeof(buf), stdin) == NULL )
            {
            printf( "Failed to read User input\n" );
            exit( EXIT_FAILURE );
            }

        buf[strlen(buf)-1] = 0;

        val.s_addr = inet_addr( buf );
```

(continued on next page)

Example E–1 (Cont.) TCP Client (Sockets API)

```
        if ( val.s_addr != INADDR_NONE )
            {
            memcpy( addrptr, &val, sizeof(struct in_addr) );
            break;
            }
        if ( (host = gethostbyname(buf)) )
            {
            memcpy( addrptr, host->h_addr, sizeof(struct in_addr) );
            break;
            }
        }
}
```

E.1.2 TCP Server

Example E–2 shows how a typical TCP IPv4 server uses the Sockets API to handle the tasks of creating a socket, accepting or rejecting client connections, writing client connection data, and then terminating client connections.

Example E–2 TCP Server (Sockets API)

```
#pragma module  tcpip$tcp_server_sock                  \
                "V5.1-00"
/*
 * Copyright 2000 Compaq Computer Corporation
 *
 * COMPAQ Registered in U.S. Patent and Trademark Office.
 *
 * Confidential computer software.  Valid license from  Compaq
 * or  authorized sublicensor required for possession, use  or
 * copying.  Consistent with FAR 12.211 and 12.212, Commercial
 * Computer Software,   Computer Software Documentation,   and
 * Technical Data for Commercial Items  are  licensed  to  the
 * U.S. Government under vendor's standard commercial license.
 *
 * ++
 * FACILITY:
 *
 *    EXAMPLES
 *
 * ABSTRACT:
 *
 *    This is an example of a TCP/IP IPv4 server using  4.x BSD
 *    socket Application Programming Interface (API) to  handle
 *    network I/O operations.
 *
 *    Refer to 'Build, Configuration, and Run Instructions' for
 *    details on how to build, configure, and run this program.
 *
 * ENVIRONMENT:
 *
 *    OpenVMS Alpha/VAX V7.1
 *    TCP/IP Services V5.0 or higher
 *
 * AUTHOR:
 *
 *    TCPIP Development Group,  CREATION DATE: 23-May-1989
 *
 * --
 */

       /* Build, Configuration, and Run Instructions */
```

(continued on next page)

Example E–2 (Cont.) TCP Server (Sockets API)

```
/*
 *  BUILD INSTRUCTIONS:
 *
 *    To build this example program use commands of the form,
 *
 *    using the DEC "C" compiler:
 *
 *      $ cc/prefix=all TCPIP$TCP_SERVER_SOCK.C
 *      $ link TCPIP$TCP_SERVER_SOCK
 *
 *    using the DEC "C++" compiler:
 *
 *      $ cxx/prefix=all/define=VMS TCPIP$TCP_SERVER_SOCK.C
 *      $ link TCPIP$TCP_SERVER_SOCK
 *
 *
 * CONFIGURATION INSTRUCTIONS:
 *
 *    No special configuration required.
 *
 *
 * RUN INSTRUCTIONS:
 *
 *    To run this example program:
 *
 *    1) Start this server program as shown below:
 *
 *       $ run tcpip$tcp_server_sock
 *       Waiting for a client connection on port: m
 *
 *    2) After the server program blocks, start the client program,
 *       entering the server host as shown below:
 *
 *       $ run tcpip$tcp_client_sock
 *       Enter remote host:
 *
 *       Note: You can specify a server host by using either an IPv4
 *             address in dotted-decimal notation (e.g. 16.20.10.56)
 *             or a host domain name (e.g. serverhost.compaq.com).
 *
 *    3) The server program then displays client connection information
 *       and client data as shown below:
 *
 *       Accepted connection from host: a.b.c.d, port: n
 *       Data sent: Hello, world!
 *
 *    You can enter "ctrl/z" at any user prompt to terminate program
 *    execution.
 *
 */
/*
```

(continued on next page)

Example E–2 (Cont.) TCP Server (Sockets API)

```
 * INCLUDE FILES:
 */
#include <in.h>                    /* define internet related constants,   */
                                   /* functions, and structures            */
#include <inet.h>                  /* define network address info          */

#include <netdb.h>                 /* define network database library info */

#include <socket.h>                /* define BSD 4.x socket api            */
#include <stdio.h>                 /* define standard i/o functions        */
#include <stdlib.h>                /* define standard library functions    */
#include <string.h>                /* define string handling functions     */

#include <unixio.h>                /* define unix i/o                      */

/*
 * NAMED CONSTANTS:
 */

#define SERV_BACKLOG    1              /* server backlog                   */
#define SERV_PORTNUM    12345          /* server port number               */


/*
 * FORWARD REFERENCES:
 */

int  main( void );                     /* server main                      */
       /* Server Main */

/*
 * FUNCTIONAL DESCRIPTION:
 *
 *    This is the server's main-line code. It handles all the tasks of the
 *    server including: socket creation, accepting and/or rejecting client
 *    connections, writing client connection data,  and terminating client
 *    connections.
 *
 *    This example program implements a typical TCP IPv4 server using  the
 *    BSD socket API to handle network i/o operations as shown below:
 *
 *      1) To create a socket:
 *
 *         socket()
 *
 *      2) To set REUSEADDR socket option:
 *
 *         setsockopt()
 *
 *      3) To bind internet address and port number to a socket:
 *
 *         bind()
 *
 *      4) To set an active socket to a passive (listen) socket:
 *
 *         listen()
 *
 *      5) To accept a connection request:
 *
 *         accept()
 *
 *      6) To transfer data:
 *
 *         send()
```

(continued on next page)

Example E–2 (Cont.) TCP Server (Sockets API)

```
 *
 *      7) To shutdown a socket (both directions):
 *
 *         shutdown()
 *
 *      8) To close a socket:
 *
 *         close()
 *
 *    This function is invoked by the DCL "RUN" command (see below);  the
 *    function's  completion  status is interpreted by DCL and if needed,
 *    an error message is displayed.
 *
 * SYNOPSIS:
 *
 *    int main( void )
 *
 * FORMAL PARAMETERS:
 *
 *    ** None **
 *
 * IMPLICIT INPUTS:
 *
 *    ** None **
 *
 * IMPLICIT OUTPUTS:
 *
 *    ** None **
 *
 * FUNCTION VALUE:
 *
 *    completion status
 *
 * SIDE EFFECTS:
 *
 *    ** None **
 *
 */
int
main( void )
{
    int optval = 1;                     /* SO_REUSEADDR'S option value (on) */

    int conn_sockfd;                    /* connection socket descriptor     */
    int listen_sockfd;                  /* listen socket descriptor         */

    unsigned int client_addrlen;        /* returned length of client socket */
                                        /* address structure                */
    struct sockaddr_in client_addr;     /* client socket address structure  */
    struct sockaddr_in serv_addr;       /* server socket address structure  */

    char buf[] = "Hello, World!";       /* server data buffer               */

    /*
     * init client's socket address structure
     */

    memset( &client_addr, 0, sizeof(client_addr) );

    /*
     * init server's socket address structure
     */
```

(continued on next page)

Example E–2 (Cont.) TCP Server (Sockets API)

```
    memset( &serv_addr, 0, sizeof(serv_addr) );
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_port        = htons( SERV_PORTNUM );
    serv_addr.sin_addr.s_addr = INADDR_ANY;

    /*
     * create a listen socket
     */

    if ( (listen_sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }

    /*
     * bind server's internet address and port number to listen socket
     */

    if ( setsockopt(listen_sockfd,
                  SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval)) < 0 )
        {
        perror( "Failed to set socket option" );
        exit( EXIT_FAILURE );
        }

    if ( bind(listen_sockfd,
              (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0 )
        {
        perror( "Failed to bind socket" );
        exit( EXIT_FAILURE );
        }

    /*
     * set listen socket as a passive socket
     */

    if ( listen(listen_sockfd, SERV_BACKLOG) < 0 )
        {
        perror( "Failed to set socket passive" );
        exit( EXIT_FAILURE );
        }

    /*
     * accept connection from a client
     */

    printf( "Waiting for a client connection on port: %d\n",
            ntohs(serv_addr.sin_port)
          );

    client_addrlen = sizeof(client_addr);

    conn_sockfd = accept( listen_sockfd,
                          (struct sockaddr *) &client_addr,
                          &client_addrlen
                        );
    if ( conn_sockfd < 0 )
        {
        perror( "Failed to accept client connection" );
        exit( EXIT_FAILURE );
        }
```

(continued on next page)

Example E–2 (Cont.) TCP Server (Sockets API)

```
    printf( "Accepted connection from host: %s, port: %d\n",
            inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port)
          );
    /*
     * connection established with a client;
     * now attempt to write on this connection
     */

    if ( send(conn_sockfd, buf, sizeof(buf), 0) < 0 )
        {
        perror( "Failed to write data to client connection" );
        exit( EXIT_FAILURE );
        }

    printf( "Data sent: %s\n", buf );      /* output server's data buffer  */

    /*
     * shutdown connection socket (both directions)
     */

    if ( shutdown(conn_sockfd, 2) < 0 )
        {
        perror( "Failed to shutdown client connection" );
        exit( EXIT_FAILURE );
        }

    /*
     * close connection socket
     */

    if ( close(conn_sockfd) < 0 )
        {
        perror( "Failed to close socket" );
        exit( EXIT_FAILURE );
        }

    /*
     * close listen socket
     */

    if ( close(listen_sockfd) < 0 )
        {
        perror( "Failed to close socket" );
        exit( EXIT_FAILURE );
        }

    exit( EXIT_SUCCESS );
}
```

E.1.3 TCP Server Accepting a Connection from the Auxiliary Server

Example E–3 shows how a typical TCP/IP IPv4 server uses the BSD Version 4.*x* Sockets API to handle network I/O operations, and how the server accepts connections from the auxiliary server.

Example E–3 TCP Auxiliary Server (Sockets API)

```
#pragma module  tcpip$tcp_server_sock_auxs             \
                "V5.1-00"

/*
 * Copyright 2000 Compaq Computer Corporation
 *
 * COMPAQ Registered in U.S. Patent and Trademark Office.
 *
 * Confidential computer software.  Valid license from  Compaq
 * or  authorized sublicensor required for possession, use  or
 * copying.  Consistent with FAR 12.211 and 12.212, Commercial
 * Computer Software,   Computer Software Documentation,   and
 * Technical Data for Commercial Items  are  licensed  to  the
 * U.S. Government under vendor's standard commercial license.
 *
 * ++
 * FACILITY:
 *
 *    EXAMPLES
 *
 * ABSTRACT:
 *
 *    This is an example of a TCP/IP IPv4 server using  4.x BSD
 *    socket Application Programming Interface (API) to  handle
 *    network I/O operations.  In addition,  it  shows  how  to
 *    accept connections from the auxiliary server.
 *
 *    Refer to 'Build, Configuration, and Run Instructions' for
 *    details on how to build, configure, and run this program.
 *
 * ENVIRONMENT:
 *
 *    OpenVMS Alpha/VAX V7.1
 *    TCP/IP Services V5.0 or higher
 *
 * AUTHOR:
 *
 *    TCPIP Development Group,  CREATION DATE: 23-May-1989
 *
 * --
 */
```

(continued on next page)

Example E–3 (Cont.) TCP Auxiliary Server (Sockets API)

```
       /* Build, Configuration, and Run Instructions */
/*
 *  BUILD INSTRUCTIONS:
 *
 *    To build this example program use commands of the form,
 *
 *    using the DEC "C" compiler:
 *
 *      $ cc/prefix=all TCPIP$TCP_SERVER_SOCK_AUXS.C
 *      $ link TCPIP$TCP_SERVER_SOCK_AUXS
 *
 *    using the DEC "C++" compiler:
 *
 *      $ cxx/prefix=all/define=VMS TCPIP$TCP_SERVER_SOCK_AUXS.C
 *      $ link TCPIP$TCP_SERVER_SOCK_AUXS
 *
 *
 * CONFIGURATION INSTRUCTIONS:
 *
 *    To configure this example program:
 *
 *    1) Create a service run command procedure, named HELLO_RUN.COM, that
 *       contains the following lines:
 *
 *       $ define sys$output ddcu:[directory]hello_service.log
 *       $ define sys$error  ddcu:[directory]hello_service.log
 *       $ run ddcu:[directory]tcpip$tcp_server_sock_auxs.exe
 *
 *       where: ddcu:[directory] is the device and directory of where the
 *              hello service run command procedure file resides
 *
 *    2) Create a service database entry for the hello service as shown below:
 *
 *       $ tcpip set service hello -
 *       _$ /port=12345 -
 *       _$ /protocol=tcp -
 *       _$ /user=vms_user_account -
 *       _$ /process_name=hello_world -
 *       _$ /file=ddcu:[directory]hello_run.com
 *
 *    3) Enable the hello service to run as shown below:
 *
 *       $ tcpip enable service hello
 *
 *
 * RUN INSTRUCTIONS:
 *
 *    To run this example program:
 *
 *    1) Start the client program, entering the server host as shown below:
 *
 *       $ run tcpip$tcp_client_sock
 *       Enter remote host:
 *
 *       Note: You can specify a server host by using either an IPv4
 *             address in dotted-decimal notation (e.g. 16.20.10.56)
 *             or a host domain name (e.g. serverhost.compaq.com).
 *
 *    2) The auxiliary server receives the hello service request, creates a
```

(continued on next page)

Example E–3 (Cont.) TCP Auxiliary Server (Sockets API)

```
 *       process, then executes the commands in hello_run.com to  run  this
 *       server program.  This server program then logs  client  connection
 *       information and client data to the service log before replying  to
 *       the client host with a message of "Hello, world!".
 *
 */

/*
 * INCLUDE FILES:
 */

#include <in.h>                     /* define internet related constants,   */
                                    /* functions, and structures            */
#include <inet.h>                   /* define network address info          */

#include <socket.h>                 /* define BSD 4.x socket api            */
#include <stdio.h>                  /* define standard i/o functions        */
#include <stdlib.h>                 /* define standard library functions    */
#include <string.h>                 /* define string handling functions     */

#include <tcpip$inetdef.h>          /* define tcp/ip network constants,     */
                                    /* structures, and functions            */

#include <unixio.h>                 /* define unix i/o                      */

/*
 * FORWARD REFERENCES:
 */

int  main( void );                      /* server main                      */
       /* Server Main */

/*
 * FUNCTIONAL DESCRIPTION:
 *
 *    This is the server's main-line code. It handles all the tasks of the
 *    server including:  socket creation,  writing client connection data,
 *    and terminating client connections.
 *
 *    This example program implements a typical TCP IPv4 server using  the
 *    BSD socket  API  to  handle network i/o operations.  In addition, it
 *    uses the auxiliary server to accept client connections.
 *
 *      1) To create a socket:
 *
 *         socket()
 *
 *      2) To transfer data:
 *
 *         send()
 *
 *      3) To close a socket:
 *
 *         close()
 *
 *    This function is invoked by the DCL "RUN" command (see below);  the
 *    function's  completion  status is interpreted by DCL and if needed,
 *    an error message is displayed.
 *
 * SYNOPSIS:
 *
 *    int main( void )
 *
 * FORMAL PARAMETERS:
```

(continued on next page)

Example E–3 (Cont.) TCP Auxiliary Server (Sockets API)

```
 *
 *    ** None **
 *
 * IMPLICIT INPUTS:
 *
 *    ** None **
 *
 * IMPLICIT OUTPUTS:
 *
 *    ** None **
 *
 * FUNCTION VALUE:
 *
 *    completion status
 *
 * SIDE EFFECTS:
 *
 *    ** None **
 *
 */

int
main( void )
{
    int sockfd;                         /* socket descriptor                */

    unsigned int client_addrlen;        /* returned length of client socket */
                                        /* address structure                */
    struct sockaddr_in client_addr;     /* client socket address structure  */

    char buf[] = "Hello, world!";       /* server data buffer               */

    /*
     * init client's socket address structure
     */

    memset( &client_addr, 0, sizeof(client_addr) );

    /*
     * create socket
     */

    if ( (sockfd = socket(TCPIP$C_AUXS, SOCK_STREAM, 0)) < 0 )
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }

    /*
     * log this client connection
     */

    client_addrlen = sizeof(client_addr);

    if ( getpeername(sockfd,
                     (struct sockaddr *) &client_addr, &client_addrlen) < 0 )
        {
        perror( "Failed to accept client connection" );
        exit( EXIT_FAILURE );
        }

    printf( "Accepted connection from host: %s, port: %d\n",
            inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port)
          );
```

(continued on next page)

Example E–3 (Cont.) TCP Auxiliary Server (Sockets API)

```
    /*
     * connection established with a client;
     * now attempt to write on this connection
     */

    if ( send(sockfd, buf, sizeof(buf), 0) < 0 )
        {
        perror( "Failed to write data to client connection" );
        exit( EXIT_FAILURE );
        }

    printf( "Data sent: %s\n", buf );      /* output server's data buffer  */

    /*
     * close socket
     */

    if ( close(sockfd) < 0 )
        {
        perror( "Failed to close socket" );
        exit( EXIT_FAILURE );
        }

    exit( EXIT_SUCCESS );
}
```

E.2 TCP Client/Server Examples (System Services)

This section contains the following examples:

- A TCP/IP IPv4 client using $QIO system services to handle network I/O operations.
- A TCP/IP IPv4 server using $QIO system services to handle network I/O operations.
- A TCP/IP IPv4 server using $QIO system services to handle network I/O operations and how the server accepts connections from the auxiliary server.

E.2.1 TCP Client

Example E–4 shows how a typical TCP IPv4 client uses $QIO system services to handle the tasks of creating a socket, initiating server connections, reading service connection data, and then terminating the server connections.

Example E–4 TCP Client (System Services)

```
#pragma module  tcpip$tcp_client_qio                    \
                "V5.1-00"
/*
 * Copyright 2000 Compaq Computer Corporation
 *
 * COMPAQ Registered in U.S. Patent and Trademark Office.
 *
 * Confidential computer software.  Valid license from  Compaq
 * or  authorized sublicensor required for possession, use  or
 * copying.  Consistent with FAR 12.211 and 12.212, Commercial
 * Computer Software,   Computer Software Documentation,   and
 * Technical Data for Commercial Items  are  licensed  to  the
 * U.S. Government under vendor's standard commercial license.
 *
 * ++
 * FACILITY:
 *
 *    EXAMPLES
 *
 * ABSTRACT:
 *
 *    This is an example of a TCP/IP IPv4 client using  OpenVMS
 *    QIO system services to handle network I/O operations.
 *
 *    Refer to 'Build, Configuration, and Run Instructions' for
 *    details on how to build, configure, and run this program.
 *
 * ENVIRONMENT:
 *
 *    OpenVMS Alpha/VAX V7.1
 *    TCP/IP Services V5.0 or higher
 *
 * AUTHOR:
 *
 *    TCPIP Development Group,  CREATION DATE: 23-May-1989
 *
 * --
 */
```

(continued on next page)

Example E–4 (Cont.) TCP Client (System Services)

```
       /* Build, Configuration, and Run Instructions */

/*
 * BUILD INSTRUCTIONS:
 *
 *    To build this example program use commands of the form,
 *
 *    using the DEC "C" compiler:
 *
 *      $ cc/prefix=all TCPIP$TCP_CLIENT_QIO.C
 *      $ link TCPIP$TCP_CLIENT_QIO
 *
 *    using the DEC "C++" compiler:
 *
 *      $ cxx/prefix=all/define=VMS TCPIP$TCP_CLIENT_QIO.C
 *      $ link TCPIP$TCP_CLIENT_QIO
 *
 *
 * CONFIGURATION INSTRUCTIONS:
 *
 *    No special configuration required.
 *
 *
 * RUN INSTRUCTIONS:
 *
 *    To run this example program:
 *
 *    1) Start the client's server program as shown below:
 *
 *       $ run tcpip$tcp_server_qio
 *       Waiting for a client connection on port: m
 *
 *    2) After the server program blocks, start this client program,
 *       entering the server host as shown below:
 *
 *       $ run tcpip$tcp_client_qio
 *       Enter remote host:
 *
 *       Note: You can specify a server host by using either an IPv4
 *             address in dotted-decimal notation (e.g. 16.20.10.56)
 *             or a host domain name (e.g. serverhost.compaq.com).
 *
 *    3) The client program then displays server connection information
 *       and server data as show below:
 *
 *       Initiated connection to host: a.b.c.d, port: n
 *       Data received: Hello, world!
 *
 *    You can enter "ctrl/z" at any user prompt to terminate program
 *    execution.
 *
 */

/*
 * INCLUDE FILES:
 */

#include <descrip.h>               /* define OpenVMS descriptors          */

#include <efndef.h>                /* define 'EFN$C_ENF' event flag       */
```

(continued on next page)

Example E–4 (Cont.) TCP Client (System Services)

```
#include <in.h>                     /* define internet related constants,   */
                                    /* functions, and structures            */
#include <inet.h>                   /* define network address info          */
#include <iodef.h>                  /* define i/o function codes            */

#include <netdb.h>                  /* define network database library info */

#include <ssdef.h>                  /* define system service status codes   */
#include <starlet.h>                /* define system service calls          */
#include <stdio.h>                  /* define standard i/o functions        */
#include <stdlib.h>                 /* define standard library functions    */
#include <string.h>                 /* define string handling functions     */
#include <stsdef.h>                 /* define condition value fields        */

#include <tcpip$inetdef.h>          /* define tcp/ip network constants,     */
                                    /* structures, and functions            */

/*
 * NAMED CONSTANTS:
 */

#define BUFSZ           1024            /* user input buffer size           */
#define SERV_PORTNUM    12345           /* server port number               */

/*
 * STRUCTURE DEFINITIONS:
 */

struct iosb
    {                               /* i/o status block                 */
    unsigned short status;          /* i/o completion status            */
    unsigned short bytcnt;          /* bytes transferred if read/write  */
    void *details;                  /* address of buffer or parameter   */
    };

struct itemlst_2
    {                               /* item-list 2 descriptor/element   */
    unsigned short length;          /* length                           */
    unsigned short type;            /* parameter type                   */
    void *address;                  /* address of item list             */
    };

struct sockchar
    {                               /* socket characteristics buffer    */
    unsigned short prot;            /* protocol                         */
    unsigned char type;             /* type                             */
    unsigned char af;               /* address format                   */
    };

/*
 * FORWARD REFERENCES:
 */

int  main( void );                  /* client main                      */
void get_serv_addr( void * );       /* get server host address          */
```

(continued on next page)

Example E–4 (Cont.) TCP Client (System Services)

```
       /* Client Main */

/*
 * FUNCTIONAL DESCRIPTION:
 *
 *    This is the client's main-line code. It handles all the tasks of the
 *    client including:  socket creation,  initiating server  connections,
 *    reading server connection data, and terminating server connections.
 *
 *    This example program implements a typical TCP IPv4 client using  QIO
 *    system services to handle network i/o operations as shown below:
 *
 *      1) To create a socket:
 *
 *         sys$assign() and sys$qiow(IO$_SETMODE)
 *
 *      2) To initiate a connection:
 *
 *         sys$qiow(IO$_ACCESS)
 *
 *      3) To transfer data:
 *
 *         sys$qiow(IO$_READVBLK)
 *
 *      4) To shutdown a socket:
 *
 *         sys$qiow(IO$_DEACCESS|IO$M_SHUTDOWN)
 *
 *      5) To close and delete a socket:
 *
 *         sys$qiow(IO$_DEACCESS) and sys$dassgn()
 *
 *    This function is invoked by the DCL "RUN" command (see below);  the
 *    function's  completion  status is interpreted by DCL and if needed,
 *    an error message is displayed.
 *
 * SYNOPSIS:
 *
 *    int main( void )
 *
 * FORMAL PARAMETERS:
 *
 *    ** None **
 *
 * IMPLICIT INPUTS:
 *
 *    ** None **
 *
 * IMPLICIT OUTPUTS:
 *
 *    ** None **
 *
 * FUNCTION VALUE:
 *
 *    completion status
 *
 * SIDE EFFECTS:
 *
 *    ** None **
 *
 */
```

(continued on next page)

Example E–4 (Cont.) TCP Client (System Services)

```
int
main( void )
{
    struct iosb iosb;                   /* i/o status block                */

    unsigned int status;                /* system service return status    */

    char buf[512];                      /* client data buffer              */
    int  buflen = sizeof( buf );        /* length of client data buffer    */

    unsigned short conn_channel;        /* connect inet device i/o channel */
    struct sockchar conn_sockchar;      /* connect socket char buffer      */

    struct sockaddr_in serv_addr;       /* server socket address structure */
    struct itemlst_2 serv_itemlst;      /* server item-list 2 descriptor   */

    $DESCRIPTOR( inet_device,           /* string descriptor with logical  */
                 "TCPIP$DEVICE" );      /* name of internet pseudodevice   */

    /*
     * init connection socket characteristics buffer
     */

    conn_sockchar.prot = TCPIP$C_TCP;
    conn_sockchar.type = TCPIP$C_STREAM;
    conn_sockchar.af   = TCPIP$C_AF_INET;

    /*
     * init server's item-list descriptor
     */

    memset( &serv_itemlst, 0, sizeof(serv_itemlst) );
    serv_itemlst.length  = sizeof( serv_addr );
    serv_itemlst.address = &serv_addr;

    /*
     * init server's socket address structure
     */

    memset( &serv_addr, 0, sizeof(serv_addr) );
    serv_addr.sin_family = TCPIP$C_AF_INET;
    serv_addr.sin_port   = htons( SERV_PORTNUM );
    get_serv_addr( &serv_addr.sin_addr );

    /*
     * assign device socket
     */

    status = sys$assign( &inet_device,      /* device name                 */
                         &conn_channel,     /* i/o channel                 */
                         0,                 /* access mode                 */
                         0                  /* not used                    */
                       );

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to assign i/o channel to TCPIP device\n" );
        exit( status );
        }

    /*
     * create connection socket
     */
```

(continued on next page)

Example E–4 (Cont.) TCP Client (System Services)

```
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       conn_channel,        /* i/o channel                  */
                       IO$_SETMODE,         /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       &conn_sockchar,      /* p1 - socket char buffer      */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to create socket\n" );
        exit( status );
        }

    /*
     * connect to specified host and port number
     */

    printf( "Initiated connection to host: %s, port: %d\n",
            inet_ntoa(serv_addr.sin_addr), ntohs(serv_addr.sin_port)
          );

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       conn_channel,        /* i/o channel                  */
                       IO$_ACCESS,          /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       &serv_itemlst,       /* p3 - remote socket name      */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to connect to server\n" );
        exit( status );
        }

    /*
     * connection established with a server;
     * now attempt to read on this connection
     */
```

(continued on next page)

Example E–4 (Cont.) TCP Client (System Services)

```
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       conn_channel,        /* i/o channel                  */
                       IO$_READVBLK,        /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       buf,                 /* p1 - buffer address          */
                       buflen,              /* p2 - buffer length           */
                       0,                   /* p3                           */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to read data from server connection\n" );
        exit( status );
        }

    printf( "Data received: %s\n", buf );   /* output client's data buffer  */

    /*
     * shutdown connection socket
     */

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       conn_channel,        /* i/o channel                  */
                       IO$_DEACCESS|IO$M_SHUTDOWN,
                                            /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       TCPIP$C_DSC_ALL,     /* p4 - discard all packets     */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to shutdown server connection\n" );
        exit( status );
        }

    /*
     * close connection socket
     */
```

(continued on next page)

Example E–4 (Cont.) TCP Client (System Services)

```
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       conn_channel,        /* i/o channel                  */
                       IO$_DEACCESS,        /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to close socket\n" );
        exit( status );
        }

    /*
     * deassign device socket
     */

    status = sys$dassgn( conn_channel );

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to deassign i/o channel to TCPIP device\n" );
        exit( status );
        }

    exit( EXIT_SUCCESS );
}
        /* Get Server Host Address */
```

(continued on next page)

Example E–4 (Cont.) TCP Client (System Services)

```
/*
 * FUNCTIONAL DESCRIPTION:
 *
 *    This function gets the server host's address from the user and then
 *    stores it in the server's socket address structure.  Note that  the
 *    user can specify a server host by using either an IPv4  address  in
 *    dotted-decimal notation (e.g. 16.20.10.126) or a host  domain  name
 *    (e.g. serverhost.compaq.com).
 *
 *    Enter "ctrl/z" to terminate program execution.
 *
 * SYNOPSIS:
 *
 *    void get_serv_addr( void *addrptr )
 *
 * FORMAL PARAMETERS:
 *
 *    addrptr  -  pointer to socket address structure's 'sin_addr' field
 *                to store the specified network address
 *
 * IMPLICIT INPUTS:
 *
 *    ** None **
 *
 * IMPLICIT OUTPUTS:
 *
 *    ** None **
 *
 * FUNCTION VALUE:
 *
 *    ** None **
 *
 * SIDE EFFECTS:
 *
 *    Program execution is terminated if unable to read user's input.
 *
 */

void
get_serv_addr( void *addrptr )
{
    char buf[BUFSZ];
    struct in_addr val;
    struct hostent *host;

    while ( TRUE )
        {
        printf( "Enter remote host: " );

        if ( fgets(buf, sizeof(buf), stdin) == NULL )
            {
            printf( "Failed to read user input\n" );
            exit( EXIT_FAILURE );
            }

        buf[strlen(buf)-1] = 0;

        val.s_addr = inet_addr( buf );
```

(continued on next page)

Example E–4 (Cont.) TCP Client (System Services)

```
        if ( val.s_addr != INADDR_NONE )
            {
            memcpy( addrptr, &val, sizeof(struct in_addr) );
            break;
            }
        if ( (host = gethostbyname(buf)) )
            {
            memcpy( addrptr, host->h_addr, sizeof(struct in_addr) );
            break;
            }
        }
}
```

E.2.2 TCP Server

Example E–5 shows how a typical TCP IPv4 server uses $QIO system services to handle the tasks of creating a socket, accepting or rejecting client connections, writing client connection data, and then terminating client connections.

Example E–5 TCP Server (System Services)

```
#pragma module  tcpip$tcp_server_qio                    \
                "V5.1-00"

/*
 * Copyright 2000 Compaq Computer Corporation
 *
 * COMPAQ Registered in U.S. Patent and Trademark Office.
 *
 * Confidential computer software.  Valid license from  Compaq
 * or  authorized sublicensor required for possession, use  or
 * copying.  Consistent with FAR 12.211 and 12.212, Commercial
 * Computer Software,   Computer Software Documentation,   and
 * Technical Data for Commercial Items  are  licensed  to  the
 * U.S. Government under vendor's standard commercial license.
 *
 * ++
 * FACILITY:
 *
 *    EXAMPLES
 *
 * ABSTRACT:
 *
 *    This is an example of a TCP/IP IPv4 server using  OpenVMS
 *    QIO system services to handle network I/O operations.
 *
 *    Refer to 'Build, Configuration, and Run Instructions' for
 *    details on how to build, configure, and run this program.
 *
 * ENVIRONMENT:
 *
 *    OpenVMS Alpha/VAX V7.1
 *    TCP/IP Services V5.0 or higher
```

(continued on next page)

Example E–5 (Cont.) TCP Server (System Services)

```
 *
 * AUTHOR:
 *
 *    TCPIP Development Group,  CREATION DATE: 23-May-1989
 *
 * --
 */
       /* Build, Configuration, and Run Instructions */

/*
 *  BUILD INSTRUCTIONS:
 *
 *    To build this example program use commands of the form,
 *
 *    using the DEC "C" compiler:
 *
 *      $ cc/prefix=all TCPIP$TCP_SERVER_QIO.C
 *      $ link TCPIP$TCP_SERVER_QIO
 *
 *    using the DEC "C++" compiler:
 *
 *      $ cxx/prefix=all/define=VMS TCPIP$TCP_SERVER_QIO.C
 *      $ link TCPIP$TCP_SERVER_QIO
 *
 *
 * CONFIGURATION INSTRUCTIONS:
 *
 *    No special configuration required.
 *
 *
 * RUN INSTRUCTIONS:
 *
 *    To run this example program:
 *
 *    1) Start this server program as shown below:
 *
 *       $ run tcpip$tcp_server_qio
 *       Waiting for a client connection on port: m
 *
 *    2) After the server program blocks, start the client program,
 *       entering the server host as shown below:
 *
 *       $ run tcpip$tcp_client_qio
 *       Enter remote host:
 *
 *       Note: You can specify a server host by using either an IPv4
 *             address in dotted-decimal notation (e.g. 16.20.10.56)
 *             or a host domain name (e.g. serverhost.compaq.com).
 *
 *    3) The server program then displays client connection information
 *       and client data as shown below:
 *
 *       Accepted connection from host: a.b.c.d, port: n
 *       Data sent: Hello, world!
 *
 *    You can enter "ctrl/z" at any user prompt to terminate program
 *    execution.
 *
 */
```

(continued on next page)

Example E–5 (Cont.) TCP Server (System Services)

```
/*
 * INCLUDE FILES:
 */

#include <descrip.h>                /* define OpenVMS descriptors          */

#include <efndef.h>                 /* define 'EFN$C_ENF' event flag       */

#include <in.h>                     /* define internet related constants,  */
                                    /* functions, and structures           */
#include <inet.h>                   /* define network address info         */
#include <iodef.h>                  /* define i/o function codes           */

#include <netdb.h>                  /* define network database library info */

#include <ssdef.h>                  /* define system service status codes  */
#include <starlet.h>                /* define system service calls         */
#include <stdio.h>                  /* define standard i/o functions       */
#include <stdlib.h>                 /* define standard library functions   */
#include <string.h>                 /* define string handling functions    */
#include <stsdef.h>                 /* define condition value fields       */

#include <tcpip$inetdef.h>          /* define tcp/ip network constants,    */
                                    /* structures, and functions           */


/*
 * NAMED CONSTANTS:
 */

#define SERV_BACKLOG    1               /* server backlog                  */
#define SERV_PORTNUM    12345           /* server port number              */

/*
 * STRUCTURE DEFINITIONS:
 */

struct iosb
    {                                   /* i/o status block                */
    unsigned short status;              /* i/o completion status           */
    unsigned short bytcnt;              /* bytes transferred if read/write */
    void *details;                      /* address of buffer or parameter  */
    };

struct itemlst_2
    {                                   /* item-list 2 descriptor/element  */
    unsigned short length;              /* length                          */
    unsigned short type;                /* parameter type                  */
    void *address;                      /* address of item list            */
    };

struct itemlst_3
    {                                   /* item-list 3 descriptor/element  */
    unsigned short length;              /* length                          */
    unsigned short type;                /* parameter type                  */
    void *address;                      /* address of item list            */
    unsigned int *retlen;               /* address of returned length      */
    };

struct sockchar
    {                                   /* socket characteristics buffer   */
    unsigned short prot;                /* protocol                        */
    unsigned char type;                 /* type                            */
    unsigned char af;                   /* address format                  */
    };
```

(continued on next page)

Example E–5 (Cont.) TCP Server (System Services)

```
/*
 * FORWARD REFERENCES:
 */

int  main( void );                      /* server main                      */
       /* Server Main */

/*
 * FUNCTIONAL DESCRIPTION:
 *
 *    This is the server's main-line code. It handles all the tasks of the
 *    server including: socket creation, accepting and/or rejecting client
 *    connections, writing client connection data,  and terminating client
 *    connections.
 *
 *    This example program implements a typical TCP IPv4 server using  QIO
 *    system services to handle network i/o operations as shown below:
 *
 *      1) To create a socket and set REUSEADDR option:
 *
 *         sys$assign() and sys$qiow(IO$_SETMODE)
 *
 *      2) To bind internet address and port number to a socket:
 *
 *         sys$qiow(IO$_SETMODE)
 *
 *      3) To accept a connection request:
 *
 *         sys$qiow(IO$_ACCESS|IO$M_ACCEPT)
 *
 *      4) To transfer data:
 *
 *         sys$qiow(IO$_WRITEVBLK)
 *
 *      5) To shutdown a socket:
 *
 *         sys$qiow(IO$_DEACCESS|IO$M_SHUTDOWN)
 *
 *      6) To close and delete a socket:
 *
 *         sys$qiow(IO$_DEACCESS) and sys$dassgn()
 *
 *    This function is invoked by the DCL "RUN" command (see below);  the
 *    function's  completion  status is interpreted by DCL and if needed,
 *    an error message is displayed.
 *
```

(continued on next page)

Example E–5 (Cont.) TCP Server (System Services)

```
 * SYNOPSIS:
 *
 *    int main( void )
 *
 * FORMAL PARAMETERS:
 *
 *    ** None **
 *
 * IMPLICIT INPUTS:
 *
 *    ** None **
 *
 * IMPLICIT OUTPUTS:
 *
 *    ** None **
 *
 * FUNCTION VALUE:
 *
 *    completion status
 *
 * SIDE EFFECTS:
 *
 *    ** None **
 *
 */
int
main( void )
{
    int one = 1;                        /* reuseaddr option value           */

    struct iosb iosb;                   /* i/o status block                 */

    unsigned int status;                /* system service return status     */

    unsigned short conn_channel;        /* connect inet device i/o channel  */
    unsigned short listen_channel;      /* listen inet device i/o channel   */

    struct sockchar listen_sockchar;    /* listen socket char buffer        */

    unsigned int client_retlen;         /* returned length of client socket */
                                        /* address structure                */
    struct sockaddr_in client_addr;     /* client socket address structure  */
    struct itemlst_3 client_itemlst;    /* client item-list 3 descriptor    */

    struct sockaddr_in serv_addr;       /* server socket address structure  */
    struct itemlst_2 serv_itemlst;      /* server item-list 2 descriptor    */

    struct itemlst_2 sockopt_itemlst;   /* sockopt item-list 2 descriptor   */
    struct itemlst_2 reuseaddr_itemlst; /* reuseaddr item-list 2 element    */

    char buf[] = "Hello, World!";       /* server data buffer               */
    int  buflen = sizeof( buf );        /* length of server data buffer     */

    $DESCRIPTOR( inet_device,           /* string descriptor with logical   */
                 "TCPIP$DEVICE" );      /* name of internet pseudodevice    */

    /*
     * init listen socket characteristics buffer
     */

    listen_sockchar.prot = TCPIP$C_TCP;
    listen_sockchar.type = TCPIP$C_STREAM;
    listen_sockchar.af   = TCPIP$C_AF_INET;
```

(continued on next page)

Example E–5 (Cont.) TCP Server (System Services)

```
    /*
     * init reuseaddr's item-list element
     */

    reuseaddr_itemlst.length  = sizeof( one );
    reuseaddr_itemlst.type    = TCPIP$C_REUSEADDR;
    reuseaddr_itemlst.address = &one;

    /*
     * init sockopt's item-list descriptor
     */

    sockopt_itemlst.length  = sizeof( reuseaddr_itemlst );
    sockopt_itemlst.type    = TCPIP$C_SOCKOPT;
    sockopt_itemlst.address = &reuseaddr_itemlst;

    /*
     * init client's item-list descriptor
     */

    memset( &client_itemlst, 0, sizeof(client_itemlst) );
    client_itemlst.length  = sizeof( client_addr );
    client_itemlst.address = &client_addr;
    client_itemlst.retlen  = &client_retlen;

    /*
     * init client's socket address structure
     */

    memset( &client_addr, 0, sizeof(client_addr) );

    /*
     * init server's item-list descriptor
     */

    serv_itemlst.length  = sizeof( serv_addr );
    serv_itemlst.type    = TCPIP$C_SOCK_NAME;
    serv_itemlst.address = &serv_addr;

    /*
     * init server's socket address structure
     */

    memset( &serv_addr, 0, sizeof(serv_addr) );
    serv_addr.sin_family      = TCPIP$C_AF_INET;
    serv_addr.sin_port        = htons( SERV_PORTNUM );
    serv_addr.sin_addr.s_addr = TCPIP$C_INADDR_ANY;

    /*
     * assign device sockets
     */

    status = sys$assign( &inet_device,      /* device name                */
                         &listen_channel,   /* i/o channel                */
                         0,                 /* access mode                */
                         0                  /* not used                   */
                       );

    if ( (status & STS$M_SUCCESS) )
        status = sys$assign( &inet_device,  /* device name                */
                             &conn_channel, /* i/o channel                */
                             0,             /* access mode                */
                             0              /* not used                   */
                       );
```

(continued on next page)

Example E–5 (Cont.) TCP Server (System Services)

```
    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to assign i/o channel(s) to TCPIP device\n" );
        exit( status );
        }

    /*
     * create a listen socket
     */

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       listen_channel,      /* i/o channel                  */
                       IO$_SETMODE,         /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       &listen_sockchar,    /* p1 - socket char buffer      */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       0,                   /* p4                           */
                       &sockopt_itemlst,    /* p5 - socket options          */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to create socket\n" );
        exit( status );
        }

    /*
     * bind server's internet address and port number to
     * listen socket; set socket as a passive socket
     */

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       listen_channel,      /* i/o channel                  */
                       IO$_SETMODE,         /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       &serv_itemlst,       /* p3 - local socket name       */
                       SERV_BACKLOG,        /* p4 - connection backlog      */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;
```

(continued on next page)

Example E–5 (Cont.) TCP Server (System Services)

```
    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to bind socket\n" );
        exit( status );
        }

    /*
     * accept a connection from a client
     */

    printf( "Waiting for a client connection on port: %d\n",
            ntohs(serv_addr.sin_port)
          );

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       listen_channel,      /* i/o channel                  */
                       IO$_ACCESS|IO$M_ACCEPT,
                                            /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       &client_itemlst,     /* p3 - remote socket name      */
                       &conn_channel,       /* p4 - i/o channel for new     */
                                            /*      connection              */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to accept client connection\n" );
        exit( status );
        }

    printf( "Accepted connection from host: %s, port: %d\n",
            inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port)
          );

    /*
     * connection established with a client;
     * now attempt to write on this connection
     */

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       conn_channel,        /* i/o channel                  */
                       IO$_WRITEVBLK,       /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       buf,                 /* p1 - buffer address          */
                       buflen,              /* p2 - buffer length           */
                       0,                   /* p3                           */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;
```

(continued on next page)

Example E–5 (Cont.) TCP Server (System Services)

```
    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to write data to client connection\n" );
        exit( status );
        }

    printf( "Data sent: %s\n", buf );       /* output server's data buffer  */

    /*
     * shutdown connection socket
     */

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       conn_channel,        /* i/o channel                  */
                       IO$_DEACCESS|IO$M_SHUTDOWN,
                                            /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       TCPIP$C_DSC_ALL,     /* p4 - discard all packets     */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to shutdown client connection\n" );
        exit( status );
        }

    /*
     * close connection socket
     */

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       conn_channel,        /* i/o channel                  */
                       IO$_DEACCESS,        /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to close socket\n" );
        exit( status );
        }
```

(continued on next page)

Example E–5 (Cont.) TCP Server (System Services)

```
    /*
     * close listen socket
     */

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       listen_channel,      /* i/o channel                  */
                       IO$_DEACCESS,        /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to close socket\n" );
        exit( status );
        }

    /*
     * deassign all device sockets
     */

    status = sys$dassgn( conn_channel );

    if ( (status & STS$M_SUCCESS) )
        status = sys$dassgn( listen_channel );

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to deassign i/o channel(s) to TCPIP device\n" );
        exit( status );
        }

    exit( EXIT_SUCCESS );
}
```

E.2.3 TCP Server Accepting a Connection from the Auxiliary Server

Example E–6 shows how a typical TCP/IP IPv4 server uses $QIO system services to handle network I/O operations and the server accepts connections from the auxiliary server.

Example E–6 TCP Auxiliary Server (System Services)

```
#pragma module  tcpip$tcp_server_qio_auxs              \
                "V5.1-00"

/*
 * Copyright 2000 Compaq Computer Corporation
 *
 * COMPAQ Registered in U.S. Patent and Trademark Office.
 *
 * Confidential computer software.  Valid license from  Compaq
 * or  authorized sublicensor required for possession, use  or
 * copying.  Consistent with FAR 12.211 and 12.212, Commercial
 * Computer Software,   Computer Software Documentation,   and
 * Technical Data for Commercial Items  are  licensed  to  the
 * U.S. Government under vendor's standard commercial license.
 *
 * ++
 * FACILITY:
 *
 *    EXAMPLES
 *
 * ABSTRACT:
 *
 *    This is an example of a TCP/IP IPv4 server using  OpenVMS
 *    QIO system services to handle network I/O operations.  In
 *    addition, it  shows  how  to  accept connections from the
 *    auxiliary server.
 *
 *    Refer to 'Build, Configuration, and Run Instructions' for
 *    details on how to build, configure, and run this program.
 *
 * ENVIRONMENT:
 *
 *    OpenVMS Alpha/VAX V7.1
 *    TCP/IP Services V5.0 or higher
 *
 * AUTHOR:
 *
 *    TCPIP Development Group,  CREATION DATE: 23-May-1989
 *
 * --
 */
```

(continued on next page)

Example E–6 (Cont.) TCP Auxiliary Server (System Services)

```
       /* Build, Configuration, and Run Instructions */

/*
 *  BUILD INSTRUCTIONS:
 *
 *    To build this example program use commands of the form,
 *
 *    using the DEC "C" compiler:
 *
 *      $ cc/prefix=all TCPIP$TCP_SERVER_QIO_AUXS.C
 *      $ link TCPIP$TCP_SERVER_QIO_AUXS
 *
 *    using the DEC "C++" compiler:
 *
 *      $ cxx/prefix=all/define=VMS TCPIP$TCP_SERVER_QIO_AUXS.C
 *      $ link TCPIP$TCP_SERVER_QIO_AUXS
 *
 *
 * CONFIGURATION INSTRUCTIONS:
 *
 *    To configure this example program:
 *
 *    1) Create a service run command procedure, named HELLO_RUN.COM, that
 *       contains the following lines:
 *
 *       $ define sys$output ddcu:[directory]hello_service.log
 *       $ define sys$error  ddcu:[directory]hello_service.log
 *       $ run ddcu:[directory]tcpip$tcp_server_qio_auxs.exe
 *
 *       where: ddcu:[directory] is the device and directory of where the
 *              hello service run command procedure file resides
 *
 *    2) Create a service database entry for the hello service as shown below:
 *
 *       $ tcpip set service hello -
 *       _$ /port=12345 -
 *       _$ /protocol=tcp -
 *       _$ /user=vms_user_account -
 *       _$ /process_name=hello_world -
 *       _$ /file=ddcu:[directory]hello_run.com
 *
 *    3) Enable the hello service to run as shown below:
 *
 *       $ tcpip enable service hello
 *
 *
 * RUN INSTRUCTIONS:
 *
 *    To run this example program:
 *
 *    1) Start the client program, entering the server host as shown below:
 *
 *       $ run tcpip$tcp_client_sock
 *       Enter remote host:
 *
 *       Note: You can specify a server host by using either an IPv4
 *             address in dotted-decimal notation (e.g. 16.20.10.56)
 *             or a host domain name (e.g. serverhost.compaq.com).
 *
 *    2) The auxiliary server receives the hello service request, creates a
 *       process, then executes the commands in hello_run.com to  run  this
```

(continued on next page)

Example E–6 (Cont.) TCP Auxiliary Server (System Services)

```
 *       server program.  This server program then logs  client  connection
 *       information and client data to the service log before replying  to
 *       the client host with a message of "Hello, world!".
 *
 */

/*
 * INCLUDE FILES:
 */

#include <descrip.h>                /* define OpenVMS descriptors          */

#include <efndef.h>                 /* define 'EFN$C_ENF' event flag       */

#include <in.h>                     /* define internet related constants,  */
                                    /* functions, and structures           */
#include <inet.h>                   /* define network address info         */
#include <iodef.h>                  /* define i/o function codes           */

#include <ssdef.h>                  /* define system service status codes  */
#include <starlet.h>                /* define system service calls         */
#include <stdio.h>                  /* define standard i/o functions       */
#include <stdlib.h>                 /* define standard library functions   */
#include <string.h>                 /* define string handling functions    */
#include <stsdef.h>                 /* define condition value fields       */

#include <tcpip$inetdef.h>          /* define tcp/ip network constants,    */
                                    /* structures, and functions           */


 * STRUCTURE DEFINITIONS:
 */

struct iosb
    {                                   /* i/o status block                */
    unsigned short status;              /* i/o completion status           */
    unsigned short bytcnt;              /* bytes transferred if read/write */
    void *details;                      /* address of buffer or parameter  */
    };

struct itemlst_2
    {                                   /* item-list 2 descriptor/element  */
    unsigned short length;              /* length                          */
    unsigned short type;                /* parameter type                  */
    void *address;                      /* address of item list            */
    };

struct itemlst_3
    {                                   /* item-list 3 descriptor/element  */
    unsigned short length;              /* length                          */
    unsigned short type;                /* parameter type                  */
    void *address;                      /* address of item list            */
    unsigned int *retlen;               /* address of returned length      */
    };

struct sockchar
    {                                   /* socket characteristics buffer   */
    unsigned short prot;                /* protocol                        */
    unsigned char type;                 /* type                            */
    unsigned char af;                   /* address format                  */
    };

/*
 * FORWARD REFERENCES:
 */
```

(continued on next page)

Example E–6 (Cont.) TCP Auxiliary Server (System Services)

```
int  main( void );                      /* server main                      */

        /* Server Main */

/*
 * FUNCTIONAL DESCRIPTION:
 *
 *    This is the server's main-line code. It handles all the tasks of the
 *    server including:  socket creation,  writing client connection data,
 *    and terminating client connections.
 *
 *    This example program implements a typical TCP IPv4 server using  QIO
 *    system  services to handle network i/o operations.  In addition,  it
 *    uses the auxiliary server to accept client connections.
 *
 *      1) To create a socket:
 *
 *         sys$assign() and sys$qiow(IO$_SETMODE)
 *
 *      2) To transfer data:
 *
 *         sys$qiow(IO$_WRITEVBLK)
 *
 *      3) To close and delete a socket:
 *
 *         sys$qiow(IO$_DEACCESS) and sys$dassgn()
 *
 *    This function is invoked by the DCL "RUN" command (see below);  the
 *    function's  completion  status is interpreted by DCL and if needed,
 *    an error message is displayed.
 *
 * SYNOPSIS:
 *
 *    int main( void )
 *
 * FORMAL PARAMETERS:
 *
 *    ** None **
 *
 * IMPLICIT INPUTS:
 *
 *    ** None **
 *
 * IMPLICIT OUTPUTS:
 *
 *    ** None **
 *
 * FUNCTION VALUE:
 *
 *    completion status
 *
 * SIDE EFFECTS:
 *
 *    ** None **
 *
 */
```

(continued on next page)

Example E–6 (Cont.) TCP Auxiliary Server (System Services)

```
int
main( void )
{
    struct iosb iosb;                   /* i/o status block                */

    unsigned int status;                /* system service return status    */

    unsigned short conn_channel;        /* connect inet device i/o channel  */
    struct sockchar conn_sockchar;      /* connect socket char buffer       */

    unsigned int client_retlen;         /* returned length of client socket */
                                        /* address structure                */
    struct sockaddr_in client_addr;     /* client socket address structure  */
    struct itemlst_3 client_itemlst;    /* client item-list 3 descriptor    */

    char buf[] = "Hello, world!";       /* server data buffer               */
    int  buflen = sizeof( buf );        /* length of server data buffer     */

    $DESCRIPTOR( inet_device,           /* string descriptor with logical   */
                 "SYS$NET"    );        /* name of internet pseudodevice    */

    /*
     * init connection socket characteristics buffer
     */

    conn_sockchar.prot = TCPIP$C_TCP;
    conn_sockchar.type = TCPIP$C_STREAM;
    conn_sockchar.af   = TCPIP$C_AUXS;

    /*
     * init client's item-list descriptor
     */

    memset( &client_itemlst, 0, sizeof(client_itemlst) );
    client_itemlst.length  = sizeof( client_addr );
    client_itemlst.address = &client_addr;
    client_itemlst.retlen  = &client_retlen;

    /*
     * init client's socket address structure
     */

    memset( &client_addr, 0, sizeof(client_addr) );

    /*
     * assign device socket
     */

    status = sys$assign( &inet_device,      /* device name                 */
                         &conn_channel,     /* i/o channel                 */
                         0,                 /* access mode                 */
                         0                  /* not used                    */
                       );

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to assign i/o channel to TCPIP device\n" );
        exit( status );
        }

    /*
     * create connection socket
     */
```

(continued on next page)

Example E–6 (Cont.) TCP Auxiliary Server (System Services)

```
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       conn_channel,        /* i/o channel                  */
                       IO$_SETMODE,         /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       &conn_sockchar,      /* p1 - socket char buffer      */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to create socket\n" );
        exit( status );
        }

    /*
     * log this client connection
     */

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       conn_channel,        /* i/o channel                  */
                       IO$_SENSEMODE,       /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       &client_itemlst,     /* p4 - peer socket name        */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to accept client connection\n" );
        exit( status );
        }

    printf( "Accepted connection from host: %s, port: %d\n",
            inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port)
          );

    /*
     * connection established with a client;
     * now attempt to write on this connection
     */
```

(continued on next page)

Example E–6 (Cont.) TCP Auxiliary Server (System Services)

```
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       conn_channel,        /* i/o channel                  */
                       IO$_WRITEVBLK,       /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       buf,                 /* p1 - buffer address          */
                       buflen,              /* p2 - buffer length           */
                       0,                   /* p3                           */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to write data to client connection\n" );
        exit( status );
        }
    printf( "Data sent: %s\n", buf );       /* output server's data buffer  */
    /*
     * close connection socket
     */
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       conn_channel,        /* i/o channel                  */
                       IO$_DEACCESS,        /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       TCPIP$C_DSC_ALL,     /* p4 - discard all packets     */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to close socket\n" );
        exit( status );
        }

    /*
     * deassign device socket
     */

    status = sys$dassgn( conn_channel );

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to deassign i/o channel to TCPIP device\n" );
        exit( status );
        }
```

(continued on next page)

Example E–6 (Cont.) TCP Auxiliary Server (System Services)

```
    exit( EXIT_SUCCESS );
}
```

E.3 UDP Client/Server Examples (Sockets API)

This section contains the following examples:

- A UDP/IP IPv4 client using BSD Version 4.*x* Sockets API to handle network I/O operations.
- A UDP/IP IPv4 server using BSD Version 4.*x* Sockets API to handle network I/O operations.

E.3.1 UDP Client

Example E–7 shows how a typical UDP IPv4 client uses the Sockets API to handle the tasks of creating a socket, writing server data, and deleting the socket.

Example E–7 UDP Client (Sockets API)

```
#pragma module  tcpip$udp_client_sock                  \
                "V5.1-00"

/*
 * Copyright 2000 Compaq Computer Corporation
 *
 * COMPAQ Registered in U.S. Patent and Trademark Office.
 *
 * Confidential computer software.  Valid license from  Compaq
 * or  authorized sublicensor required for possession, use  or
 * copying.  Consistent with FAR 12.211 and 12.212, Commercial
 * Computer Software,   Computer Software Documentation,   and
 * Technical Data for Commercial Items  are  licensed  to  the
 * U.S. Government under vendor's standard commercial license.
 *
 * ++
 * FACILITY:
 *
 *    EXAMPLES
 *
 * ABSTRACT:
 *
 *    This is an example of a UDP/IP IPv4 client using 4.x BSD
 *    socket Application Programming Interface (API) to handle
 *    network I/O operations.
 *
 *    Refer to 'Build, Configuration, and Run Instructions' for
 *    details on how to build, configure, and run this program.
 *
 * ENVIRONMENT:
 *
 *    OpenVMS Alpha/VAX V7.1
 *    TCP/IP Services V5.0 or higher
 *
```

(continued on next page)

Example E–7 (Cont.) UDP Client (Sockets API)

```
 * AUTHOR:
 *
 *    TCPIP Development Group,  CREATION DATE: 23-May-1989
 *
 * --
 */
       /* Build, Configuration, and Run Instructions */

/*
 * BUILD INSTRUCTIONS:
 *
 *    To build this example program use commands of the form,
 *
 *    using the DEC "C" compiler:
 *
 *      $ cc/prefix=all TCPIP$UDP_CLIENT_SOCK.C
 *      $ link TCPIP$UDP_CLIENT_SOCK
 *
 *    using the DEC "C++" compiler:
 *
 *      $ cxx/prefix=all/define=VMS TCPIP$UDP_CLIENT_SOCK.C
 *      $ link TCPIP$UDP_CLIENT_SOCK
 *
 *
 * CONFIGURATION INSTRUCTIONS:
 *
 *    No special configuration required.
 *
 *
 * RUN INSTRUCTIONS:
 *
 *    To run this example program:
 *
 *    1) Start the client's server program as shown below:
 *
 *       $ run tcpip$udp_server_sock
 *       Waiting for a client datagram on port: m
 *
 *    2) After the server program blocks, start this client program,
 *       entering the server host as shown below:
 *
 *       $ run tcpip$udp_client_sock
 *       Enter remote host:
 *
 *       Note: You can specify a server host by using either an IPv4
 *             address in dotted-decimal notation (e.g. 16.20.10.56)
 *             or a host domain name (e.g. serverhost.compaq.com).
 *
 *    3) The client program then displays server address information
 *       and server data as show below:
 *
 *       Sent a datagram to host: a.b.c.d, port: n
 *       Data sent: Hello, world!
 *
 *    You can enter "ctrl/z" at any user prompt to terminate program
 *    execution.
 *
 */
```

(continued on next page)

Example E–7 (Cont.) UDP Client (Sockets API)

```
/*
 * INCLUDE FILES:
 */

#include <in.h>                    /* define internet related constants,   */
                                   /* functions, and structures            */
#include <inet.h>                  /* define network address info          */

#include <netdb.h>                 /* define network database library info */

#include <socket.h>                /* define BSD 4.x socket api            */
#include <stdio.h>                 /* define standard i/o functions        */
#include <stdlib.h>                /* define standard library functions    */
#include <string.h>                /* define string handling functions     */

#include <unixio.h>                /* define unix i/o                      */

/*
 * NAMED CONSTANTS:
 */

#define BUFSZ           1024           /* user input buffer size           */
#define SERV_PORTNUM    12345          /* server port number               */


/*
 * FORWARD REFERENCES:
 */

int  main( void );                     /* client main                      */
void get_serv_addr( void * );          /* get server host address          */
        /* Client Main */

/*
 * FUNCTIONAL DESCRIPTION:
 *
 *    This is the client's main-line code. It handles all the tasks of the
 *    client including: socket creation, writing server data, and deleting
 *    the socket.
 *
 *    This example program implements a typical UDP IPv4 client using  the
 *    BSD socket API to handle network i/o operations as shown below:
 *
 *      1) To create a socket:
 *
 *         socket()
 *
 *      2) To transfer data:
 *
 *         sendto()
 *
 *      3) To close a socket:
 *
 *         close()
 *
 *    This function is invoked by the DCL "RUN" command (see below);  the
 *    function's  completion  status is interpreted by DCL and if needed,
 *    an error message is displayed.
 *
 * SYNOPSIS:
 *
 *    int main( void )
 *
 * FORMAL PARAMETERS:
```

(continued on next page)

Example E–7 (Cont.) UDP Client (Sockets API)

```
 *
 *    ** None **
 *
 * IMPLICIT INPUTS:
 *
 *    ** None **
 *
 * IMPLICIT OUTPUTS:
 *
 *    ** None **
 *
 * FUNCTION VALUE:
 *
 *    completion status
 *
 * SIDE EFFECTS:
 *
 *    ** None **
 *
 */

int
main( void )
{
    int sockfd;                         /* udp socket descriptor            */

    char buf[] = "Hello, World!";       /* client data buffer               */

    struct sockaddr_in serv_addr;       /* server socket address structure  */

    /*
     * init server's socket address structure
     */

    memset( &serv_addr, 0, sizeof(serv_addr) );
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port   = htons( SERV_PORTNUM );
    get_serv_addr( &serv_addr.sin_addr );

    /*
     * create udp socket
     */

    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 )
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }

    /*
     * write datagram to server
     */

    if ( sendto(sockfd, buf, sizeof(buf), 0,
                (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0 )
        {
        perror( "Failed to write datagram to server" );
        exit( EXIT_FAILURE );
        }

    printf( "Sent a datagram to host: %s, port: %d\n",
            inet_ntoa(serv_addr.sin_addr), ntohs(serv_addr.sin_port)
          );
    printf( "Data sent: %s\n", buf );       /* output data buffer           */
```

(continued on next page)

Example E–7 (Cont.) UDP Client (Sockets API)

```
    /*
     * close udp socket
     */

    if ( close(sockfd) < 0 )
        {
        perror( "Failed to close socket" );
        exit( EXIT_FAILURE );
        }

    exit( EXIT_SUCCESS );
}
        /* Get Server Host Address */

/*
 * FUNCTIONAL DESCRIPTION:
 *
 *    This function gets the server host's address from the user and then
 *    stores it in the server's socket address structure.  Note that  the
 *    user can specify a server host by using either an IPv4  address  in
 *    dotted-decimal notation (e.g. 16.20.10.126) or a host  domain  name
 *    (e.g. serverhost.compaq.com).
 *
 *    Enter "ctrl/z" to terminate program execution.
 *
 * SYNOPSIS:
 *
 *    void get_serv_addr( void *addrptr )
 *
 * FORMAL PARAMETERS:
 *
 *    addrptr  -  pointer to socket address structure's 'sin_addr' field
 *                to store the specified network address
 *
 * IMPLICIT INPUTS:
 *
 *    ** None **
 *
 * IMPLICIT OUTPUTS:
 *
 *    ** None **
 *
 * FUNCTION VALUE:
 *
 *    ** None **
 *
 * SIDE EFFECTS:
 *
 *    Program execution is terminated if unable to read user's input.
 *
 */

void
get_serv_addr( void *addrptr )
{
    char buf[BUFSZ];
    struct in_addr val;
    struct hostent *host;

    while ( TRUE )
        {
        printf( "Enter remote host: " );
```

(continued on next page)

Example E–7 (Cont.) UDP Client (Sockets API)

```
        if ( fgets(buf, sizeof(buf), stdin) == NULL )
            {
            printf( "Failed to read user input\n" );
            exit( EXIT_FAILURE );
            }

        buf[strlen(buf)-1] = 0;

        val.s_addr = inet_addr( buf );

        if ( val.s_addr != INADDR_NONE )
            {
            memcpy( addrptr, &val, sizeof(struct in_addr) );
            break;
            }

        if ( (host = gethostbyname(buf)) )
            {
            memcpy( addrptr, host->h_addr, sizeof(struct in_addr) );
            break;
            }
        }
}
```

E.3.2 UDP Server

Example E–8 shows how a typical UDP IPv4 server uses the Sockets API to handle the tasks of creating a socket, binding a socket to the server's internet address and port, and reading client data.

Example E–8 UDP Server (Sockets API)

```
#pragma module  tcpip$udp_server_sock                  \
                "V5.1-00"

/*
 * Copyright 2000 Compaq Computer Corporation
 *
 * COMPAQ Registered in U.S. Patent and Trademark Office.
 *
 * Confidential computer software.  Valid license from  Compaq
 * or  authorized sublicensor required for possession, use  or
 * copying.  Consistent with FAR 12.211 and 12.212, Commercial
 * Computer Software,   Computer Software Documentation,   and
 * Technical Data for Commercial Items  are  licensed  to  the
 * U.S. Government under vendor's standard commercial license.
 *
 * ++
 * FACILITY:
 *
 *    EXAMPLES
 *
 * ABSTRACT:
 *
 *    This is an example of a UDP/IP IPv4 server using 4.x BSD
 *    socket Application Programming Interface (API) to handle
 *    network I/O operations.
 *
 *    Refer to 'Build, Configuration, and Run Instructions' for
 *    details on how to build, configure, and run this program.
 *
 * ENVIRONMENT:
 *
 *    OpenVMS Alpha/VAX V7.1
 *    TCP/IP Services V5.0 or higher
 *
 * AUTHOR:
 *
 *    TCPIP Development Group,  CREATION DATE: 23-May-1989
 *
 * --
 */
```

(continued on next page)

Example E–8 (Cont.) UDP Server (Sockets API)

```
       /* Build, Configuration, and Run Instructions */

/*
 * BUILD INSTRUCTIONS:
 *
 *    To build this example program use commands of the form,
 *
 *    using the DEC "C" compiler:
 *
 *      $ cc/prefix=all TCPIP$UDP_SERVER_SOCK.C
 *      $ link TCPIP$UDP_SERVER_SOCK
 *
 *    using the DEC "C++" compiler:
 *
 *      $ cxx/prefix=all/define=VMS TCPIP$UDP_SERVER_SOCK.C
 *      $ link TCPIP$UDP_SERVER_SOCK
 *
 *
 * CONFIGURATION INSTRUCTIONS:
 *
 *    No special configuration required.
 *
 *
 * RUN INSTRUCTIONS:
 *
 *    To run this example program:
 *
 *    1) Start this server program server as shown below:
 *
 *       $ run tcpip$udp_server_sock
 *       Waiting for a client datagram on port: m
 *
 *    2) After the server program blocks, start the client program,
 *       entering the server host as shown below:
 *
 *       $ run tcpip$udp_client_sock
 *       Enter remote host:
 *
 *       Note: You can specify a server host by using either an IPv4
 *             address in dotted-decimal notation (e.g. 16.20.10.56)
 *             or a host domain name (e.g. serverhost.compaq.com).
 *
 *    3) The server program then displays client address information
 *       and client data as show below:
 *
 *       Received a datagram from host: a.b.c.d, port: n
 *       Data received: Hello, world!
 *
 *    You can enter "ctrl/z" at any user prompt to terminate program
 *    execution.
 *
 */

/*
 * INCLUDE FILES:
 */

#include <in.h>                    /* define internet related constants,   */
                                   /* functions, and structures            */
#include <inet.h>                  /* define network address info          */

#include <netdb.h>                 /* define network database library info */
```

(continued on next page)

Example E–8 (Cont.) UDP Server (Sockets API)

```
#include <socket.h>                /* define BSD 4.x socket api           */
#include <stdio.h>                 /* define standard i/o functions       */
#include <stdlib.h>                /* define standard library functions   */
#include <string.h>                /* define string handling functions    */

#include <unixio.h>                /* define unix i/o                     */

/*
 * NAMED CONSTANTS:
 */

#define SERV_PORTNUM    12345          /* server port number              */

/*
 * FORWARD REFERENCES:
 */

int  main( void );                     /* server main                     */
        /* Server Main */

/*
 * FUNCTIONAL DESCRIPTION:
 *
 *    This is the server's main-line code. It handles all the tasks of the
 *    server including: socket creation, binding a socket to the  server's
 *    internet address and port, and reading client data.
 *
 *    This example program implements a typical UDP IPv4 server using  the
 *    BSD socket API to handle network i/o operations as shown below:
 *
 *      1) To create a socket:
 *
 *         socket()
 *
 *      2) To set REUSEADDR socket option:
 *
 *         setsockopt()
 *
 *      3) To bind internet address and port number to a socket:
 *
 *         bind()
 *
 *      4) To transfer data:
 *
 *         recvfrom()
 *
 *      5) To close a socket:
 *
 *         close()
 *
 *    This function is invoked by the DCL "RUN" command (see below);  the
 *    function's  completion  status is interpreted by DCL and if needed,
 *    an error message is displayed.
 *
 * SYNOPSIS:
 *
 *    int main( void )
 *
 * FORMAL PARAMETERS:
 *
 *    ** None **
```

(continued on next page)

Example E–8 (Cont.) UDP Server (Sockets API)

```
 *
 * IMPLICIT INPUTS:
 *
 *    ** None **
 *
 * IMPLICIT OUTPUTS:
 *
 *    ** None **
 *
 * FUNCTION VALUE:
 *
 *    completion status
 *
 * SIDE EFFECTS:
 *
 *    ** None **
 *
 */

int
main( void )
{
    char buf[512];                      /* server data buffer               */
    int optval = 1;                     /* SO_REUSEADDR'S option value (on) */
    int sockfd;                         /* socket descriptor                */
    unsigned int client_addrlen;        /* returned length of client socket */
                                        /* address structure                */
    struct sockaddr_in client_addr;     /* client socket address structure  */
    struct sockaddr_in serv_addr;       /* server socket address structure  */

    /*
     * init client's socket address structure
     */

    memset( &client_addr, 0, sizeof(client_addr) );

    /*
     * init server's socket address structure
     */

    memset( &serv_addr, 0, sizeof(serv_addr) );
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_port        = htons( SERV_PORTNUM );
    serv_addr.sin_addr.s_addr = INADDR_ANY;

    /*
     * create a udp socket
     */

    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 )
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }

    /*
     * bind server's internet address and port number to socket
     */
```

(continued on next page)

Example E–8 (Cont.) UDP Server (Sockets API)

```
    if ( setsockopt(sockfd,
                  SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval)) < 0 )
        {
        perror( "Failed to set socket option" );
        exit( EXIT_FAILURE );
        }

    if ( bind(sockfd,
              (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0 )
        {
        perror( "Failed to bind socket" );
        exit( EXIT_FAILURE );
        }

    /*
     * read datagram from client
     */

    printf( "Waiting for a client datagram on port: %d\n",
            ntohs(serv_addr.sin_port)
          );

    client_addrlen = sizeof(client_addr);

    if ( recvfrom(sockfd, buf, sizeof(buf), 0,
                  (struct sockaddr *) &client_addr, &client_addrlen) < 0 )
        {
        perror( "Failed to read datagram from client" );
        exit( EXIT_FAILURE );
        }

    printf( "Received a datagram from host: %s, port: %d\n",
            inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port)
          );
    printf( "Data received: %s\n", buf );  /* output client's data buffer  */

    /*
     * close udp socket
     */

    if ( close(sockfd) < 0 )
        {
        perror( "Failed to close socket" );
        exit( EXIT_FAILURE );
        }

    exit( EXIT_SUCCESS );
}
```

E.4 UDP Client/Server Examples (System Services)

This section contains the following examples:

- A UDP/IP IPv4 client using OpenVMS system services to handle network I/O operations.
- A UDP/IP IPv4 server using OpenVMS system services to handle newtork I/O operations.

E.4.1 UDP Client

Example E–9 shows how a typical UDP IPv4 client uses $QIO system services to handle the tasks of creating a socket, writing server data, and deleting the socket.

Example E–9 UDP Client (System Services)

```
#pragma module  tcpip$udp_client_qio                    \
                "V5.1-00"

/*
 * Copyright 2000 Compaq Computer Corporation
 *
 * COMPAQ Registered in U.S. Patent and Trademark Office.
 *
 * Confidential computer software.  Valid license from  Compaq
 * or  authorized sublicensor required for possession, use  or
 * copying.  Consistent with FAR 12.211 and 12.212, Commercial
 * Computer Software,   Computer Software Documentation,   and
 * Technical Data for Commercial Items  are  licensed  to  the
 * U.S. Government under vendor's standard commercial license.
 *
 * ++
 * FACILITY:
 *
 *    EXAMPLES
 *
 * ABSTRACT:
 *
 *    This is an example of a UDP/IP IPv4 client using  OpenVMS
 *    QIO system services to handle network I/O operations.
 *
 *    Refer to 'Build, Configuration, and Run Instructions' for
 *    details on how to build, configure, and run this program.
 *
 * ENVIRONMENT:
 *
 *    OpenVMS Alpha/VAX V7.1
 *    TCP/IP Services V5.0 or higher
 *
 * AUTHOR:
 *
 *    TCPIP Development Group,  CREATION DATE: 23-May-1989
 *
 * --
 */
```

(continued on next page)

Example E–9 (Cont.) UDP Client (System Services)

```
        /* Build, Configuration, and Run Instructions */

/*
 * BUILD INSTRUCTIONS:
 *
 *    To build this example program use commands of the form,
 *
 *    using the DEC "C" compiler:
 *
 *      $ cc/prefix=all TCPIP$UDP_CLIENT_QIO.C
 *      $ link TCPIP$UDP_CLIENT_QIO
 *
 *    using the DEC "C++" compiler:
 *
 *      $ cxx/prefix=all/define=VMS TCPIP$UDP_CLIENT_QIO.C
 *      $ link TCPIP$UDP_CLIENT_QIO
 *
 *
 * CONFIGURATION INSTRUCTIONS:
 *
 *    No special configuration required.
 *
 *
 * RUN INSTRUCTIONS:
 *
 *    To run this example program:
 *
 *    1) Start the client's server program as shown below:
 *
 *       $ run tcpip$udp_server_qio
 *       Waiting for a client datagram on port: m
 *
 *    2) After the server program blocks, start this client program,
 *       entering the server host as shown below:
 *
 *       $ run tcpip$udp_client_qio
 *       Enter remote host:
 *
 *       Note: You can specify a server host by using either an IPv4
 *             address in dotted-decimal notation (e.g. 16.20.10.56)
 *             or a host domain name (e.g. serverhost.compaq.com).
 *
 *    3) The client program then displays server address information
 *       and server data as show below:
 *
 *       Sent a datagram to host: a.b.c.d, port: n
 *       Data sent: Hello, world!
 *
 *    You can enter "ctrl/z" at any user prompt to terminate program
 *    execution.
 *
 */

/*
 * INCLUDE FILES:
 */

#include <descrip.h>                /* define OpenVMS descriptors          */

#include <efndef.h>                 /* define 'EFN$C_ENF' event flag       */
```

(continued on next page)

Example E–9 (Cont.) UDP Client (System Services)

```
#include <in.h>                     /* define internet related constants,   */
                                    /* functions, and structures            */
#include <inet.h>                   /* define network address info          */
#include <iodef.h>                  /* define i/o function codes            */

#include <netdb.h>                  /* define network database library info */

#include <ssdef.h>                  /* define system service status codes   */
#include <starlet.h>                /* define system service calls          */
#include <stdio.h>                  /* define standard i/o functions        */
#include <stdlib.h>                 /* define standard library functions    */
#include <string.h>                 /* define string handling functions     */
#include <stsdef.h>                 /* define condition value fields        */

#include <tcpip$inetdef.h>          /* define tcp/ip network constants,     */
                                    /* structures, and functions            */


/*
 * NAMED CONSTANTS:
 */

#define BUFSZ           1024            /* user input buffer size           */
#define SERV_PORTNUM    12345           /* server port number               */

/*
 * STRUCTURE DEFINITIONS:
 */

struct iosb
    {                                   /* i/o status block                 */
    unsigned short status;              /* i/o completion status            */
    unsigned short bytcnt;              /* bytes transferred if read/write  */
    void *details;                      /* address of buffer or parameter   */
    };

struct itemlst_2
    {                                   /* item-list 2 descriptor/element   */
    unsigned short length;              /* length                           */
    unsigned short type;                /* parameter type                   */
    void *address;                      /* address of item list             */
    };

struct sockchar
    {                                   /* socket characteristics buffer    */
    unsigned short prot;                /* protocol                         */
    unsigned char type;                 /* type                             */
    unsigned char af;                   /* address format                   */
    };


/*
 * FORWARD REFERENCES:
 */

int  main( void );                      /* client main                      */
void get_serv_addr( void * );           /* get server host address          */
```

(continued on next page)

Example E–9 (Cont.) UDP Client (System Services)

```
        /* Client Main */

/*
 * FUNCTIONAL DESCRIPTION:
 *
 *    This is the client's main-line code. It handles all the tasks of the
 *    client including: socket creation, writing server data, and deleting
 *    the socket.
 *
 *    This example program implements a typical UDP IPv4 client using  QIO
 *    system services to handle network i/o operations as shown below:
 *
 *      1) To create a socket:
 *
 *         sys$assign() and sys$qiow(IO$_SETMODE)
 *
 *      2) To transfer data:
 *
 *         sys$qiow(IO$_WRITEVBLK)
 *
 *      3) To close and delete a socket:
 *
 *         sys$qiow(IO$_DEACCESS) and sys$dassgn()
 *
 *    This function is invoked by the DCL "RUN" command (see below);  the
 *    function's  completion  status is interpreted by DCL and if needed,
 *    an error message is displayed.
 *
 * SYNOPSIS:
 *
 *    int main( void )
 *
 * FORMAL PARAMETERS:
 *
 *    ** None **
 *
 * IMPLICIT INPUTS:
 *
 *    ** None **
 *
 * IMPLICIT OUTPUTS:
 *
 *    ** None **
 *
 * FUNCTION VALUE:
 *
 *    completion status
 *
 * SIDE EFFECTS:
 *
 *    ** None **
 *
 */

int
main( void )
{
    struct iosb iosb;                   /* i/o status block                */
    unsigned int status;                /* system service return status    */
```

(continued on next page)

Example E–9 (Cont.) UDP Client (System Services)

```
    unsigned short inet_channel;        /* inet device i/o channel          */
    struct sockchar udp_sockchar;       /* udp socket char buffer           */

    struct sockaddr_in serv_addr;       /* server socket address structure  */
    struct itemlst_2 serv_itemlst;      /* server item-list 2 descriptor    */

    $DESCRIPTOR( inet_device,           /* string descriptor with logical   */
                 "TCPIP$DEVICE" );      /* name of internet pseudodevice    */

    char buf[]  = "Hello, World!";      /* client data buffer               */
    int  buflen = sizeof( buf );        /* length of client data buffer     */


    /*
     * init client socket characteristics buffer
     */

    udp_sockchar.prot = TCPIP$C_UDP;
    udp_sockchar.type = TCPIP$C_DGRAM;
    udp_sockchar.af   = TCPIP$C_AF_INET;

    /*
     * init server's item-list descriptor
     */

    memset( &serv_itemlst, 0, sizeof(serv_itemlst) );
    serv_itemlst.length  = sizeof( serv_addr );
    serv_itemlst.address = &serv_addr;

    /*
     * init server's socket address structure
     */

    memset( &serv_addr, 0, sizeof(serv_addr) );
    serv_addr.sin_family = TCPIP$C_AF_INET;
    serv_addr.sin_port   = htons( SERV_PORTNUM );
    get_serv_addr( &serv_addr.sin_addr );

    /*
     * assign device socket
     */

    status = sys$assign( &inet_device,      /* device name                  */
                         &inet_channel,     /* i/o channel                  */
                         0,                 /* access mode                  */
                         0                  /* not used                     */
                       );

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to assign i/o channel to TCPIP device\n" );
        exit( status );
        }

    /*
     * create udp socket
     */
```

(continued on next page)

Example E–9 (Cont.) UDP Client (System Services)

```
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       inet_channel,        /* i/o channel                  */
                       IO$_SETMODE,         /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       &udp_sockchar,       /* p1 - socket char buffer      */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to create socket\n" );
        exit( status );
        }

    /*
     * write datagram to server
     */

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       inet_channel,        /* i/o channel                  */
                       IO$_WRITEVBLK,       /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       buf,                 /* p1 - buffer address          */
                       buflen,              /* p2 - buffer length           */
                       &serv_itemlst,       /* p3 - remote socket name      */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to write datagram to server\n" );
        exit( status );
        }

    printf( "Sent a datagram to host: %s, port: %d\n",
            inet_ntoa(serv_addr.sin_addr), ntohs(serv_addr.sin_port)
          );
    printf( "Data sent: %s\n", buf );       /* output data buffer           */

    /*
     * close udp socket
     */
```

(continued on next page)

Example E–9 (Cont.) UDP Client (System Services)

```
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       inet_channel,        /* i/o channel                  */
                       IO$_DEACCESS,        /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       0,                   /* p3                           */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to close socket\n" );
        exit( status );
        }

    /*
     * deassign device socket
     */

    status = sys$dassgn( inet_channel );

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to deassign i/o channel to TCPIP device\n" );
        exit( status );
        }

    exit( EXIT_SUCCESS );
}
        /* Get Server Host Address */
```

(continued on next page)

Example E–9 (Cont.) UDP Client (System Services)

```
/*
 * FUNCTIONAL DESCRIPTION:
 *
 *    This function gets the server host's address from the user and then
 *    stores it in the server's socket address structure.  Note that  the
 *    user can specify a server host by using either an IPv4  address  in
 *    dotted-decimal notation (e.g. 16.20.10.126) or a host  domain  name
 *    (e.g. serverhost.compaq.com).
 *
 *    Enter "ctrl/z" to terminate program execution.
 *
 * SYNOPSIS:
 *
 *    void get_serv_addr( void *addrptr )
 *
 * FORMAL PARAMETERS:
 *
 *    addrptr  -  pointer to socket address structure's 'sin_addr' field
 *                to store the specified network address
 *
 * IMPLICIT INPUTS:
 *
 *    ** None **
 *
 * IMPLICIT OUTPUTS:
 *
 *    ** None **
 *
 * FUNCTION VALUE:
 *
 *    ** None **
 *
 * SIDE EFFECTS:
 *
 *    Program execution is terminated if unable to read user's input.
 *
 */

void
get_serv_addr( void *addrptr )
{
    char buf[BUFSZ];
    struct in_addr val;
    struct hostent *host;

    while ( TRUE )
        {
        printf( "Enter remote host: " );

        if ( fgets(buf, sizeof(buf), stdin) == NULL )
            {
            printf( "Failed to read user input\n" );
            exit( EXIT_FAILURE );
            }

        buf[strlen(buf)-1] = 0;

        val.s_addr = inet_addr( buf );
```

(continued on next page)

Example E–9 (Cont.) UDP Client (System Services)

```
        if ( val.s_addr != INADDR_NONE )
            {
            memcpy( addrptr, &val, sizeof(struct in_addr) );
            break;
            }
        if ( (host = gethostbyname(buf)) )
            {
            memcpy( addrptr, host->h_addr, sizeof(struct in_addr) );
            break;
            }
        }
}
```

E.4.2 UDP Server

Example E–10 shows how a typical UDP IPv4 server uses $QIO system services to handle the tasks of creating a socket, binding a socket to the server's internet address and port, and reading client data.

Example E–10 UDP Server (System Services)

```
#pragma module  tcpip$udp_server_qio                    \
                "V5.1-00"
/*
 * Copyright 2000 Compaq Computer Corporation
 *
 * COMPAQ Registered in U.S. Patent and Trademark Office.
 *
 * Confidential computer software.  Valid license from  Compaq
 * or  authorized sublicensor required for possession, use  or
 * copying.  Consistent with FAR 12.211 and 12.212, Commercial
 * Computer Software,   Computer Software Documentation,   and
 * Technical Data for Commercial Items  are  licensed  to  the
 * U.S. Government under vendor's standard commercial license.
 *
 * ++
 * FACILITY:
 *
 *    EXAMPLES
 *
 * ABSTRACT:
 *
 *    This is an example of a UDP/IP IPv4 server using  OpenVMS
 *    QIO system services to handle network I/O operations.
 *
 *    Refer to 'Build, Configuration, and Run Instructions' for
 *    details on how to build, configure, and run this program.
 *
 * ENVIRONMENT:
 *
 *    OpenVMS Alpha/VAX V7.1
 *    TCP/IP Services V5.0 or higher
 *
 * AUTHOR:
 *
 *    TCPIP Development Group,  CREATION DATE: 23-May-1989
```

(continued on next page)

Example E–10 (Cont.) UDP Server (System Services)

```
 *
 * --
 */

       /* Build, Configuration, and Run Instructions */

/*
 * BUILD INSTRUCTIONS:
 *
 *    To build this example program use commands of the form,
 *
 *    using the DEC "C" compiler:
 *
 *      $ cc/prefix=all TCPIP$UDP_SERVER_QIO.C
 *      $ link TCPIP$UDP_SERVER_QIO
 *
 *    using the DEC "C++" compiler:
 *
 *      $ cxx/prefix=all/define=VMS TCPIP$UDP_SERVER_QIO.C
 *      $ link TCPIP$UDP_SERVER_QIO
 *
 *
 * CONFIGURATION INSTRUCTIONS:
 *
 *    No special configuration required.
 *
 *
 * RUN INSTRUCTIONS:
 *
 *    To run this example program:
 *
 *    1) Start this server program as shown below:
 *
 *       $ run tcpip$udp_server_qio
 *       Waiting for a client datagram on port: m
 *
 *    2) After the server program blocks, start the client program,
 *       entering the server host as shown below:
 *
 *       $ run tcpip$udp_client_qio
 *       Enter remote host:
 *
 *       Note: You can specify a server host by using either an IPv4
 *             address in dotted-decimal notation (e.g. 16.20.10.56)
 *             or a host domain name (e.g. serverhost.compaq.com).
 *
 *    3) The server program then displays client address information
 *       and client data as show below:
 *
 *       Received a datagram from host: a.b.c.d, port: n
 *       Data received: Hello, world!
 *
 *    You can enter "ctrl/z" at any user prompt to terminate program
 *    execution.
 *
 */

/*
 * INCLUDE FILES:
 */
#include <descrip.h>               /* define OpenVMS descriptors          */
```

(continued on next page)

Example E–10 (Cont.) UDP Server (System Services)

```
#include <efndef.h>                 /* define 'EFN$C_ENF' event flag       */

#include <in.h>                     /* define internet related constants,  */
                                    /* functions, and structures           */
#include <inet.h>                   /* define network address info         */
#include <iodef.h>                  /* define i/o function codes           */

#include <netdb.h>                  /* define network database library info */

#include <ssdef.h>                  /* define system service status codes  */
#include <starlet.h>                /* define system service calls         */
#include <stdio.h>                  /* define standard i/o functions       */
#include <stdlib.h>                 /* define standard library functions   */
#include <string.h>                 /* define string handling functions    */
#include <stsdef.h>                 /* define condition value fields       */

#include <tcpip$inetdef.h>          /* define tcp/ip network constants,    */
                                    /* structures, and functions           */


/*
 * NAMED CONSTANTS:
 */

#define SERV_PORTNUM    12345           /* server port number              */

/*
 * STRUCTURE DEFINITIONS:
 */

struct iosb
    {                               /* i/o status block                    */
    unsigned short status;          /* i/o completion status               */
    unsigned short bytcnt;          /* bytes transferred if read/write     */
    void *details;                  /* address of buffer or parameter      */
    };

struct itemlst_2
    {                               /* item-list 2 descriptor/element      */
    unsigned short length;          /* length                              */
    unsigned short type;            /* parameter type                      */
    void *address;                  /* address of item list                */
    };

struct itemlst_3
    {                               /* item-list 3 descriptor/element      */
    unsigned short length;          /* length                              */
    unsigned short type;            /* parameter type                      */
    void *address;                  /* address of item list                */
    unsigned int *retlen;           /* address of returned length          */
    };

struct sockchar
    {                               /* socket characteristics buffer       */
    unsigned short prot;            /* protocol                            */
    unsigned char type;             /* type                                */
    unsigned char af;               /* address format                      */
    };


/*
 * FORWARD REFERENCES:
 */

int  main( void );                  /* client main                         */
```

(continued on next page)

Example E–10 (Cont.) UDP Server (System Services)

```
        /* Server Main */

/*
 * FUNCTIONAL DESCRIPTION:
 *
 *    This is the server's main-line code. It handles all the tasks of the
 *    server including: socket creation, binding a socket to the  server's
 *    internet address and port, and reading client data.
 *
 *    This example program implements a typical UDP IPv4 server using  QIO
 *    system services to handle network i/o operations as shown below:
 *
 *      1) To create a socket and set REUSEADDR option:
 *
 *         sys$assign() and sys$qiow(IO$_SETMODE)
 *
 *      2) To bind internet address and port number to a socket:
 *
 *         sys$qiow(IO$_SETMODE)
 *
 *      3) To transfer data:
 *
 *         sys$qiow(IO$_READVBLK)
 *
 *      4) To close and delete a socket:
 *
 *         sys$qiow(IO$_DEACCESS) and sys$dassgn()
 *
 *    This function is invoked by the DCL "RUN" command (see below);  the
 *    function's  completion  status is interpreted by DCL and if needed,
 *    an error message is displayed.
 *
 * SYNOPSIS:
 *
 *    int main( void )
 *
 * FORMAL PARAMETERS:
 *
 *    ** None **
 *
 * IMPLICIT INPUTS:
 *
 *    ** None **
 *
 * IMPLICIT OUTPUTS:
 *
 *    ** None **
 *
 * FUNCTION VALUE:
 *
 *    completion status
 *
 * SIDE EFFECTS:
 *
 *    ** None **
 *
 */
```

(continued on next page)

Example E–10 (Cont.) UDP Server (System Services)

```
int
main( void )
{
    int one = 1;                        /* reuseaddr option value          */

    char buf[512];                      /* server data buffer              */
    int  buflen = sizeof( buf );        /* length of server data buffer    */

    struct iosb iosb;                   /* i/o status block                */

    unsigned int status;                /* system service return status    */

    unsigned short inet_channel;        /* inet device i/o channel         */
    struct sockchar udp_sockchar;       /* socket char buffer              */

    unsigned int client_retlen;         /* returned length of client socket */
                                        /* address structure               */
    struct sockaddr_in client_addr;     /* client socket address structure  */
    struct itemlst_3 client_itemlst;    /* client item-list 3 descriptor    */

    struct sockaddr_in serv_addr;       /* server socket address structure  */
    struct itemlst_2 serv_itemlst;      /* server item-list 2 descriptor    */

    struct itemlst_2 sockopt_itemlst;   /* sockopt item-list 2 descriptor   */
    struct itemlst_2 reuseaddr_itemlst; /* reuseaddr item-list 2 element    */

    $DESCRIPTOR( inet_device,           /* string descriptor with logical   */
                 "TCPIP$DEVICE" );      /* name of internet pseudodevice    */

    /*
     * init udp socket characteristics buffer
     */

    udp_sockchar.prot = TCPIP$C_UDP;
    udp_sockchar.type = TCPIP$C_DGRAM;
    udp_sockchar.af   = TCPIP$C_AF_INET;

    /*
     * init reuseaddr's item-list element
     */

    reuseaddr_itemlst.length  = sizeof( one );
    reuseaddr_itemlst.type    = TCPIP$C_REUSEADDR;
    reuseaddr_itemlst.address = &one;

    /*
     * init sockopt's item-list descriptor
     */

    sockopt_itemlst.length  = sizeof( reuseaddr_itemlst );
    sockopt_itemlst.type    = TCPIP$C_SOCKOPT;
    sockopt_itemlst.address = &reuseaddr_itemlst;

    /*
     * init client's item-list descriptor
     */

    memset( &client_itemlst, 0, sizeof(client_itemlst) );
    client_itemlst.length  = sizeof( client_addr );
    client_itemlst.address = &client_addr;
    client_itemlst.retlen  = &client_retlen;

    /*
     * init client's socket address structure
     */

    memset( &client_addr, 0, sizeof(client_addr) );
```

(continued on next page)

Example E–10 (Cont.) UDP Server (System Services)

```
/*
 * init server's item-list descriptor
 */

serv_itemlst.length  = sizeof( serv_addr );
serv_itemlst.type    = TCPIP$C_SOCK_NAME;
serv_itemlst.address = &serv_addr;

/*
 * init server's socket address structure
 */

memset( &serv_addr, 0, sizeof(serv_addr) );
serv_addr.sin_family      = TCPIP$C_AF_INET;
serv_addr.sin_port        = htons( SERV_PORTNUM );
serv_addr.sin_addr.s_addr = TCPIP$C_INADDR_ANY;

/*
 * assign device socket
 */

status = sys$assign( &inet_device,      /* device name                 */
                     &inet_channel,     /* i/o channel                 */
                     0,                 /* access mode                 */
                     0                  /* not used                    */
                   );

if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to assign i/o channel to TCPIP device\n" );
    exit( status );
    }

/*
 * create udp socket
 */

status = sys$qiow( EFN$C_ENF,           /* event flag                  */
                   inet_channel,        /* i/o channel                 */
                   IO$_SETMODE,         /* i/o function code           */
                   &iosb,               /* i/o status block            */
                   0,                   /* ast service routine         */
                   0,                   /* ast parameter               */
                   &udp_sockchar,       /* p1 - socket char buffer     */
                   0,                   /* p2                          */
                   0,                   /* p3                          */
                   0,                   /* p4                          */
                   &sockopt_itemlst,    /* p5 - socket options         */
                   0                    /* p6                          */
                 );

if ( status & STS$M_SUCCESS )
    status = iosb.status;

if ( !(status & STS$M_SUCCESS) )
    {
    printf( "Failed to create socket\n" );
    exit( status );
    }

/*
 * bind server's internet address and port number to socket
 */
```

(continued on next page)

Example E–10 (Cont.) UDP Server (System Services)

```
    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       inet_channel,        /* i/o channel                  */
                       IO$_SETMODE,         /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       0,                   /* p1                           */
                       0,                   /* p2                           */
                       &serv_itemlst,       /* p3 - local socket name       */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to bind socket\n" );
        exit( status );
        }

    /*
     * read datagram from client
     */

    printf( "Waiting for a client datagram on port: %d\n",
            ntohs(serv_addr.sin_port)
          );

    status = sys$qiow( EFN$C_ENF,           /* event flag                   */
                       inet_channel,        /* i/o channel                  */
                       IO$_READVBLK,        /* i/o function code            */
                       &iosb,               /* i/o status block             */
                       0,                   /* ast service routine          */
                       0,                   /* ast parameter                */
                       buf,                 /* p1 - buffer address          */
                       buflen,              /* p2 - buffer length           */
                       &client_itemlst,     /* p3 - remote socket name      */
                       0,                   /* p4                           */
                       0,                   /* p5                           */
                       0                    /* p6                           */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to read datagram from client\n" );
        exit( status );
        }
```

(continued on next page)

Example E–10 (Cont.) UDP Server (System Services)

```
    printf( "Received a datagram from host: %s, port: %d\n",
            inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port)
          );
    printf( "Data received: %s\n", buf );   /* output client's data        */

    /*
     * close udp socket
     */

    status = sys$qiow( EFN$C_ENF,           /* event flag                  */
                       inet_channel,        /* i/o channel                 */
                       IO$_DEACCESS,        /* i/o function code           */
                       &iosb,               /* i/o status block            */
                       0,                   /* ast service routine         */
                       0,                   /* ast parameter               */
                       0,                   /* p1                          */
                       0,                   /* p2                          */
                       0,                   /* p3                          */
                       0,                   /* p4                          */
                       0,                   /* p5                          */
                       0                    /* p6                          */
                     );

    if ( status & STS$M_SUCCESS )
        status = iosb.status;

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to close socket\n" );
        exit( status );
        }

    /*
     * deassign device socket
     */

    status = sys$dassgn( inet_channel );

    if ( !(status & STS$M_SUCCESS) )
        {
        printf( "Failed to deassign i/o channel to TCPIP device\n" );
        exit( status );
        }

    exit( EXIT_SUCCESS );
}
```

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

W