DECnet–VAXmate

Programmer's Reference Manual

Order No. AA–GV34A–TH

September 1986

This manual details the software requirements and design considerations necessary for creating DECnet–VAXmate network applications.

Supersession/Update Information:	This is a new manual.
Operating System and Version:	VAXmate MS–DOS V3.10
Software Version:	DECnet–VAXmate V1.0

digital

AA-GV34A-TH
First Printing, September 1986

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital or its affiliated companies.

The postage-prepaid Reader's Comments form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	PDP	VAX
DECmate	P/OS	VAXcluster
DECnet	Professional	VAXmate
DECUS	Rainbow	VMS
DECwriter	RSTS	VT
DIBOL	RSX	Work Processor
digital	RT	
MASSBUS	UNIBUS	

MS™, XENIX™, MS-DOS™, MS-NET™, and MS-Windows™ are trademarks of Microsoft Corporation.
IBM is a registered trademark of International Business Machines Corporation.
PC/XT and Personal Computer AT are trademarks of the International Business Machines Corporation.

This manual was produced by Networks and Communications Publications.

Contents

4 C Language

5 DECnet Utility Functions

6 Assembly Language

A Socket Definitions

H DECnet–VAXmate Programming Examples

Figures

Tables

Preface

DECnet–VAXmate V1.0 is a software communications product that enables individual personal computer systems to communicate with one another and with other computer systems, in a DECnet network.

Manual Objectives

The *DECnet–VAXmate Programmer's Reference Manual* discusses software requirements for creating DECnet–VAXmate applications. It provides detailed information on the use of C and MS–DOS system calls supported by DECnet–VAXmate. It discusses software considerations when developing DECnet–VAXmate applications in C or assembly programming languages.

It also assumes that you are familiar with the DECnet and the MS–DOS environments.

Intended Audience

This manual is designed for application developers who are responsible for creating DECnet–VAXmate applications.

Structure of the Manual

This manual consists of 6 chapters and 8 appendixes.

- Chapter 1 presents an overview of the DECnet–VAXmate programming environment. It discusses the features of DECnet task-to-task communication.
- Chapter 2 describes standard MS–DOS function calls used for transparent file access.
- Chapter 3 discusses standard MS–DOS function calls used for transparent task-to-task communication.

- Chapter 4 details the use of C language for nontransparent task-to-task communication. The socket interface calls are described in alphabetical order. Each description includes the call's syntax, argument(s) and associated error/completion status codes.
- Chapter 5 discusses socket interface utilities including those that access network-related databases.
- Chapter 6 discusses the use of assembly language for nontransparent task-to-task communication. It details the network process function calls supported by DECnet–VAXmate. They are described in a manner similar to the calls in Chapter 4.
- Appendix A lists common definitions for the socket network interface.
- Appendix B details the format of the data structures used with the socket interface and assembly language network process interface calls.
- Appendix C lists error completion codes for DECnet–VAXmate task-to-task communications and transparent file access operations.
- Appendix D summarizes the DECnet system error messages.
- Appendix E lists DAP error messages that provide extended error information to transparent file access operations.
- Appendix F summarizes extended error messages for transparent file access operations.
- Appendix G lists specific socket interface and assembly language network process interface calls which cannot be transported to a DECnet–ULTRIX system.
- Appendix H contains C programming examples using socket interface function calls.

Graphic Conventions Used in This Manual

Convention	Meaning
bold	Words in **boldface** are considered to be literal and are typed exactly as shown. The call name is always shown as literal.
Monospaced type	Monospaced type indicates examples of system output or user input. System output is in black; user input is in red.
italics	Italics in commands and examples indicate that either the system supplies or you should supply a value.
CTRL/*x*	CTRL/*x* indicates that you should hold down the CONTROL key while you press the *x* key, where *x* represents a specific key.
key	Indicates that you should press the specified key.
#include <file.h>	When #include appears in a SYNTAX section, it indicates an include or header file. This type of file contains a collection of definitions commonly used throughout a program. You must include this file whenever it is required by a specific function call.

Associated Documents

You should have the following documents available for reference:

- *DECnet–VAXmate Installation Guide*
- *DECnet–VAXmate Getting Started*
- *DECnet–VAXmate User's Guide*
- *DECnet–VAXmate Mini-Reference Guide*
- *DECnet–VAXmate Release Notes*
- The installation guide and introductory manuals for your computer

Introduction

DECnet is the name given to a family of software and hardware communications products that provide a network interface for Digital operating systems. The relationships between the various network components are governed by a set of standards called the Digital Network Architecture (DNA).

DECnet enables multiple computer systems to participate in communications and resource sharing within a specific network. The individual computer systems, called nodes, are connected by physical communications paths. Tasks that run on different nodes and exchange data are connected by logical links. Logical links are temporary software information paths established between two communicating tasks in a DECnet network.

DECnet–VAXmate is a nonrouting implementation of the Phase IV Digital Network Architecture. The DECnet–VAXmate software includes the following features:

- Task-to-task communication between a VAXmate system and any Phase III system connected to a DECnet router, or any other Phase IV system.
- Limited network management and maintenance functions.
- Transport facilities that permit programs to access remote files.
- Virtual disk and printer, and remote terminal capabilities.
- Mail utility that lets you send messages and text files to other nodes in the network.
- File access, for other users in the network, to the files local to your personal computer.

1
Introduction to Network Programming

DECnet–VAXmate allows VAXmate personal computers to participate as end nodes in DECnet computer networks. DECnet–VAXmate nodes can connect directly either to an Ethernet local area network or to an adjacent routing node. The DECnet–VAXmate software product offers the following set of functions:

- **Task-to-task communication.** Task-to-task communication is a feature common to all DECnet implementations. It allows tasks or application programs to communicate with each other. Cooperating tasks on different nodes issue DECnet calls which enable them to exchange data over a logical link.

  DECnet–VAXmate allows you to create C and assembly language programs that use nontransparent task-to-task communication functions. A set of DECnet utility functions enables you to access the network node database and manipulate the data.

  DECnet–VAXmate supports the DECnet–ULTRIX task-to-task network communications interface. Any DECnet–VAXmate C program, using compatible DECnet–ULTRIX network interface calls, can be transported to DECnet–ULTRIX systems. (See Appendix G for possible exceptions.)

  A simpler programming interface allows DECnet–VAXmate tasks to exchange data with a remote network program. To perform transparent operations, tasks use standard MS–DOS I/O system calls.

- **Network resource access.** DECnet–VAXmate offers a set of utilities that allows a user to access network resources. Using a DECnet–VAXmate utility, you can transfer files between a VAXmate and another node in the network. Transparent file access is also available to a DECnet–VAXmate task by adding network location information to an MS–DOS path name string. Accessing remote DECnet files is accomplished with standard MS–DOS I/O system function requests.

With DECnet–VAXmate, users can establish a virtual terminal connection to a multi-user remote DECnet system. (The remote system must provide similar support.) This feature allows for sharing of resources and application development tools of larger DECnet systems.

DECnet–VAXmate provides the ability to access remote network devices. A utility program allows you to share remote printers and disks as if they were directly connected to your personal computer.

- **Network management.** Limited network management is available with DECnet–VAXmate. This feature serves three primary functions: to configure a DECnet–VAXmate node, display statistical and error information, and test the operation of the network connection.

Figure 1–1 shows personal computers running DECnet–VAXmate in an Ethernet local area network.

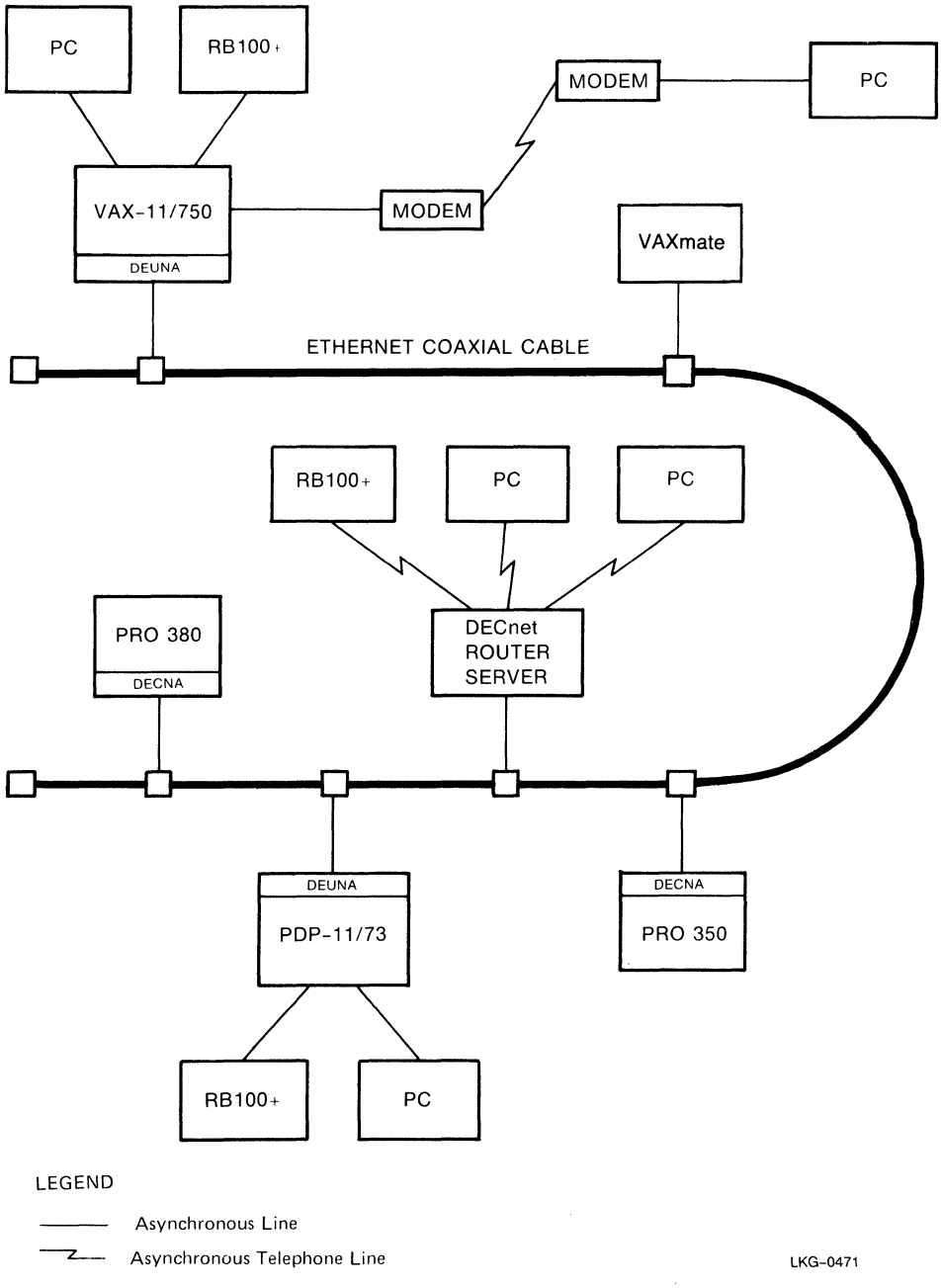


Figure 1–1: Personal Computers in an Ethernet LAN

1.1 Concepts

Before you can create a DECnet–VAXmate application, you need to understand the following programming concepts:

- **Client and server tasks.** Client and server tasks communicate through sockets. These tasks exchange data over logical links.
- **Sockets.** Sockets are the basic building blocks for DECnet–VAXmate task-to-task communication. They are created by tasks for sending and receiving data. They contain information about the status of the logical link connection.

1.1.1 Client and Server Tasks

DECnet–VAXmate communication requires cooperation between two programs or tasks. For the purposes of defining the DECnet–VAXmate programming interface, a distinction is made between the client task, which initiates a connect request, and the server task, which waits for and accepts or rejects the connection.

A client task is the program which initiates a connect request with another task. A server task is the program which waits for and accepts or rejects the pending connect request.

Once a logical link is established, the client and server tasks have a peer-to-peer relationship. The operations performed on their respective sockets are symmetrical. Either task can act as the source or receiving task and can send and receive data, or terminate the logical link at any time.

1.1.2 Sockets

The basic building block for DECnet–VAXmate communication is the socket: an addressable endpoint of communications within a program or task. A task uses the socket to send and receive data to and from a similar socket in another task. Figure 1–2 illustrates the use of sockets within DECnet–VAXmate tasks.

DECnet–VAXmate supports stream sockets and sequenced packet sockets. Stream sockets cause bytes to accumulate until internal DECnet buffers are full. The receiving task does not know how many bytes were sent in each write operation. Sequenced sockets cause bytes to be sent immediately. The receiving task receives those bytes in one "record".

A DECnet–VAXmate program can detect any potential problems by polling the socket's status or by receiving error status in response to network requests.

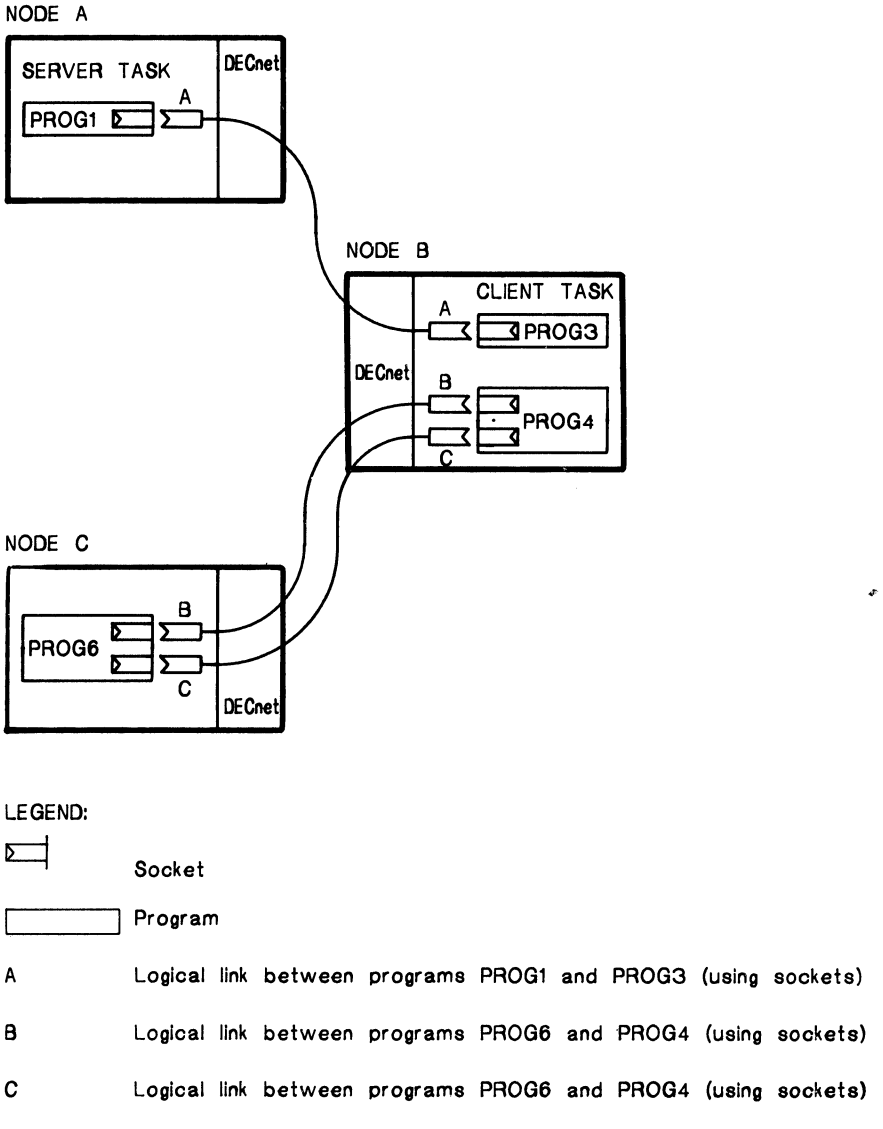


Figure 1–2: Sockets: Basic Building Blocks for DECnet–VAXmate Intertask Communication

1.1.3 Blocking and Nonblocking I/O Operations

DECnet–VAXmate allows tasks to send and receive data without waiting for the completion of the operation. This mode of operation is called nonblocking I/O. When nonblocking is set, socket operations return to the calling program after the operation has been started, but not necessarily completed. Some operations must be restarted. On the other hand, the default mode for socket operations is blocking I/O. When blocking I/O is set, DECnet–VAXmate does not return control to the calling program until the operation has been completed.

1.1.4 Node Names and Addresses

Each system in a DECnet network has a unique node name and address. A node name can have 1 to 6 alphanumeric characters with at least one alphabetic character. A node address is a binary number which consists of an area number and a node number. An area consists of a group of interrelated nodes. Multiple areas are typically used only in large networks. When initiating a connection with a remote node, you must identify that node with a name or address.

1.1.5 Network Object Numbers and Task Names

Client tasks can specify the server task that they want to communicate with by using network object numbers and task names. Network object numbers range from 1 to 255. Numbers 1 to 127 are assigned to generic network servers. Numbers 128 through 255 are available for user-written tasks. When a user specifies a task name, the object number must be zero.

The server task must be installed as a network task on the remote node. In the context of DECnet–VAXmate, the server task declares his network object number and task name with the *bind* function call.

1.1.6 Access Control Information

Access control information contains arguments that define your access rights at the remote node. It consists of three character strings: user ID, password, and account number. Access control verification is performed according to the conventions of the destination system. For some systems, the access information is the log-in data used by the client program.

1.1.7 Optional User Data

DECnet–VAXmate allows up to 16 bytes of optional user data to be exchanged between tasks when connecting to or disconnecting from logical links.

1.2 Forms of Task-to-Task Communication

DECnet–VAXmate supports two forms of task-to-task communication: transparent and nontransparent. Transparent communication provides a subset of the functions used by a program to exchange data with other network programs. Nontransparent communication allows you to use the full range of DECnet–VAXmate task-to-task communication.

1.2.1 Transparent Communication

DECnet–VAXmate transparent communication provides C and assembly language tasks with the basic functions to communicate over the network. These tasks perform standard MS–DOS system I/O operations. This form of I/O lets you move data with little concern for the underlying DECnet interface.

The DECnet–VAXmate transparent functions include the initiation and establishment of a logical link, the orderly exchange of messages between both tasks, and the controlled termination of the communication process. Chapter 3 discusses the MS–DOS function requests that support transparent task-to-task communication.

Figure 1–3 shows the series of events that occurs during transparent task-to-task communication.

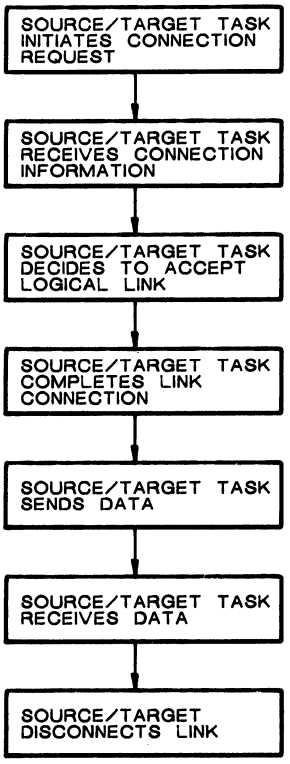


Figure 1–3: Transparent Task-to-Task Communication

1.2.2 Nontransparent Communication

DECnet–VAXmate nontransparent communication provides the same functions as DECnet–VAXmate transparent communication plus additional system and I/O functions. For example, you can use network protocol features such as optional user data on connects and disconnects and out-of-band messages.

DECnet–VAXmate allows you to create C and assembly language programs that use nontransparent communication functions. A C program should use the socket interface calls to perform DECnet functions. These calls are detailed in Chapter 4 of this manual. A set of DECnet utility functions is also provided with DECnet–VAXmate. These functions are used for accessing the network node database and manipulating the data. The DECnet utility functions are detailed in Chapter 5 of this manual.

An assembly language program uses the MS–DOS interrupt function request 6EH to request network process access. Information regarding I/O operations is passed with this MS–DOS interrupt function request. The information is contained in an I/O Con-

trol Block (IOCB) data structure. Chapter 6 details how to create a DECnet–VAXmate program using the Assembly language network process interface calls. It also details the set of calls used for assembly language programs.

1.3 Task-to-Task Communication Functions

This section describes the functions that the client and server tasks request to communicate with each other. Illustrations that appear in this section use the socket interface calls to show task-to-task communication capabilities.

1.3.1 Establishing a Logical Link

The creation of a logical link is a cooperative venture. Two tasks must agree to communicate before you can have an established logical link. The process of establishing a logical link is detailed in Figure 1–4. A logical link connection is required before data can be exchanged between two tasks.

To begin the process, the server task creates a socket supported by DECnet. When this socket is first created, it has no assigned name or number. An object name or number is assigned to the socket. The name or number is required for use in future listening operations. The socket declares itself as a server which is available for client connections.

In turn, the client task must create a socket supported by DECnet. The client task can set up access control information and/or optional connect data. (See Sections 1.1.6 and 1.1.7 for an explanation of each.) The system returns an integer value called a socket number. Subsequent DECnet–VAXmate function calls on this socket will reference the associated socket number. At this point, the client task requests a logical link connection to another task. Any optional user data and/or access control information is sent along with the connection request.

The server task can define how it accepts or rejects an incoming connection request. There are two options:

- The server task can immediately accept the connection request. In this case, any optional user data and/or optional access control information is not used by the task. A logical link has successfully been established between the two tasks. Another socket is also created for the server task. Using the new socket, the server task can exchange data with the client task. The original socket remains open for the server task to listen for other incoming connection requests.
- For nontransparent communication only, the server task can defer making a decision about accepting or rejecting the incoming connection request. When the deferred option is set, the server task can examine any optional user data and/or optional access control information. It can then send optional user data with an acceptance or rejection message to the client task. The client task then retrieves the optional user data and/or status message.

If the connection fails, another attempt can be made at establishing a logical link. If the connection is successful, the logical link is established and another socket is created for the server task. Using the new socket, the server task can exchange data with the client task. The original socket remains open for the server task to listen for other incoming connection requests.

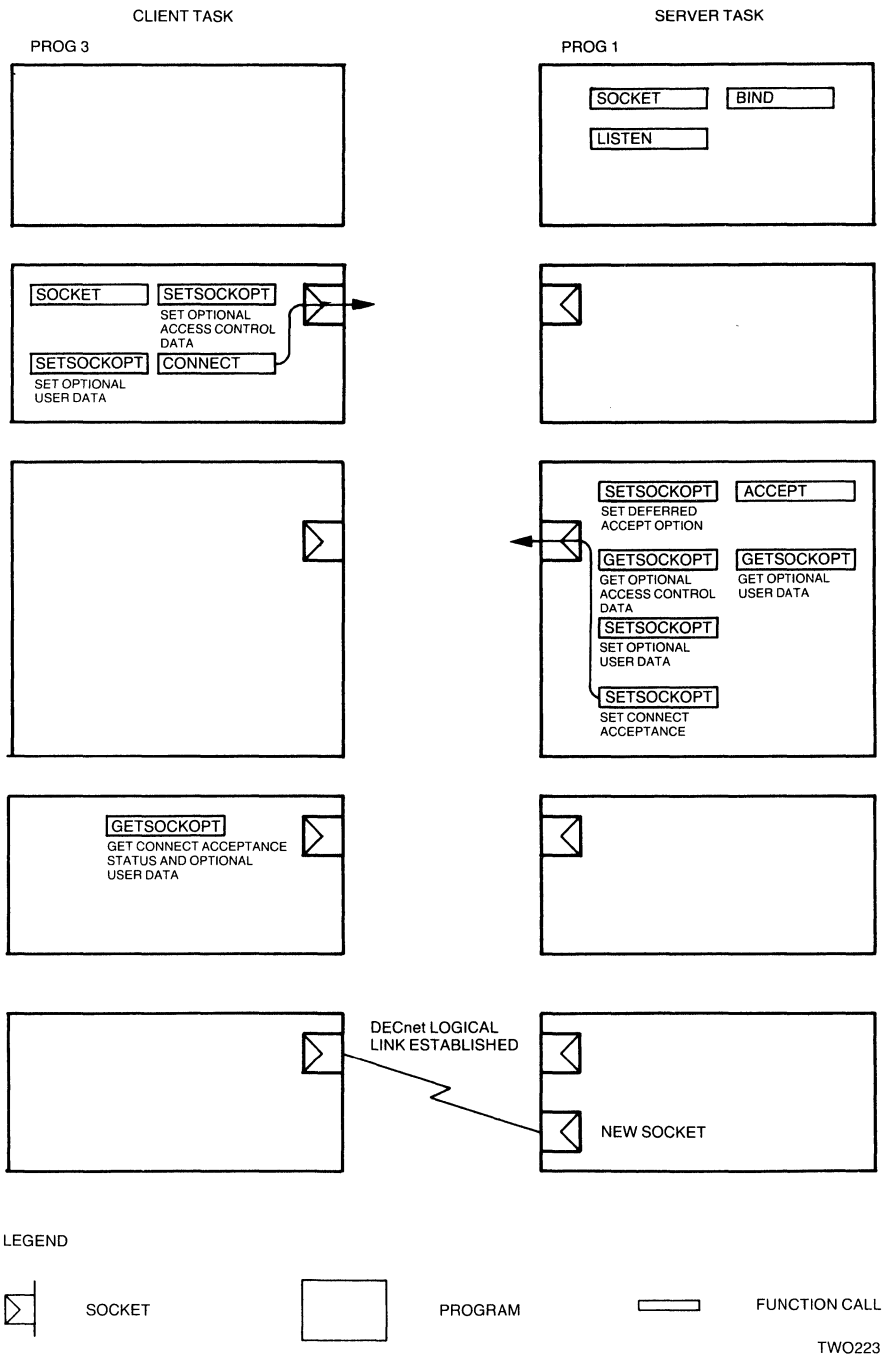


Figure 1–4: Establishing a DECnet Logical Link

1.3.2 Sending Normal Data Messages

Either task can send normal data to its peer. It must specify the socket used for transmitting the message, the buffer containing the outgoing message, and the buffer size. If the socket is set to blocking I/O, and no buffer space is available to hold the outgoing message, the transmission is blocked until resources are freed up. If the socket is set to nonblocking I/O, an appropriate error message is returned to the user.

When the asynchronous form of the *SEND* call is used, control returns to the calling program immediately after the DECnet network process records the request. The network process may complete the request immediately or wait for a later time. To check to see if the data was sent, you can either use a callback routine or poll for status. Section 6.7.15 details the asynchronous form of the *SEND* call.

Figure 1–5 shows normal data being sent over a logical link.

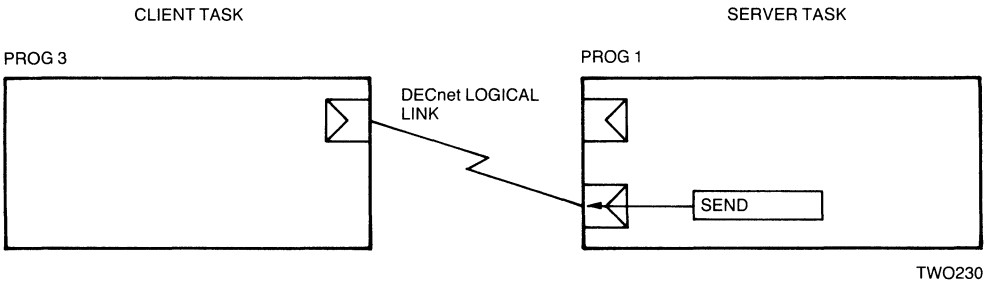


Figure 1–5: Sending Data Messages over a DECnet Logical Link

1.3.3 Receiving Normal Data Messages

Either task can receive normal data from its peer. It must specify the socket used for receiving the message, the address of the buffer which will store the incoming message, and the buffer size. If the socket is set to blocking I/O, and no messages are received, the task waits for a message to arrive. If the socket is set to nonblocking I/O, and no data is ready to be received, an appropriate error message is returned to the user.

When the asynchronous form of the *RCVD* call is used, control returns to the calling program immediately after the DECnet network process records the request. The network process may complete the request immediately or wait for a later time. To check to see if the data was received, you can either use a callback routine or poll for status. Section 6.7.12 details the asynchronous form of the *RCVD* call.

Figure 1–6 shows data being received over a logical link.

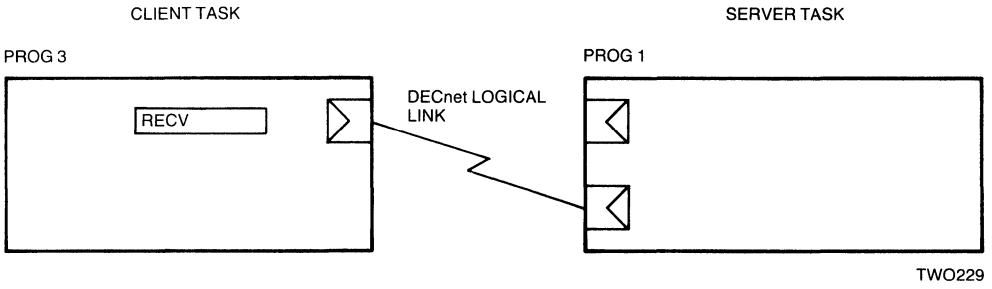


Figure 1–6: Receiving Data Messages Over a DECnet Logical Link

1.3.4 Out-of-Band Messages

Out-of-band messages are unsolicited, high priority messages sent between nontransparent communication tasks over a logical link. An out-of-band message usually informs the receiving task of an unusual or abnormal event in the sending task. The valid range for the message size is 1 to 16 bytes.

Out-of-band messages are always sent ahead of outstanding normal messages. Unless certain error conditions exist, an out-of-band message is always sent even if the socket is set to blocking or nonblocking I/O.

1.3.5 Terminating Network Activity and Closing the Logical Link

The process of terminating network activity can begin with either the client or server task. To initiate close down for nontransparent communication, either task can send up to 16 bytes of optional disconnect data to the other task. The optional disconnect data is sent with an abort or disconnect option.

Figure 1–7 provides an example of how sockets are deactivated and the logical link connection is broken. The close-down steps, as depicted in Figure 1–7, include:

- The server task sets up optional disconnect data.
- The server task deactivates its socket used for exchanging data.
- The server task can also deactivate the original socket on which it listened for incoming connection requests.
- The client task retrieves any optional disconnect data. At some point in time, the task should also deactivate its original socket.
- The logical link connection is broken.

Hence, the sockets are reclaimed as network resources which are made available for future assignment to this task or another task.

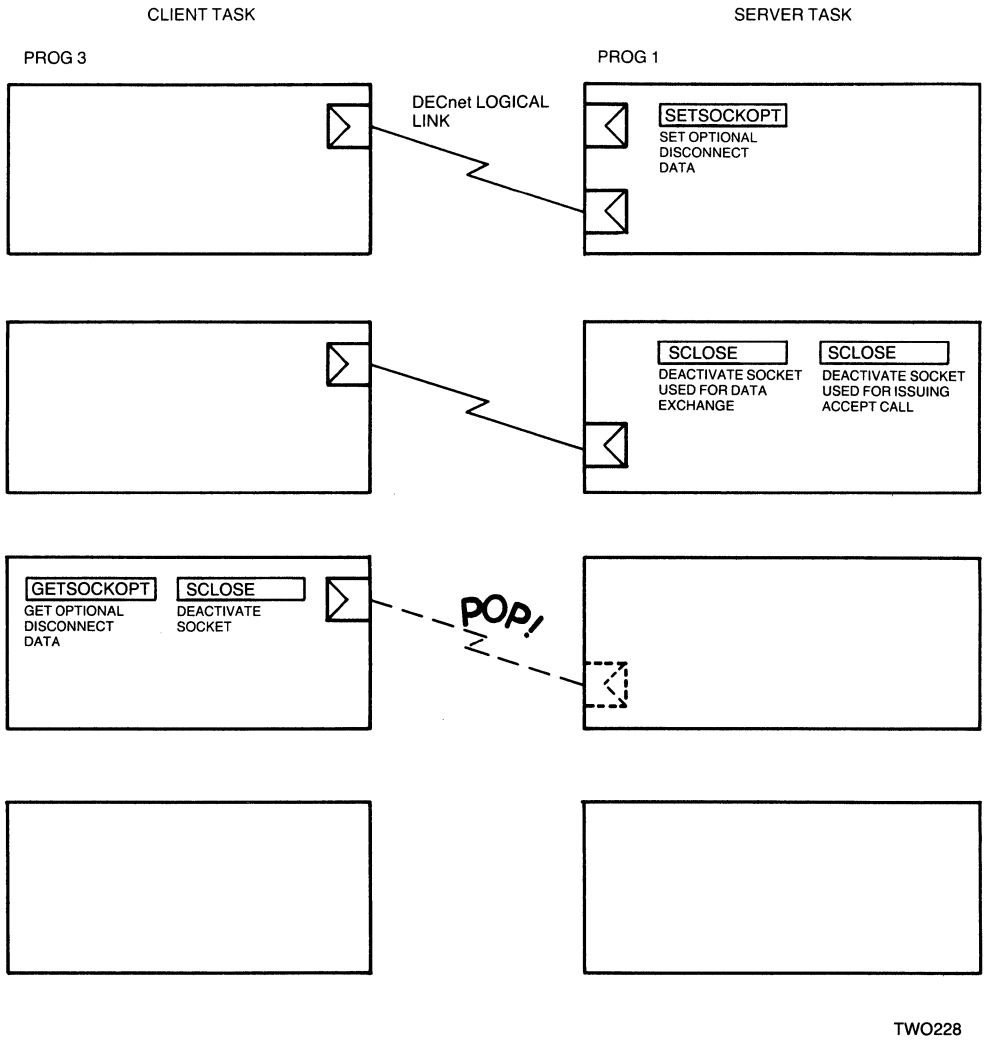


Figure 1–7: Closing Down the DECnet Logical Link Connection

1.4 Transparent File Access

Using DECnet, programs in one node can transparently access a file in another node. Transparent file access enables user programs to perform standard MS–DOS system I/O operations. This form of I/O allows you to move data with little concern for the underlying DECnet interface.

To perform transparent access operations on remote files, use the MS–DOS function requests detailed in Chapter 2.

2
Transparent File Access Operations

Using DECnet, a program in one node can transparently access a file in another node. Transparent file access enables user programs to perform standard MS–DOS system I/O operations. This form of I/O allows you to move data with little concern for the underlying DECnet interface.

Transparent file access functions allow you to access files on a rmote node. You can open and close a file, create or delete a file, and read from or write records to a file on a remote node. Transparent file access also allows you to submit a batchjob to a remote node and search a directory on a remote node for a specific file or files.

NOTE

The VAXmate MS–NET software provides transparent access to files on remote nodes differently than the Transparent File Access (TFA) utility described in this chapter. See the *VAXmate Technical Reference Manual* for more information about TFA and the MS–NET networking environment.

2.1 Introduction to Transparent File Access

In the context of transparent file access, the program that requests remote file access is called the client program. At the remote node, the DECnet system program that receives the request is called the server program. For DECnet–VAXmate, the client program is a user-written program. The server program is a form of the File Access Listener (FAL). FAL receives remote file access requests from the network. FAL completes connections initiated by remote accessing user programs and translates them into calls to the file system at the remote node. FAL then sends or receives the resulting file data back to the accessing program. At the client node, special routines reformat the data to make it conform to local or remote file structures (depending upon I/O direction).

Before accessing any remote files, you must install the Transparent File Access (TFA) utility. TFA can be installed at every boot time (see the *DECnet–VAXmate Installation Guide* for installation details) or installed by typing the following start-up command line:

```
E> TFA (RET)
```

The system responds with one of the following messages:

```
DECnet - TFA Version 1.1 installed -
```

or

```
DECnet - TFA Version 1.1 has already been installed -
```

Once the utility is installed, you can request network access by including a subset of MS–DOS system calls in your program. (Refer to Table 2–3.) These calls activate Transparent File Access Routines (TFARs). The TFARs build, send, receive and interpret DECnet file access messages. These messages are defined by the Data Access Protocol (DAP). DAP messages control the execution of remote file access and outline procedures to accomplish specific file operations.

A user program does not handle remote file access operations directly. DECnet–VAXmate includes system software that sends and receives DAP messages on behalf of user programs. DAP-speaking TFARs at the local node exchange DAP messages with the DAP-speaking server FAL at the target node.

2.2 Initiating Transparent File Access

Transparent file access operations require an extended handshake sequence being performed at the beginning of the operation. This extended handshake occurs when the user program issues an initial file access call (for example, open, create, delete, submit, and so forth) to a remote file. The handshake sequence includes the TFARs setting up the logical link connection and exchanging file access messages to initialize the DAP environment for pending file operations.

The initial access to the remote file passes the following information to TFARs in the local file system:

- Remote file name string
- Access control information

2.2.1 Remote File Name

The file name string specifies the remote node name and the file on that node to be accessed. It includes the device, directory, file name, extension, and version number of the remote file. Since the remote file system actually carries out the requested file operation, you must be familiar with the conventions used for identifying files at the remote node.

2.2.2 Access Control Information

Access control information provides access rights to the remote system. It consists of a user identification name, a password associated with the user identification, and additional accounting information required by the remote system. To supply access control information, you must follow the syntax detailed in Section 2.7.1 of this chapter.

2.3 File Characteristics

A file has specific characteristics that determine how the file is transferred between local and remote systems.

The following sections discuss the effects of file characteristics on remote input and remote output files. A remote input file is a file located on a remote DECnet host that is read by an MS–DOS system. A remote output file is a file located on a remote DECnet host that is written to by an MS–DOS system.

2.3.1 File Attributes

File attributes for remote input files are determined by the remote file and the remote file system. For example, an ASCII file cannot be redefined as an image file.

File attributes for remote output files are determined by the format of the local file and the type of remote file system.

The set of file attributes are:

- **File organization.** DECnet–VAXmate supports sequential files only. A sequential file has records arranged one after the other. The first record written is the first record in the file, and the record written most recently is the last record in the file.
- **Data type.** A remote file can have either an ASCII or an image data type. Depending on the file's record attribute, record format, and the remote file system type, DECnet–VAXmate can reformat ASCII data. On the other hand, DECnet–VAXmate cannot convert image data type files.
- **Record format.** This particular file attribute indicates how records are formatted within the file. A record can have one of the following formats:

  Undefined records have no declared formats.

  Stream records consist of a continuous series of ASCII characters delimited by carriage return/line feed pairs.

  Fixed-length records are identical in size.

  Variable-length records can be different lengths, up to a maximum size that you specify. The maximum size is fixed at file-creation time and cannot be changed for the life of the file.

Variable-with-fixed-control (VFC) records include a fixed-length control field that precedes the variable–length data portion. This format allows you to construct records with additional data that labels the contents of variable-length portion of the record.

- **Record attributes.** This characteristic indicates how data is formatted within a record. Record attributes (RATs) include implied carriage return/line feed pairs, embedded carriage control, FORTRAN carriage control, print file carriage control, line sequence ASCII, block, and MACY11.
- **Fixed size.** If the record format of the remote input file is VFC, this characteristic defines the size of each fixed length header.
- **Maximum record size.** If the record format is fixed, maximum record size (MRS) defines the length of each record. For fixed-length records, the default size is 128 bytes. For variable-length records, MRS declares the maximum record size. There is no default size.

2.4 Performing Data Conversions

To read or write files successfully on other nodes, data must be formatted according to the conventions of the respective file system. A system running MS–DOS supports stream formatted files. Remote DECnet hosts running heterogeneous operating systems can support stream and nonstream file systems: variable-length, fixed-length and VFC records.

The following tables summarize file structure interdependencies when you read or write files on remote systems.

Table 2–1: Remote Input File Transfers

File Type	Remote System Type	Record Attributes	Processing Mode
Image	Not applicable	Ignored	No conversion.
ASCII	Stream	Ignored	No conversion with embedded carriage control.
ASCII	Nonstream	Implied CR/LF pair	CR/LF pair appended to each record.
ASCII	Nonstream	FORTRAN, Print, LSA carriage control.	Conversion of carriage control done.
ASCII	Nonstream	Null, Embedded, block, MACY11	No conversion.

Table 2–2: Remote Output File Transfers

File Type	Remote System Type	Record Attributes	Processing Mode
ASCII	Stream	None	No conversion. Records determined by LFs.
ASCII	Nonstream	Variable, implied CR/LF	New line and CR characters are dropped. Records determined by LFs.
Image	Stream	None	No conversion.
Image	Nonstream	Fixed: record size = 128 bytes.	No conversion. Last record is padded if necessary.

The following sections detail how the TFARs handle data conversions for remote input and output files.

2.5 Converting Remote Input Files

During remote file input, a file located on a remote DECnet host is transferred to an MS–DOS system. Local handling of the remote file is determined by its attributes and the type of remote file system.

2.5.1 Binary Image Files

No record to stream conversion is performed on a binary image file. The file is passed to the calling task one record at a time. The size of each record is set by the record attributes and the maximum record size of the remote input file. If the remote file system does not support record attributes (for example, stream), the size is set to 128 bytes.

The remote file's record format (RFM) and record attributes (RATs) are ignored in this type of file transfer.

2.5.2 ASCII Files

No data interpretation is performed on an ASCII file coming from a stream file system. For ASCII files being copied from a nonstream file system, data handling is determined by the remote file system, the remote file's record format and record attributes.

A nonstream file system can support variable-length, fixed-length, variable-with-fixed-control (VFC) record formats, and stream record formats. It also supports a number of record attributes. The TFARs perform data conversion on ASCII files depending on the record attributes of the remote file. The following section describes how TFARs interpret data based on the remote file's record attributes:

- **RAT = Implied LF/CR.** No conversion of embedded carriage control characters. A CR/LF pair is appended to each record.

- **RAT = FTN, PRN.** FORTRAN (FTN) and Print (PRN) carriage control characters are converted appropriately to stream file systems.
- **RAT = NULL, Embedded carriage control, Block, MACY11.** No data is inserted between records.

2.6 Converting Remote Output Files

Remote output files are transferred from the local system to a remote DECnet host. By default, files are transferred in ASCII mode. The file's record format and record attributes are determined by the remote file system. A logical record consists of data up to and including a CR/LF pair. The TFARs send a file as image if the first logical record is not terminated by either a CR/LF or a LF/CR pair.

2.6.1 ASCII Files

If the local file is transferred to a remote stream file system, the logical records are passed with no data conversion being performed. The file structure defaults are:

Record Format	Record Attribute
Stream	NONE

If the file is copied to a nonstream file system, delimiting CR/LF pairs are dropped from the logical records before they are sent. The file structure defaults are:

Record Format	Record Attribute
Variable length	Implied CR/LF

2.6.2 Image Files

If the remote output file is an image file, the records are passed with the following default file structure:

Record Format	Record Attribute	Maximum Record Size (bytes)
Fixed-length	NONE	128

Since the default data type for a remote output file is ASCII, the TFARs initially create an ASCII file on the remote node. If the first logical record is not terminated by a CR/LF or a LF/CR pair, the TFARs will issue a DAP access message complete (purge) message. In this case, the TFARs create a new image file on the remote node. This particular file is then sent using the data handling defaults for an image file transfer.

2.7 Using Network File Specifications for Network Access

MS–DOS system calls requesting network access must pass a specifically formatted network file specification string. The TFARs intercept these calls, recognize the network access request and perform the DAP functions necessary to complete the specified network operation.

The network file specification consists of a node specification string, optional access control information, and a file name specification string.

To request network access, use the network file specification string as shown below:

\\f\node\userid\password\account\\file-specification

2.7.1 Node Specification

A node specification identifies the remote system where file access operations will take place. The node specification string is preceded by two backslashes, the letter f and another backslash. The optional access control string follows the node information. Each element is separated by a single backslash character.

If no access control information is supplied, then the default access control information set with NCP is used. If there is no default access information, only the node name is used by the TFAR subroutines for processing.

The node specification string uses one of the following formats:

1. To access the remote node with access control information:

   \\f\node\userid\password\account

2. To access the remote node without optional access control data:

   \\f\node

where:

f	specifies that a data file will be accessed for this TFA transaction.
node	specifies either the name or address of the remote node. A node name has a maximum of 6 alphanumeric characters with at least one alphabetic character. A node address is a numeric string including the area number in the range of 1 to 63, and the node number in the range of 1 to 1023.
userid	identifies a user name or log–in ID on the remote system. The user name and password set the user's privileges for accessing the remote task. A user name has a maximum of 39 alphabetic characters.
password	defines a user's password which is associated with *user*. A user's password has a maximum of 39 alphabetic characters.

account identifies a billing account number which is used with the user name and password information on some systems. An account number has a maximum of 39 characters. If the account information is not required, you can omit it from the string.

2.7.2 File Name Specifications

The file name specification string identifies the remote file to be accessed. File specification strings cannot contain spaces, semi-colons, left (<) or right (>) angle brackets. File names must conform to the conventions of the target node. Any unspecified elements default to the target system's conventions. Refer to the appropriate programmer's reference manual for more details.

2.8 Passing User Parameters

Whenever an I/O file operation is invoked, system control is transferred to the TFARs. Users pass parameters to and receive results from the TFARs by way of the MS–DOS function requests. The parameters are loaded into or returned to one or more 8086/8088 registers. The contents of each register is defined by the specific MS–DOS function call.

The TFAR subroutines check to see if the proposed I/O operation is a network supported call. Network access is signaled by the string \\f\ which begins the network file specification string. (See Section 2.7.)

The TFARs parse the network file specification string. If a function call returns a handle, the TFARs save the handle (and related file specification data) in a database for subsequent calls requiring network access over the same path. Once the requested DAP operation is completed, control is returned to the system and the TFARs.

2.9 Using the Transparent Network Task Control Utility

The Transparent Network Task (TNT) Control utility, Version 1.1, reports the status of the Transparent File Access (TFA) utility as well as the Transparent Task-to-Task (TTT) utility. It features an on-line help routine which lists supported TNT commands. TNT returns DAP messages and other extended error information for assisting in fault isolation. Using TNT, you can deinstall TFA (and/or TTT) from memory.

This section deals only with the use of TNT for transparent file access operations. Chapter 3 discusses TNT and its role in transparent task-to-task communication.

2.9.1 Displaying Network Status of the Transparent File Access Utility

To display the status of TFA, you run TNT. The system responds with a start-up message and one or more network status message(s).

All errors returned by the Transparent File Access utility are standard MS–DOS error messages. However, the Transparent Network Task Control utility provides extended error support to transparent file access operations. DAP and other extended error messages, returned by this utility, can help you locate problem areas. See Appendixes C, E and F for a complete list of these error messages.

NOTE

When you run TNT, the status of the Transparent Task-to-Task utility is also reported.

To invoke TNT, type the following command:

```
E>TNT (RET)
```

The system responds with a start-up message:

```
Transparent Network Task Control V1.1
```

and one or more of the following status message(s):

```
DECnet TFA is not installed.

DECnet TFA has no errors to report.

DECnet TFA Errors are:
remote-file-specification: extended-error-message
.
.
.
```

or

```
DECnet TTT is not installed.

DECnet TTT has no errors to report.

DECnet TTT Errors are:
remote-file-specification : extended-error-message
.
.
.
```

where

extended-error-message returns an error code from one of the following groups of error messages:

- Error codes contained in the external variable *errno* — Appendix C.
- DAP error messages (maccode/miccode in octal) — Appendix E.
- Transparent File Access Routines error messages — Appendix F.

2.9.2 On-line Help

On-line help provides you with a list of supported TNT commands. To obtain help, type:

```
E> TNT HELP (RET)
```

The system responds with the following help text:

```
Transparent Network Task Control V1.1
Transparent Network Task commands are:

TNT                         Display status of both TTT and TFA.
TNT HELP                    Display this text.
TNT TTT OFF                 Remove TTT from memory.
TNT TFA OFF                 Remove TFA from memory.
```

If you mistype a command, TNT responds with an error message and the list of supported TNT commands:

```
Transparent Network Task Control V1.1
Transparent Network Task command error.
Transparent Network Task commands are:

TNT                         Display status of both TTT and TFA.
TNT HELP                    Display this text.
TNT TTT OFF                 Remove TTT from memory.
TNT TFA OFF                 Remove TFA from memory.
```

2.9.3 Deinstalling TFA

You can remove TFA from memory. Enter the following command line:

```
E> TNT TFA OFF (RET)
```

The system responds with the following text:

```
Transparent Network Task Control V1.1
The task was removed successfully.
```

If TFA could not be removed, one of the following messages is displayed:

```
Transparent Network Task Control V1.1
TFA cannot be removed because it is not installed or is not installed
last.
```

or if MS–DOS failed on the remove call,

```
Transparent Network Task Control V1.1
The task could not be removed.
```

NOTE

TFA traps MS–DOS interrupt function call 21H as do other software applications. If you want to remove TFA from memory, it must be the last task installed which intercepts interrupt 21H. Otherwise, you should remove any tasks installed after TFA that also trap 21H, or reboot your system to remove TFA.

2.10 TFA Programming Considerations

There are specific MS–DOS function requests that support DECnet–VAXmate transparent file access operations. Table 2–3 provides you with a summary of these calls. When creating TFA applications, you should note the following:

- Some user programs may not accept the TFA network specification string.
- You should not use unsupported MS–DOS function requests to perform transparent file access operations.
- If you issue a CTRL/C while TFA is active, network operation may be blocked. To clear this condition, run the TNT utility.

2.11 MS–DOS Function Requests

The following table summarizes the MS–DOS function requests used for transparent file access operations. Call descriptions appear in Sections 2.11.1 through 2.11.9.

Table 2–3: MS–DOS Function Requests for Transparent File Access

Hexadecimal Value	Function	Network Access
3CH	Create a file.	Initiate a logical link request to create a remote file.
3DH	Open a file.	Initiate a logical link request to open a remote file.
3EH	Close a file handle.	Close a remote file and terminate a logical link connection.
3FH	Read from a file/device.	Read data from a remote file.
40H	Write to a file/device.	Write data to a remote file.
41H	Delete a file from a specified directory.	Delete a file from a remote directory.
4BH	Load and execute a program.	Submit a remote command file to be executed.
4EH	Find first matching file.	Search for the first remote file that matches the specified file characteristics.
4FH	Find next matching file.	Find the next file entry that matches the name specified on the previous find first call.

2.11.1 Close

NAME

Close – close a remote file, terminate a logical link connection and deactivate the handle used for data exchange.

On Entry	8086/8088 Register Contents
AH	3EH
BX	Handle for logical link access.

On Return	8086/8088 Register Contents
AX	No errors (if carry bit is clear)
	Error code (if carry bit is set)

Figure 2–1: Close Function Call

DESCRIPTION

The *Close* function closes the remote file, terminates the logical link connection, and deactivates the handle used for data exchange.

On entry, the BX register contains the 16-bit handle value returned by the open or create I/O operation. If the close operation completes successfully, no error is returned in the AX register. If an error condition occurs, the appropriate error code is returned in the AX register.

The following error code can occur:

Hexadecimal Value	Meaning
6	An invalid handle value was detected.

NOTE

If you issue the *Close* call with a handle equal to –1, the TFARs will interpret the call as a request to abort any active Find Matching File operations.

2.11.2 Create

NAME

Create – initiate a logical link request to create a remote file.

On Entry	8086/8088 Register Contents
AH	3CH
DS:DX	Address of remote file specification string.

On Return	8086/8088 Register Contents
AX	Handle for logical link access (if carry bit is clear) Error code (if carry bit is set)

Figure 2–2: Create Function Call

DESCRIPTION

The *Create* function call enables a source task to initiate a logical link request to create a remote file. When the request is made, the file is opened for write operations. On entry, DS:DX contains the address of the remote file specification string. Any optional access control information is passed as part of the string to the target task.

On return, the AX register contains an error code or a 16-bit handle associated with the source task. The returned handle value must be used for subsequent read and write I/O operations.

The TFARs exchange a series of DAP messages with the remote FAL in order to initialize the DAP environment and define the requested network access. Each link initialization involves an exchange of DAP configuration, attributes, and access messages. The configuration messages include information regarding the operating and file systems of the source and target systems, and the buffer size. The attributes messages supply information about the file to be accessed. Undefined file attributes are set to default values determined by the remote file system. The access message establishes the type of access to the remote file.

If you are unable to initiate a logical link connection, an error code is returned in the AX register. The following set of error codes can occur:

Hexadecimal Value	Meaning
2	The network process may not be loaded. The node name to node address mapping is not found in the database file. The target task on the outgoing connection is not available. The network is unreachable.
3	The target task was not found.
4	There are too many active logical link connections.
5	The remote object rejected the request.

2.11.3 Delete

NAME

Delete – delete a remote file from a specified directory.

On Entry	8086/8088 Register Contents
AH	41H
DS:DX	Address of network file specification string

On Return	8086/8088 Register Contents
AX	Error code (if carry bit is set)

Figure 2–3: Delete Function Call

DESCRIPTION

The *Delete* function call deletes a remote file from a specified directory.

On entry, DS:DX contains the address of the remote file specification string. Any optional access control information is passed as part of the string to the target task.

If an error condition occurs, the error code is returned in the AX register. The following set of error codes can occur:

Hexadecimal Value	Meaning
2	The network process may not be loaded. The node name to node address mapping is not found in the database file. The target task on the outgoing connection is not available. The network is not reachable.
5	The remote object rejected the request.

2.11.4 Find First Matching File

NAME

Find First Matching File – search for the first remote file that matches the specified file characteristics.

On Entry	8086/8088 Register Contents
AH	4EH
DS:DX	Pointer to remote directory specification string

On Return	8086/8088 Register Contents
AX	Error codes (if carry bit is set)
current DMA data block	File name, creation date, file size and creation time (if carry bit is clear)

Figure 2–4: Find First Matching File Function Call

DESCRIPTION

The *Find First Matching File* function searches for the first file that matches the directory specification set by the user. If a directory specification is given without a file specification or includes wildcards, the first matching file is returned. On entry, the DS:DX register contains the address of the network file specification string.

If a file is found that matches the specification string, the carry bit is cleared and the information is returned into the current DMA data block as follows:

- File name bytes 30–42
- Creation time bytes 22–23
- Creation date bytes 24–25
- File size bytes 26–29 (26–27 low, 28–29 high)

The DMA is a portion of memory that is allocated for passing data between processes. The user can obtain a pointer to this area by issuing an MS–DOS *Get Disk Transfer Address* function call 2FH.

If only one matching file is found, the carry bit is **not set** and an 18 is returned in the AX register. If no matching file is found, the carry bit is **set**, and an error code of 2 is returned in the AX register.

The following set of error codes can occur:

Hexadecimal Value	Meaning
2	The network process may not be loaded. The node name to node address mapping is not found in the database file. The target task on the outgoing connection is not available. The network is unreachable.
18	There are no matching files.

2.11.5 Find Next Matching File

NAME

Find Next Matching File – find the next file entry that matches the name specified on the previous find first call.

On Entry	8086/8088 Register Contents
AH	4FH
current DMA address	must point to a data block returned by the previous Find First Matching File function call

On Return	8086/8088 Register Contents
AX	error codes; if carry bit is set

Figure 2–5: Find Next Matching File Function Call

DESCRIPTION

This function call finds the next matching entry in a directory. On entry, the current DMA address must point to a data block returned by the previous *Find First Matching File* function call.

If a file is found that matches the specification string, the information is returned into the current DMA data block as follows:

- File name bytes 30–42
- Creation time bytes 22–23
- Creation date bytes 24–25
- File size bytes 26–29 (26–27 low, 28–29 high)

The DMA is a portion of memory that is allocated for passing data between processes. The user can obtain a pointer to this area by issuing an MS–DOS *Get Disk Transfer Address* function call 2FH.

When a matching file is found, and it is the last matching file in the directory, an error code 18 is returned in the AX register and the carry bit is cleared. Likewise, if no matching file is found, the carry bit is set and an error code 18 is returned in the AX register.

The following error code can occur:

Hexadecimal Value	Meaning
18	There are no more files.

2.11.6 Load and Execute a Program

NAME

Load and Execute a Program – submit a remote command file.

On Entry	8086/8088 Register Contents
AH	4BH
DS:DX	pointer to remote file specification string

On Return	8086/8088 Register Contents
AX	error code; if carry bit is set

Figure 2–6: Load and Execute a Program Function Call

DESCRIPTION

This function call allows an existing remote file to be submitted as a batch or command file for remote execution. (The remote file is not deleted after completion of this call.) On entry, the DS:DX register contains the address of the network file specification string. This string points to the name of the remote file to be loaded and executed. The ES:BX register points to a parameter block defining the command file's environment. These registers are ignored for remote command file submission.

If the load and execute is unsuccessful, the carry bit is set and the error reason is returned in the AX register. The following error code can occur:

Hexadecimal Value	Meaning
2	The network process may not be loaded. The node name to node address mapping is not found in the database file. The target task on the outgoing connection is not available. The network is unreachable.

2.11.7 Open

NAME

Open – initiate a logical link request to open a remote file.

On Entry	8086/8088 Register Contents
AH	3DH
DS:DX	Address of remote file specification string
AL	Access mode

On Return	8086/8088 Register Contents
AX	Handle for logical link access (if carry bit is clear)
	error code (if carry bit is set)

Figure 2–7: Open Function Call

DESCRIPTION

The *Open* call enables a task to initiate a logical link request to open a remote file. On entry, DS:DX contains the address of the remote file specification string. If you include wildcards as part of the specification string, only one file is opened.

Any optional access control information is passed as part of the string to the target task.

The access mode is defined in AL and consists of one of the following values:

- 0 open file for reading
- 1 open file for writing
- 2 open file for reading and writing

If the *Open* call completes successfully, a 16–bit handle is returned in the AX register. The handle value must be used for subsequent read and write I/O operations.

If an error condition occurs, the carry bit is set and an error code is returned in the AX register.

The following set of error codes can occur:

Hexadecimal Value	Meaning
3	The target task was not found.
4	There are too many active logical link connections.
5	Network access was denied.

2.11.8 Read

NAME

Read – receive data from a remote file.

On Entry	8086/8088 Register Contents
AH	3FH
DS:DX	Address of network message buffer
CX	Size of network message buffer
BX	Handle for logical link access

On Return	8086/8088 Register Contents
AX	Number of bytes received over the logical link (if carry bit is clear) Error codes (if carry bit is set)

Figure 2–8: Read Function Call

DESCRIPTION

The *Read* function call allows the target task to read data from a remote file.

On entry, the BX register contains the 16-bit handle value. The CX register contains the number of bytes to be received. DS:DX contains the address of the network message buffer.

On return, the AX register contains the number of bytes successfully received by the target task. If the carry bit is clear and AX = 0, an end-of-file status is indicated. A single read function returns a maximum of one logical record.

If the buffer is too small for one logical record, no error occurs. The next read continues to return bytes until the entire logical record has been read.

If an error condition occurs, the carry bit is set, and an error code is returned in the AX register.

The following set of error codes can occur:

Hexadecimal Value	Meaning
5	The logical link was disconnected.
6	An invalid handle was detected.

2.11.9 Write

NAME

Write – write data to a remote file.

On Entry	8086/8088 Register Contents
AH	40H
DS:DX	Address of network message buffer
CX	Size of network message buffer
BX	Handle for logical link access

On Return	8086/8088 Register Contents
AX	Number of bytes sent over the logical link (if carry bit is clear)
	Error codes (if carry bit is set)

Figure 2–9: Write Function Call

DESCRIPTION

The *Write* function call allows the source task to write data to a remote file.

On entry, the BX register contains the 16-bit handle value. The CX register contains the number of bytes to be sent. DS:DX contains the address of the network message buffer.

On return, the AX register contains the number of bytes successfully sent by the source task.

If an error condition occurs, the carry bit is set and an error code is returned in the AX register.

The following set of error codes can occur:

Hexadecimal Value	Meaning
5	Network access was denied.
6	An invalid handle was detected.

3
Transparent Task-to-Task Communication

DECnet–VAXmate supports transparent task-to-task communication for high level language and assembly language programs. DECnet–VAXmate supports DOS Version 2.0 XENIX-compatible I/O handle calls starting with function request 2FH. Using specific calls, a task can perform standard I/O operations and communicate with another task over the network.

3.1 Transparent Task-to-Task Communication

Transparent communication provides the basic functions necessary for tasks to communicate over the network. These functions include the initiation, acceptance, and establishment of a logical link; the orderly exchange of messages between DECnet tasks; and the controlled termination of the communication process.

When accessing the network transparently, you use no DECnet-specific calls to perform these functions. Instead, you use normal I/O statements provided by the applicable high level language. An assembly language task uses a subset of the MS–DOS function requests to perform the same communication activities.

3.2 Transparent Communication Functions

This section describes the functions that the client and server tasks use to communicate
ork.

3.2.1 Initiating a Logical Link Connection

Transparent communication can only take place after a logical link is established between two cooperating tasks. You establish the logical link by issuing a client task call that requests a logical link connection. The request is sent to the server task on the remote node.

The interaction that takes place prior to establishing a logical link is termed a handshaking sequence. Using transparent task-to-task communication, an MS–DOS program can act as either a client or a server.

3.2.2 Handshaking Sequence for a Client Task

To initiate the logical link request transparently, a client task performs a file create or open operation. This task supplies the following information:

- **The identification of the server node.** Every node in the network has a unique identifier that distinguishes it from other nodes in the network. Transparent communication uses a node specification string to indicate the name of the server node. (See Section 3.4.1.)
- **The identification of the server task.** Client tasks specify the server task that they want to communicate with by using a network task specification string. This string uses network object numbers and task names. Network object numbers range from 1 to 255. Numbers 1 to 127 are assigned to generic network servers. Numbers 128 to 255 are available for user-written tasks.

  When a user specifies a task name, the object number is zero.

High level language client tasks can use standard file opening statements to request a logical link connection to the remote task. An assembly language client task uses the MS–DOS *Create* or *Open* function request to perform the same operation.

3.2.3 Handshaking Sequence for a Server Task

A high level language server task performs a file create or open operation to accept the logical link connection request. If *SYS$NET* is specified as the node name, the task is always a server task. An assembly language server task can accept the logical link request with either the MS–DOS *Create* or *Open* function request.

3.2.4 Exchanging Data Messages over a Logical Link

Once the logical link is established, either task can send and receive data messages. A coordinated set of write and read operations enables the exchange of data over the logical link. For high level language tasks, standard read and write calls are used for data exchange. An assembly language task uses the MS–DOS *Read* and *Write* function requests. The handle returned by the previous *Create* and *Open* function requests must be specified in all *Read* and *Write* function requests.

3.2.5 Terminating the Logical Link

The termination of a logical link signals the end of the communication process between two tasks. When network activity is no longer required, either high level language task can issue a file closing statement to break the link. Likewise, either assembly language tasks can issue the MS–DOS *Close* function request to terminate the connection. This particular MS–DOS call closes the logical link and deactivates the original handle used for data exchange.

3.3 Creating a Transparent Communication Task

Before creating a transparent communication task, you must install the Transparent Task-to-Task (TTT) utility. To install this utility, type the following start-up command line:

```
E> TTT (RET)
```

The system responds with either:

```
DECnet - TTT Version 1.1 installed
```

or

```
DECnet - TTT Version 1.1 has already been installed
```

Once the utility is installed, a high level language task can invoke standard I/O function calls. An assembly language task can use the following MS–DOS function requests.

Table 3–1: MS–DOS Function Requests for Transparent Intertask Communication

Function	Network Access
Create/Open	Initiate a logical link request. Accept a logical link request.
Close	Terminate a logical link connection.
Read	Receive data over a logical link.
Write	Send data over a logical link.

Whether you are running a high level or assembly language task, network access requires the use of specially formatted task names. These task names are implemented as network task specification strings. The strings must be specified with all create and open file operation calls.

3.4 Network Task Specifications

The network task specifications consist of a node specification string with optional access control information and a target task specification string. Access control information contains arguments that define your access rights at the remote node. The control string contains three fields: user name, password, and account number. Access control verification is performed according to the conventions of the remote node.

The target task can be identified as either a named or numbered object. Named objects are user-written tasks which are referenced by a name during a connect request and an accept request. The object number for such tasks is 0. Numbered objects are tasks which are referenced by a number. The object numbers range from 1 to 255. Numbers 1 to 127 are reserved for DECnet-specific tasks. Numbers 128 to 255 are available for user-written tasks.

You can access the target task by its object name or number. The network task specification string uses one of the following formats:

1. To access the target task by object name with access control information

   *t**node**userid**password**account**object-name*

2. To access the target task by object name without access control information

   *t**node**object-name*

3. To access the target task by object number with access control information

   *t**node**user**password**account*\\#*object-number*

4. To access the target task by object number without access control information

   *t**node*\\#*object-number*

5. To establish a server task by object name

   *t**SYS$NET**object-name*

6. To establish a server task by object number

   *t**SYS$NET*\\#*object-number*

3.4.1 Node Specifications

A node specification for a client task names the remote node and supplies optional access control data. The node specification string is preceded by two backslashes, the letter t and another backslash. The optional access control string follows the node information. Each element is separated by a backslash.

The node specification string takes the following format:

*t**node**user**password**account*\\

or

*t**node*\\

where:

t	specifies that a program will be accessed for this TTT transaction.
node	specifies either the name or address of the remote node. A node name has a maximum of 6 alphanumeric characters with at least one alphabetic character. A node address is a numeric string including the area number in the range of 1 to 63, and the node number in the range of 1 to 1023.

userid identifies a user name or log-in ID on the remote system. The user name and password set the user's privileges for accessing the remote task. A user name has a maximum of 39 alphabetic characters.

password defines a user's password which is associated with *user*. A user's password has a maximum of 39 alphabetic characters.

account identifies a billing account number which is used with the user name and password information on some systems. An account number has a maximum of 39 characters. If the account information is not required, you can omit it from the string.

A node specification string for a server task is always *t**SYS$NET*\\.

3.4.2 Task Specifications

For a client task, the task specification identifies the cooperating task on the remote system. The server task specification identifies the server task. The task can be specified as a named or a numbered object.

The task specification string is expressed in one of the following formats:

object-name

or

#object-number

where:

object-name specifies the task as a named object. User-written tasks are usually addressed as object type 0 plus a name. Digital-specific tasks can be addressed by object name. The object name has a maximum of 16 characters.

object-number specifies the task as a numbered object. The valid range is 1 to 255.

3.5 MS–DOS Intercept Routine

Whenever an I/O file operation call is invoked, system control is transferred to the MS–DOS task-to-task intercept routine. Network access is signaled by the string *t*\ which begins the network task specification string. (See Section 3.4 for formats.) The MS–DOS intercept routine checks to see if the proposed I/O operation is a network supported call.

The intercept routine parses the network task specification string. It stores away the socket numbers for the handles used with the open and create I/O calls. These values must be specified with subsequent read, write and close operations. Before one of these calls can complete, the intercept routine must verify the current status of the network handles.

3.6 Using the Transparent Network Task Control Utility

The Transparent Network Task (TNT) Control utility, Version 1.1, reports the status of the Transparent Task-to-Task (TTT) utility as well as the Transparent File Access (TFA) utility. It features an on-line help routine which lists supported TNT commands. TNT returns extended error information for assisting in fault isolation. Using TNT, you can deinstall TTT (and/or TFA) from memory.

This section deals only with the use of TNT for transparent task-to-task communication. Chapter 2 discusses TNT and its role in transparent file access operations.

3.6.1 Displaying Status of the Transparent Task-to-Task Utility

To display the status of TTT, you run TNT. The system responds with a start-up message and one or more status message(s).

All errors returned by the Transparent Task-to-Task utility are standard MS–DOS error messages. However, the Transparent Network Task Control utility provides extended error support to transparent task-to-task communication. Extended error messages, returned by this utility, can help you locate problem areas.

NOTE

When you run TNT, the status of the Transparent File Access utility is also reported.

To invoke TNT, type the following command:

```
E> TNT (RET)
```

The system responds with a start-up message:

```
Transparent Network Task Control V1.1
```

and one or more of the following status message(s):

```
DECnet TTT is not installed.

DECnet TTT has no errors to report.

DECnet TTT Errors are:
remote-file-specification: extended-error-message
.
.
.
```

or

```
DECnet TFA is not installed.

DECnet TFA has no errors to report.

DECnet TFA Errors are:
remote-file-specification : extended-error-message
.
.
.
```

where

extended-error-message is a message contained in the external variable *errno*. See Appendix C for a list of *errno* messages.

3.6.2 On-line Help

On-line help provides you with a list of supported TNT commands. To obtain help, type:

```
E> TNT HELP (RET)
```

The system responds with the following help text:

```
Transparent Network Task Control V1.1
Transparent Network Task commands are:

TNT                     Display status of both TTT and TFA.
TNT HELP                Display this text.
TNT TTT OFF             Remove TTT from memory.
TNT TFA OFF             Remove TFA from memory.
```

If you mistype a command, TNT responds with an error message and the list of supported TNT commands:

```
Transparent Network Task Control V1.1
Transparent Network Task command error.
Transparent Network Task commands are:

TNT                     Display status of both TTT and TFA.
TNT HELP                Display this text.
TNT TTT OFF             Remove TTT from memory.
TNT TFA OFF             Remove TFA from memory.
```

3.6.3 Deinstalling TTT

You can remove TTT from memory. Enter the following command line:

```
E> TNT TTT OFF (RET)
```

The system responds with the following text:

```
Transparent Network Task Control V1.1
The task was removed successfully.
```

If TTT could not be removed, one of the following messages is displayed:

```
Transparent Network Task Control V1.1
TTT cannot be removed because it is not installed or is not installed
last.
```

or if MS–DOS failed on the remove call,

```
Transparent Network Task Control V1.1
The task could not be removed.
```

NOTE

TTT traps MS–DOS interrupt function call 21H as do other software applications. If you want to remove TTT from memory, it must be the last task installed which intercepts interrupt 21H. Otherwise, you must remove any tasks installed after TTT that also trap 21H, or reboot your system to remove TTT.

3.7 TTT Programming Considerations

There are specific MS–DOS function requests that support DECnet–VAXmate transparent task-to-task communication. Table 3–1 provides you with a summary of these calls. When creating TTT applications, you should note the following:

- Some user programs may not accept the TTT network specification string.
- You should not use unsupported MS–DOS function calls to perform transparent task-to-task communication.
- If you issue a CTRL/C while TTT is active, network operation may be blocked. To clear this condition, run the TNT utility.

3.8 MS–DOS Function Requests for Transparent Task-to-Task Communication

The following sections describe the MS–DOS function requests and provide specific guidelines. A drawing of the 8086/8088 registers shows their contents before and after each function request.

The function requests are discussed in alphabetical order.

3.8.1 Close

NAME

Close – terminate a logical link connection and deactivate the original handle.

On Entry	8086/8088 Register Contents
AH	3EH
BX	Handle for logical link access

On Return	8086/8088 Register Contents
AX	No errors (if carry bit is clear) Error code (if carry bit is set)

Figure 3–1: Close Function Request

DESCRIPTION

The *Close* function request terminates the logical link connection and deactivates the handle used for data exchange. Either task can issue the *Close* call.

On entry, the BX register contains the 16-bit handle value returned by the open or create I/O operation. If the close operation completes successfully and the carry bit is clear, no error is returned in the AX register. If an error condition occurs and the carry bit is set, the appropriate error code is returned in the AX register.

The following error code can occur:

Hexadecimal Value	Meaning
6	An invalid handle value was detected.

3.8.2 Create/Open

NAME

Create/Open – initiate or accept a logical link connection request.

On Entry	8086/8088 Register Contents
AH	3CH or 3DH
DS:DX	Address of network task specification string
AL	0

On Return	8086/8088 Register Contents
AX	Handle for logical link access (if carry bit is clear) Error codes (if carry bit is set)

Figure 3–2: Create/Open Function Request

DESCRIPTION

In the context of DECnet–VAXmate, the *Create* and *Open* calls perform the same functions. Either call can initiate and/or accept a logical link connection. However, if *SYS$NET* is specified as the node name in the network task specification, and supplied with either call, the function is **only** interpreted as the task accepting a logical link connection.

- **To initiate a logical link connection.** The *Create* or *Open* call enables a source task to initiate a logical link connection. On entry, DS:DX contains the address of the network task specification string. Any optional access control information is passed as part of the string to the target task.

  On return, the AX register contains an error code or a 16-bit handle associated with the source task. The returned handle value must be used for subsequent read and write I/O operations.

- **To accept a logical link connection.** The *Create* or *Open* call accepts a logical link request from another network task. The 16-bit handle value is returned in the AX register. This handle must be used for subsequent read and write operations. On entry, DS:DX contains the address of the network task specification string.

If you are unable to initiate a logical link connection, an error code is returned in the AX register. To obtain extended error information, run the TNT utility.

The following error code can occur:

Hexadecimal Value	Meaning
2	The network process may not be loaded. The node name to node address mapping is not found in the database file. The target task on the outgoing connection is not available. The network is unreachable. Too many files are currently open. There is an error in the path specification string. The target task was not found. There are too many active logical link connections. The remote object rejected the request.

If you are unable to accept a logical link connection, an error code is returned in the AX register. To obtain extended error information, run the TNT utility.

The following error code can occur:

Hexadecimal Value	Meaning
2	The target task was not found. Network access was denied.

3.8.3 Read

NAME

Read – receive data over a logical link connection.

On Entry	8086/8088 Register Contents
AH	3FH
DS:DX	Address of network message buffer
CX	Size of network message buffer
BX	Handle for logical link access

On Return	8086/8088 Register Contents
AX	Number of bytes received over the logical link (carry bit is clear)

Figure 3–3: Read Function Request

DESCRIPTION

The *Read* function request allows the target task to receive data sent over the logical link.

On entry, the BX register contains the 16-bit handle value. The CX register contains the number of bytes to be received. DS:DX contains the address of the network message buffer.

On return, the AX register contains the number of bytes successfully received by the target task. If an error condition occurs, zero bytes are returned. To obtain extended error information, run the TNT utility. See Appendix D for a list of extended error messages.

3.8.4 Write

NAME

Write – send data over a logical link connection.

On Entry	8086/8088 Register Contents
AH	40H
DS:DX	Address of network message buffer
CX	Size of network message buffer
BX	Handle for logical link access

On Return	8086/8088 Register Contents
AX	Number of bytes sent over the logical link (if carry bit is clear)
	Error codes (if carry bit is set)

Figure 3–4: Write Function Request

DESCRIPTION

The *Write* function request allows the source task to send data over the logical link.

On entry, the BX register contains the 16-bit handle value. The CX register contains the number of bytes to be sent. DS:DX contains the address of the network message buffer.

On return, the AX register contains the number of bytes successfully sent by the source task. If an error condition occurs, the error code is returned in the AX register. To obtain extended error information, run the TNT utility.

The following error code can occur:

Hexadecimal Value	Meaning
5	Network access was denied.

4
C Language

DECnet–VAXmate includes C language source files which are used to create a linkable library for DECnet–VAXmate applications. This library provides compatibility with the network socket interface supported by DECnet–ULTRIX.

4.1 Creating the DECnet–VAXmate Programming Interface Library

The file DNETLIB.SRC contains three types of files: the .C files (C language sources), the .H files (header files that contain definitions for the network interface) and the .ASM files (assembly language sources). You should refer to the appropriate installation guide for a complete list of these files.

In order to interface to the DECnet–VAXmate network process, you should create a library against which to link your DECnet–VAXmate program(s).

Use the following procedure to create a DECnet–VAXmate programming interface library:

1. The Break Source utility, BREAKSRC, allows you to break the source file, DNETLIB.SRC, into separate source files for compilation and assemblies. BREAKSRC is supplied with the DECnet–VAXmate distribution kit. When you run the DECnet–VAXmate Installation Procedure (DIP), you can select to have the DNETLIB.SRC file split into separate files. The BREAKSRC utility will then be run automatically for you. For instructions on how to run DIP, refer to the appropriate installation guide.

To use BREAKSRC manually, follow this format:

BREAKSRC <input_file_spec> <output_device:\path>

For example:

BREAKSRC A:DNETLIB.SRC C:\DECNET\SRC\

2. Use your C language compiler to compile each C language source module. Use your assembler to assemble each assembly source module.

3. After you produce an object module for each source module, build a library against which to link your DECnet–VAXmate applications programs.

4.1.1 DECnet–VAXmate Programming Considerations

The following programming considerations should be noted when writing and developing your DECnet–VAXmate applications:

1. Using the External Variable *errno* — Most DECnet–VAXmate programming interface functions use the external variable *errno* as a place to return error detail. It is assumed that *errno* has been defined externally to the programming interface as an *int*. It may already be defined in your C language run-time library; if not, your applications program should define it.

   Appendix C lists the error codes returned by DECnet–VAXmate in *errno*.

2. Checking Software Compatibility — When creating DECnet–VAXmate applications, make sure that you resolve any C language compiler incompatibilities before compiling the C language source modules such as long variable names or certain type definitions.

3. If your C compiler does not do so by default, you should compile the sources so that all data structures are stored without extra space (packed) for alignment of members on "int" boundaries.

4. Using Assembly Source Modules — There are assembly source modules included in DNETLIB.SRC. Before you can successfully call these functions from sources compiled by your C language compiler, you should fulfill any assembly-format requirements such as segment names. (Refer to the header file, *begin.h*, on the distribution kit as an example of specific C language compiler segment naming requirements.)

5. Using Specific Macros — There are references to the macros/functions such as *toupper* and *islower* in some of the C language source modules. It is assumed that your C language compiler has provided a standard macro/function for them. If not, you can simply provide your own macros/functions.

6. If you are not using a C compiler that supports the *signal* handling function (for example, CTRL/C trapping), then you will need either to provide your own *signal* function or to comment out from the code in *dnet_conn* any references to the *signal()* function and the reference to the include file, <signal.h>.

The CTRL/C trapping, provided in *dnet_conn*, allows you to abort a utility which appears to be waiting indefinitely for connections to complete. (This applies only to utilities that use *dnet_conn* for making connections.) The *connect* function in *dnet_conn* is invoked in a nonblocking socket mode.

If CTRL/C trapping code is commented out from *dnet_conn*, there is the potential problem of leaving hung sockets and running out of system resources, should users decide to CTRL/C in the middle of nonblocking connection request attempts.

7. If you are using a compiler that does not define the following variables: *int daylight*, *long timezone* and *char *tzname[2]*, either comment these external declarations out of the header file <time.h> or define them in your C program code.

8. The DECnet–VAXmate programming interface library was based on a 2-segment model: one code segment and one data segment.

CAUTION

If a program terminates with any active logical links (sockets), the links remain active. In this way, another program can start and use the same links.

If the logical links are not needed, you must issue an *sclose* function call before a program is terminated with an EXIT, CTRL/Z or CTRL/C. If too many links are left active, an error message, indicating that no more buffers are available or that there are insufficient network resources, is reported. This causes the network to become unusable. To completely deactivate any logical links, run the Network Control Program (NCP) and issue the SET KNOWN LINKS command with STATE OFF. (See the *DECnet–VAXmate User's Guide* for more information on this command.)

4.2 How to Read the Socket Interface Call Descriptions

The socket interface calls are presented as separate entries in this manual. They are documented in a consistent manner. Each call is described under the following headings:

NAME	gives the exact name and a brief description of its function.
SYNTAX	shows the complete syntax. Possible call options and the type of expected argument(s) are indicated.
DESCRIPTION	provides more detail on what the call does, and how its action is modified by the options.
DIAGNOSTICS	gives explanations of error messages that may be produced.

4.3 Understanding a SYNTAX Section

Each socket interface call documented in this chapter has a SYNTAX entry. It shows how a call is defined. The SYNTAX entry consists of several components.

The SYNTAX section for the *bind* call illustrates these components:

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

The first line represents the function call and a list of input arguments. Each function call should do one specific task. Data may be passed to the called function by way of arguments. The input arguments follow the function name. They are separated by commas and surrounded by parentheses.

The next set of lines lists the formal arguments and their respective data types (char, int, or structure).

Using the *bind* call example, *name declares name to be a pointer to the structure type *sockaddr_dn*. This structure contains several modifiable fields.

4.4 Socket Function Calls

The following sections describe the socket interface function calls for C programs. They also provide you with specific guidelines. Some of the socket interface calls use specific data structures. Appendix B details how each data structure is formatted.

The socket interface calls are summarized in the table below:

Table 4–1: Socket Interface Calls

Socket Call	Description
accept	Accept an incoming connection request on a socket, and return a socket number.
bind	Assign an object name or number to a socket.
connect	Initiate a connection request on a socket.
getpeername	Get the name of a connected peer on a socket.
getsockname	Get the current name for the specified socket.
getsockopt	Get options associated with sockets.
listen	Listen for pending connections on a socket.
recv	Receive data and out-of-band messages on a socket.
sclose	Terminate a logical link connection and deactivate a socket.
select	Check the I/O status of the network sockets.
send	Send data and out-of-band messages on a socket.
setsockopt	Set options associated with sockets.
shutdown	Shutdown part or all of a full duplex logical link connection.
sioctl	Control the operations of sockets.
socket	Create an endpoint for communication and return a socket number.
sread	Read data on a socket.
swrite	Write data to a socket.

4.4.1 Example Socket Interface Calling Sequence

The following program segments illustrate the socket interface calls used by DECnet–VAXmate client and server tasks.

Socket calls issued by a client task:

```
s = socket(...)                  /* get a DECnet socket          */
setsockopt(s,,DSO_CONDATA,,)     /* set up optional data         */
setsockopt(s,,DSO_CONACCESS,,)   /* set up access                */
                                 /* control information          */
connect(s,...)                   /* initiate connection to       */
                                 /* the server task              */
getsockopt(s,,DSO_CONDATA,,)     /* get returned status and      */
                                 /* optional data                */
send(s,...)                      /* send data (or write)         */
recv(s,...)                      /* receive data (or read)       */
setsockopt(s,,DSO_DISDATA)       /* set up optional              */
                                 /* data for disconnect          */
sclose(s)                        /* terminate connection         */
```

Socket calls issued by a server task:

```
s = socket(...)                  /* get a DECnet socket          */
bind(s,...)                      /* bind a name to the socket    */
listen(s,...)                    /* make the socket available    */
                                 /* for client connections       */
setsockopt(s,,DSO_ACCMODE,       /* accept mode,                 */
        ACC_DEFER,)              /* deferred/immediate           */
ns = accept(s,...)               /* await connect(s) on the      */
                                 /* new socket, ns               */
getsockopt(ns,,DSO_CONACCESS,,)/* get access                    */
                                 /* control information          */
getsockopt(ns,,DSO_CONDATA,,)    /* get optional connect data    */
setsockopt(ns,,DSO_CONDATA,,)    /* set up optional data         */
setsockopt(ns,,DSO_CONACCEPT,,)/* finally accept the             */
                                 /* connection request           */
recv(ns,...)                     /* receive data (or read)       */
send(ns,...)                     /* send data (or write)         */
setsockopt(ns,,DSO_DISDATA,,)    /* set up optional              */
                                 /* disconnect data              */
sclose(ns)                       /* terminate connection         */
sclose(s)                        /* terminate DECnet path        */
```

4.4.2 accept

NAME

accept — accept an incoming connection request on a socket and return a socket number.

SYNTAX

```
#include <types.h>
#include <socket.h>
#include <dn.h>

int         accept(s, sorcblk, sorclen)
int         s, *sorclen;
struct      sockaddr_dn *sorcblk;
```

DESCRIPTION

The *accept* call extracts the first connection request on the queue of pending connections, creates a new socket with a new number having the same properties of the original listening socket. The original socket remains opened.

If the socket is set to nonblocking I/O, and there are no queued connection requests, *io_status* will return a –1 and *errno* will contain EWOULBLOCK.

There are two modes of accepting an incoming connection. They are immediate and deferred modes. These modes of acceptance are set by using the *setsockopt* call. When immediate mode is in effect, the connection is established immediately. The deferred mode indicates that the server task completes the *accept* call without fully completing the connection to the client task. In this case, the server task can examine the access control or optional data before it decides to accept or reject the connection request. The server task can then issue the *setsockopt* call with the appropriate reject or accept option.

Input Arguments

s — specifies the number for a socket which was created with the *socket* call, bound to a name or number by the *bind* call, and was set to listen for connects by the *listen* call.

sorcblk — is a value result argument. It specifies an address of a structure *sorcblk* of the data type *sockaddr_dn*. This argument will be filled in with the information of the entity requesting the connection.

sorclen — is a value result argument. It specifies the address of an int. The value of *sorclen* should initially contain the size of the *sorcblk*.

Return Arguments

sorclen specifies the actual length of the returned data in bytes.

sorcblk specifies the socket address data structure, *sockaddr_dn*. A user retrieves data from the fields filled in by this function call. (See Appendix B on how *sockaddr_dn* is formatted.)

The following data fields are filled in by this function call:

sdn_family is the address family AF_DECnet.

sdn_objnum is the object number for the client task. It can be a number 0 to 255. It is set to 0 only when the object name is used.

sdn_objnamel is the size of the object name.

sdn_objname is the object name of the client task. It can be up to a 16-element array of char. It is used only when *sdn_objnum* equals 0.

sdn_add is the node address structure for the client task. (See Appendix B on how *dn_naddr* is formatted.)

Return Value

If the call succeeds, it returns a nonnegative integer called a socket number. This number will be used for communications over a logical link connection. If an error occurs, the call returns a –1. When an error condition exists, the external variable *errno* will contain error detail. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

[EBADF]	The argument *s* does not contain a valid socket number.
[ECONNABORTED]	The client task disconnected before the *accept* call completed.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENFILE]	There are no more available sockets.
[EWOULDBLOCK]	The socket is marked for nonblocking and no connections are waiting to be accepted.

4.4.3 bind

NAME

bind — assign an object name or number to a socket.

SYNTAX

```
#include <types.h>
#include <socket.h>
#include <dn.h>

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

DESCRIPTION

The *bind* call assigns an object name or number to a socket. When a socket is first created with the *socket* call, it exists in a name space but has no assigned name or number. The *bind* call is used primarily by server tasks. The object name is required before a server task can listen for incoming connection requests using the *listen* call. It can also be used by client tasks to identify themselves to server tasks. See also the *accept* (Section 4.4.2), *connect* (see Section 4.4.4), *getpeername* (Section 4.4.5), and *getsockname* (Section 4.4.6) calls.

NOTE

VAX/VMS proxy access by user name is made possible if the client task uses the *bind* call specifying his user name as the object name. Refer to the SO_REUSEADDR option for the *setsockopt* call (Section 4.4.12) if you want to make more than one proxy connection with the same name.

Input Arguments

s specifies the number for a socket which has been created with the *socket* call.

name specifies the address of the structure *name* of data type *sockaddr_dn*. A user fills in the data for each field. (See Appendix B on how *sockaddr_dn* is formatted.)

The following data fields can be modified:

sdn_family specifies the address family as AF_DECnet.

sdn_flags specifies the object flag option. It must be set to 0.

sdn_objnum defines the object number for the server task. It can be a number 0 to 255. It is set to 0 only when the object name is used.

sdn_objnamel is the size of the object name.

sdn_objname defines the object name of the server or client task. It can be up to a 16-element array of char. It is used only when *sdn_objnum* equals 0.

sdn_add specifies the node address structure for the server task. This data member is ignored.

namelen specifies the size of the name structure.

Return Value

If the bind is successful, a 0 value is returned. An unsuccessful bind returns a value of –1. When an error condition exists, the external variable *errno* will contain error detail.

DIAGNOSTICS

[EADDRINUSE] The specified name or number is already used by another socket.

[EBADF] The argument *s* does not contain a valid socket number.

[EINVAL] The socket *s* is already bound to a name or number.

[ENETUNREACH] The network is unreachable. The network process is not installed.

4.4.4 connect

NAME

connect — initiate a connection request on a socket.

SYNTAX

```
#include <types.h>
#include <socket.h>
#include <dn.h>

int         connect(s, destblk, destlen)
int         s, destlen;
struct      sockaddr_dn, *destblk;
```

DESCRIPTION

The *connect* call issues a connection request to another socket. The other socket is specified by *destblk* which is a pointer to the *destblk* data structure.

Optional data as well as access control information may be passed with this function call. This data must be previously set by the *setsockopt* call. If subsequent *connect* calls are issued on the same socket, a task must reissue the *setsockopt* call to set up new optional user data and/or access control information.

Input Arguments

s specifies the number for the socket which has been created with the *socket* call. This socket number is used for establishing a connection between the user tasks. It is also used with subsequent send and receive function calls.

destblk specifies the address of the structure *destblk* of the data type *sockaddr_dn*. A user fills in the data for each field. (See Appendix B on how *sockaddr_dn* is formatted.)

The following data fields can be modified:

sdn_family specifies the address family as AF_DECnet.

sdn_flags specifies the object flag option. It must be set to 0.

sdn_objnum defines the object number for the server task. It can be a number 0 to 255.

sdn_objnamel is the size of the object name.

sdn_objname defines the object name of the server task. It can be up to a 16-element array of char. It is used only when *sdn_objnum* equals 0.

sdn_add specifies the node address structure for the server task. (See Appendix B on how *dn_naddr* is formatted.)

destlen specifies the size of the destination block structure.

Return Value

If the call succeeds, it returns a value of 0. Otherwise, the call returns a value of −1. When an error condition exists, the external variable *errno* will contain error detail. If the socket is set to nonblocking I/O (see also *sioctl*, Section 4.4.14), and you issue a *connect*, the function returns a −1, and the error message, EINPROGRESS.

DIAGNOSTICS

[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this particular socket.
[EBADF]	The argument *s* does not contain a valid socket number.
[EBUSY]	The socket is not in idle state. The socket is in the process of being connected or disconnected; it is currently a connected or listening socket.
[ECONNABORTED]	The peer task has disconnected and the connection was aborted.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ECONNRESET]	The remote task has failed.
[EHOSTUNREACH]	The remote node is unreachable.
[EINPROGRESS]	The connection request is now in progress.
[ENETDOWN]	The network is down. The Executor name and address may not have been set and/or the Executor state may not have been set ON.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ERANGE]	The object number of the server task is invalid. The valid range is 0 to 255.
[ESRCH]	The server object does not exist on the remote node.
[ETIMEDOUT]	Connection establishment was timed out before a connection was established.
[ETOOMANYREFS]	The remote node has accepted the maximum number of connection requests.

4.4.5 getpeername

NAME

getpeername — get the name of a connected peer on a socket.

SYNTAX

```
#include <types.h>
#include <socket.h>
#include <dn.h>

int         getpeername(s, destblk,destlen)
int         s, *destlen;
struct      sockaddr_dn *destblk;
```

DESCRIPTION

The *getpeername* call returns information about the peer socket connected to the specified socket. This information is the same information returned by the *accept* call. It may be used by a client or server task anytime after a connection has been established between two tasks or peers.

Input Arguments

s specifies the number for a socket which has been created by the *socket* or the *accept* call.

destblk specifies the address of the *destblk* structure of the data type *sockaddr_dn*. This argument will be filled in with peer information returned by *getpeername*.

destlen is a value result argument. It specifies the address of an int. The value of *destlen* should be initialized to the size of the *destblk*.

Return Arguments

destlen specifies the actual size of the destination block (in bytes).

destblk specifies the socket address data structure, *sockaddr_dn*. A user retrieves data from the fields filled in by this function call. (See Appendix B on how *sockaddr_dn* is formatted.)

The following data fields can be filled in by this function call:

sdn_family is the address family AF_DECnet.

sdn_objnum is the object number for the peer task. It can be a number 0 to 255.

sdn_objnamel is the size of the object name.

sdn_objname is the object name of the peer task. It can be up to a 16-element array of char. It is only used when *sdn_objnum* equals 0.

sdn_add is the address structure for the peer node. (See Appendix B on how *dn_naddr* is formatted.)

Return Value

If the call succeeds, a value of 0 is returned. If an error occurs, the call returns a −1. When an error condition exists, the external variable *errno* will contain error detail. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

[EBADF]	The argument *s* does not contain a valid socket number.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOTCONN]	The socket *s* is not connected, it has no peer.

4.4.6 getsockname

NAME

getsockname — get the current object name or number for the specified socket.

SYNTAX

```
#include <types.h>
#include <socket.h>
#include <dn.h>

int       getsockname(s, destblk, destlen)
int       s, *destlen;
struct    sockaddr_dn *destblk;
```

DESCRIPTION

The *getsockname* call returns the bound object name or number of the specified socket.

Input Arguments

s specifies the number for a socket which has been created by the *socket* or the *accept* call.

destblk specifies the address of a structure of the data type *sockaddr_dn*. This argument will be filled in with local task information returned by *getsockname*.

destlen is a value result argument. It specifies the address of an int. The value of *destlen* should be initialized to the size of the *destblk*.

Return Arguments

destlen specifies the actual size of the destination block (in bytes).

destblk specifies the socket address data structure, *sockaddr_dn*. A user retrieves data from the fields filled in by this function call. (See Appendix B on how *sockaddr_dn* is formatted.)

The following data fields can be filled in by this function call:

sdn_family is the address family AF_DECnet.

sdn_objnum is the object number for the local task. It can be a number 0 to 255.

sdn_objnamel is the size of the object name.

sdn_objname is the object name of the local task. It can be up to a 16-element array of char. It is only used when *sdn_objnum* equals 0.

sdn_add is the address structure for the local node. (See Appendix B on how *dn_naddr* is formatted.)

Return Value

If the call succeeds, a value of 0 is returned. If an error occurs, the call returns a −1. When an error condition exists, the external variable *errno* will contain error detail. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

[EBADF]	The argument *s* does not contain a valid socket number.
[ENETUNREACH]	The network is unreachable. The network process is not installed.

4.4.7 listen

NAME

listen — listen for pending connections on a socket.

SYNTAX

```
int     listen(s, backlog)
int     s;
int     backlog;
```

DESCRIPTION

The *listen* call declares your socket as a server which is available for client connections. The server uses the bound name or number in order to listen for incoming client connections. This call must be issued before an incoming connection can be accepted or rejected. See also the *accept* (Section 4.4.2), the *bind* (Section 4.4.3) and the *select* (Section 4.4.10) calls.

If you detach a listening socket while the socket is receiving client connections, then all links associated with the listening socket immediately abort and all outstanding data is lost.

Input Arguments

s specifies the number for a socket which has been created with the *socket* call and bound to a name or number by the *bind* call.

backlog defines the maximum number of unaccepted incoming connects which are allowed on this particular socket. The maximum allowable number is 5. If a connection request arrives when the queue is full, the client task will receive an error with an indication of ECONNREFUSED.

Return Value

If the call succeeds, a value of 0 is returned. If an error occurs, the call returns a −1. When an error condition exists, the external variable *errno* will contain error detail. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

[EBADF]	The argument *s* does not contain a valid socket number.
[ECONNREFUSED]	The connection request was rejected.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[EOPNOTSUPP]	The socket type does not support the listen operation.

NOTE

You may issue a *listen(s, 0)* call while already processing data over a previously accepted connected socket. If this is done, subsequent incoming connection requests will be rejected by the network process. When communications are completed over the currently connected socket, the *listen(s, backlog)* call should be reissued to allow for subsequent acceptance of incoming connection requests.

4.4.8 recv

NAME

recv — receive data or out-of-band messages on a socket.

SYNTAX

```
#include <socket.h>

int     recv(s, buffer, buflen, flags)
int     s, buflen, flags;
char    *buffer;
```

DESCRIPTION

The *recv* call is used to receive data from your peer. See also the *sread* call (Section 4.4.16).

If no messages are available at the socket, the *recv* call waits for a message to arrive unless the socket is nonblocking. (See *sioctl*, Section 4.4.14.) In this case, a status of –1 is returned with the external variable *errno* set to EWOULDBLOCK.

If the link is disconnected, queued data can still be received on the socket. However, if you shut down the socket or detach it, queued data cannot be received. When the logical link is not in a connected state, and all data has been read, the *recv* call returns zero bytes.

The *select* call may be used to determine when more data has arrived. (See Section 4.4.10.)

Out-of-band messages are delivered to a receiving task ahead of normal data messages. These messages can be received by specifying *MSG_OOB* as the flag argument.

Input Arguments

s specifies the number for a socket returned by the *socket* or the *accept* call.

buffer specifies the address of a buffer which will contain the received message.

buflen specifies the size of the message buffer.

flags set to 0 indicates that the task will receive normal messages. If set to *MSG_OOB*, the task will receive out-of-band messages. Only one out-of-band message can be outstanding at any time. You can also set the *flags* argument to *MSG_PEEK* to read the next pending message without removing it from the receive queue.

Output Argument

buffer specifies the buffer which contains the received message.

Return Value

If the call succeeds, the number of received characters is returned.

If the call returns a zero, you have either received a zero length message or the logical link has been disconnected. To determine the state of the logical link, use the *getsockopt* function call with the *DSO_LINKINFO* option (see Section 4.4.12), or the DECnet utility function, *dnet_eof* (see Chapter 5). If the link has been disconnected, then all subsequent receives will return zero bytes.

If an error occurs, a value of −1 is returned. Additional error detail will be specified in the external variable *errno*. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

When receiving **normal data**, the following set of error messages can occur:

Blocking I/O

Message	Description
[EBADF]	The argument *s* does not contain a valid socket number.
[ENETUNREACH]	The network is unreachable. The network process is not installed.

Nonblocking I/O

Message	Description
[EBADF]	The argument *s* does not contain a valid socket number.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[EWOULDBLOCK]	The receive operation would block because there is currently no data to receive.

When receiving **out-of-band data**, the following set of error messages can occur:

Blocking I/O

Message	Description
[EBADF]	The argument *s* does not contain a valid socket number.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[EWOULDBLOCK]	The receive operation would block because there is currently no data to receive.

Message	Description
[EBADF]	The argument *s* does not contain a valid socket number.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[EWOULDBLOCK]	The receive operation would block because there is currently no data to receive.

4.4.9 sclose

NAME

sclose — terminate a logical link connection and deactivate a socket.

SYNTAX

```
int     sclose(s)
int     s;
```

DESCRIPTION

The *sclose* call terminates an outstanding connection over the socket referenced by *s*. It also deactivates the socket.

NOTE

Before you can terminate a connection over a socket with the option SO_KEEPALIVE set, you must first issue a *setsockopt* call with SO_KEEPALIVE turned off. To turn off SO_KEEPALIVE, you must precede SO_KEEPALIVE with a tilde (~), as in, ~SO_KEEPALIVE. (~SO_KEEPALIVE is the default condition.)

You then issue the *sclose* call. The logical links (if any) are disconnected, and the socket and associated sockets (if any) are deallocated. However, if you issue *sclose* without turning off SO_KEEPALIVE, the sockets remain allocated, and the links (if any) stay active.

The effect of *sclose* on unsent data queued for a remote task depends on the linger option set with the *setsockopt* function call. (See Section 4.4.12.) If SO_LINGER is set, control is returned to the task, but the link is not disconnected until the unqueued data is sent. If SO_DONTLINGER is set, control is returned to the task, and any unqueued data is lost.

NOTE

The DECnet–VAXmate function call *sclose* is not compatible with DECnet–ULTRIX systems. See Appendix G for information on how to transport DECnet–VAXmate programs that use *sclose*.

Input Argument

s specifies the number for a socket which was returned by the *socket* or the *accept* call.

Return Value

If the call succeeds, a value of 0 is returned. If an error occurs, a value of –1 is returned. Additional error detail will be specified in the external variable *errno*. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

[EBADF]	The argument *s* does not contain a valid socket number.
[ENETUNREACH]	The network is unreachable. The network process is not installed.

4.4.10 select

NAME

select — check the I/O status of the network sockets.

SYNTAX

#include <time.h>

```
int             select(nfds, readfds, writefds, exceptfds, timeout)
int             nfds;
unsigned long   *readfds, *writefds, *exceptfds;
struct          timeval *timeout;
```

DESCRIPTION

The *select* call checks the network sockets specified by the bit masks ***readfds***, ***writefds***, and *exceptfds*, respectively, to see if they are ready for reading, writing, or have any outstanding out-of-band messages. The *select* call does not tell you if the logical link connection has been broken.

You should use the *select* call to help manage your *accept*, *send*, *recv*, *swrite*, and ***sread*** calls.

The *readfds*, *writefds*, and *exceptfds* I/O descriptors are long words which contain bit masks. Each bit in a mask represents one socket number. For example, socket "3" is the fourth bit or has a hex value of 8.

NOTE

The *select* call can only check socket numbers in the range 0 to 31.

To specify the bit for any socket number, use the value returned by the ***socket*** or the *accept* call, as "1<<*s*".

Input Arguments

nfds specifies the highest socket number to be checked. The bits from (1<<0) to (1<<(nfds-1)) are examined.

readfds specifies the socket numbers to be examined for read ready. For listening sockets, a read ready condition indicates that an incoming connection request can be read and either accepted or rejected. For sequenced sockets, there is a complete message to be read. For stream sockets, there is some data to be read. If a socket disconnects or aborts, a read ready condition will always occur.

NOTE

To prevent a program from hanging on a stream socket, issue the *sioctl* call with the FIONREAD function argument (see Section 4.4.14), and then read

only those numbers of bytes returned by the call. You should also perform socket operations in nonblocking I/O mode.

This descriptor can be given as a null pointer if of no interest.

writefds specifies the socket numbers to be examined for write ready. A write ready condition exists when the logical link is available. This descriptor can be passed as a null pointer if of no interest.

exceptfds specifies the socket numbers to be examined for out-of-band data ready. There is a pending out-of-band message to receive. This descriptor can be given as a null pointer if of no interest.

NOTE

The bit mask *exceptfds* is presently not supported by DECnet–ULTRIX.

timeout specifies a pointer to a data structure of type *timeval*. If this pointer is null, then the *select* call will wait until an event occurs. If the pointer is non-null, and the time value is greater than zero, then the *select* call will return either after *n* seconds have expired or when an event occurs, whichever one comes first. If the pointer is non-null and the time value is zero, then the *select* call will return after an immediate poll.

timeval specifies the amount of time to wait. The data members are:

tv_sec specifies the time in seconds.

tv_usec this data member is ignored.

Output Arguments

read_fds If a socket is read ready, the bit is returned "on", and *read_fds* returns the socket numbers (as bit masks) to be examined. If the socket is not read ready, the bit is cleared.

write_fds If a socket is write ready, the bit is returned "on", and *write_fds* returns the socket numbers (as bit masks) to be examined. If the socket is not write ready, the bit is cleared.

except_fds If the socket is out-of-band data ready, the bit is returned "on", and *except_fds* returns the socket numbers (as bit masks) to be examined. If the socket is not out-of-band data ready, the bit is cleared.

Return Value

The value returned by the *select* call is the number of bits set in all the masks. The bit masks contain the set bits that correspond to the sockets in which events have occurred. If the time period expires, a value of 0 is returned.

If an error occurs, a value of –1 is returned. Additional error detail will be contained in the external variable *errno*. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

[EBADF]	One of the specified bit masks is an invalid descriptor.
[ENETUNREACH]	The network is unreachable. The network process is not installed.

4.4.11 send

NAME

send — send data or out-of-band messages on a socket.

SYNTAX

```
#include <socket.h>

int         send(s, buffer, buflen, flags)
int         s, buflen, flags;
char        *buffer;
```

DESCRIPTION

The *send* call is used to transmit data to your peer. The client task uses the socket number returned by the *socket* call. The server task uses the socket number returned by the *accept* call.

If you cannot get enough buffer space while building the outgoing message on a blocking socket, the message is blocked. You must wait until current transmissions are finished. For a nonblocking socket, the error message, EWOULDBLOCK, is returned. If a socket disconnects, any outstanding data to be sent is discarded.

The flag option, *MSG_OOB*, can be set to indicate that out-of-band data will be sent to your peer socket. An out-of-band message is a high priority message that you can send to your peer. This message bypasses any normal messages waiting to be received. An out-of-band message must be received by your peer before another message can be sent.

The *select* call can be used to determine if it is possible to send more data. (See Section 4.4.10.)

Input Arguments

s	specifies the number for a socket returned by the *socket* or the *accept* call.
buffer	specifies the address of the buffer which contains the outgoing message.
buflen	specifies the size of the outgoing message.
flags	can be set to 0 to indicate normal messages. It can be set to *MSG_OOB* for out-of-band messages.

Return Value

If the call succeeds, the number of characters sent is returned. If an error occurs, a value of −1 is returned. Additional error detail will be specified in the external variable *errno*. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

When sending **normal data**, the following set of error messages can occur:

Blocking I/O

Message	Description
[EBADF]	The argument *s* does not contain a valid socket number.
[EMSGSIZE]	The size of the outgoing message is more than 2048 bytes.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOTCONN]	The *send* call did not complete and the link was disconnected.
[EPIPE]	The link has been disconnected, aborted, or shut down. No further messages can be sent.

Nonblocking I/O

Message	Description
[EBADF]	The argument *s* does not contain a valid socket number.
[EMSGSIZE]	The size of the outgoing message is more than 2048 bytes.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOTCONN]	The *send* call did not complete and the link was disconnected.
[EPIPE]	The link has been disconnected, aborted, or shut down. No further messages can be sent.
[EWOULDBLOCK]	The outbound quota was full, and the message could not be sent. Try again later.

When sending **out-of-band data**, the following set of error messages can occur:

Blocking I/O

Message	Description
[EALREADY]	The out-of-band message could not be sent. A similar transmission request is still in progress.
[EBADF]	The argument *s* does not contain a valid socket number.
[EMSGSIZE]	The size of the outgoing message is more than 16 bytes.

[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOTCONN]	The *send* call did not complete and the link was disconnected.
[EPIPE]	The link has been disconnected, aborted, or shut down. No further messages can be sent.

Nonblocking I/O

Message	Description
[EALREADY]	The out-of-band message could not be sent. A similar transmission request is still in progress.
[EBADF]	The argument *s* does not contain a valid socket number.
[EMSGSIZE]	The size of the outgoing message is more than 16 bytes.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOTCONN]	The *send* call did not complete and the link was disconnected.
[EPIPE]	The link has been disconnected, aborted, or shut down. No further messages can be sent.

4.4.12 setsockopt and getsockopt

NAME

setsockopt and getsockopt — set and get the options associated with sockets.

SYNTAX

```
#include <types.h>
#include <socket.h>
#include <dn.h>

int     setsockopt(s, level, optname, optval, optlen)
int     s, level, optname, optlen;
char    *optval;

int     getsockopt(s,level, optname, optval, optlen)
int     s, level optname, *optlen;
char    *optval;
```

DESCRIPTION

The *setsockopt* and *getsockopt* calls manipulate various options associated with a socket. Options exist at multiple levels and you must specify the level number for the desired operation.

At the socket level (SOL_SOCKET), the options include:

- **SO_KEEPALIVE.** If this option is set on a socket, any links and sockets associated with this socket will remain active, despite any attempts to disconnect them.

  NOTE

  Before you can terminate a connection over a socket with the option SO_KEEPALIVE set, you must first issue a *setsockopt* call with SO_KEEPALIVE turned off. To turn off SO_KEEPALIVE, you must precede SO_KEEPALIVE with a tilde (~), as in, ~SO_KEEPALIVE. (~SO_KEEPALIVE is the default condition.)

  You then issue the *sclose* call. The logical links (if any) are disconnected, and the socket and associated sockets (if any) are deallocated. However, if you issue *sclose* without turning off SO_KEEPALIVE, the sockets remain allocated, and the links (if any) stay active.

- **SO_LINGER.** SO_LINGER controls the actions taken when unsent messages are queued on a socket and the *sclose* call is issued. If SO_LINGER is set, the connection is maintained until the outstanding messages have been sent. This is the default condition.

- **SO_DONTLINGER.** SO_DONTLINGER also controls the actions of unsent messages. If SO_DONTLINGER is set, and the *sclose* call is issued, any outstanding messages queued to be sent will be lost. The connection is then terminated.
- **SO_REUSEADDR.** SO_REUSEADDR allows the reuse of a name already bound to a socket. For most situations, a name is bound to a socket only once. However, this option enables you to reuse the same name. This particular option **must** only be used for outgoing connection requests. It cannot be used for incoming connections.

At the DECnet level (DNPROTO_NSP), socket options may specify the way in which a connection request is accepted or rejected, may be used to set up optional user data and/or access control information, or may be used to obtain current link state information. The following socket options can be specified:

- **DSO_ACCEPTMODE.** The accept option mode is used at the DECnet level for processing *accept* calls. A socket must be bound (see Section 4.4.3) before specifying this option. There are two values which can be supplied for this option. They are immediate mode, ACC_IMMED, and deferred mode, ACC_DEFER.
  - **ACC_IMMED.** ACC_IMMED mode is the default condition for this option. When immediate mode is in effect, control is immediately returned to the server task following an *accept* call with the connection request accepted. The access control information and/or optional data is ignored by the server task.
  - **ACC_DEFER.** ACC_DEFER mode enables the server task to complete the *accept* call without fully completing the connection to the client task. In this case, the server task can examine the access control or optional data before it decides to accept or reject the connection request. The server task can then issue the *setsockopt* call with the appropriate reject or accept option.
- **DSO_CONACCEPT.** DSO_CONACCEPT allows the server task to accept the pending connection on the socket returned by the *accept* call. The original listening socket was set to deferred accept mode. Any optional data previously set by DSO_CONDATA will also be sent.
- **DSO_CONREJECT.** DSO_CONREJECT allows the server task to reject the pending connection on the socket returned by the *accept* call. The original listening socket was set to deferred accept mode. Any optional data previously set by DSO_DISDATA will also be sent. The reject reason is the value passed with this option.
- **DSO_CONDATA.** DSO_CONDATA allows up to 16 bytes of optional user data to be set by the *setsockopt* call. It can be sent as a result of the *connect* or the *accept* (with the deferred option) calls. The optional data is passed in a structure of type *optdata_dn*. (See Appendix B on how *optdata_dn* is formatted.) The data is read by the task issuing the *getsockopt* call with this option.

- **DSO_DISDATA.** DSO_DISDATA allows up to 16 bytes of optional data to be set by the *setsockopt* call. It can be sent as a result of the *sclose* call. The optional data is passed in a structure of type *optdata_dn*. (See Appendix B on how *optdata_dn* is formatted.) The data is read by the task issuing the *getsockopt* call with this option.
- **DSO_CONACCESS.** DSO_CONACCESS allows access control information to be passed by the user task. This information is set with the *setsockopt* call. The access data is sent to the server task. It is passed with the *connect* call in a structure of type *accessdata_dn*. (See Appendix B on how *accessdata_dn* is formatted.) The access data is read by the task issuing the *getsockopt* call with this option.
- **DSO_LINKINFO.** DSO_LINKINFO determines the state of the logical link connection.

  When the *getsockopt* call is issued with this option, the state of the logical link is returned in a logical link information data structure, *linkinfo_dn*. (See Appendix B on how *linkinfo_dn* is formatted.)

Input Arguments

s — specifies the number for a socket returned by the *socket* or the *accept* call.

level — specifies the level at which options are manipulated. The level is either SOL_SOCKET or DNPROTO_NSP. (See Appendix A for details.)

optname — specifies options to be interpreted at the level specified. For example, SO_LINGER at the SOL_SOCKET level.

optval, optlen — specify access option values used with the *setsockopt* and the *getsockopt* calls. The interpretation of each argument is function dependent as shown here:

setsockopt call

optval — specifies the address for a buffer which contains information for setting option values.

optlen — specifies the size of the option value buffer.

getsockopt call

optval — specifies the address of a buffer which will contain the returned value for the requested option(s).

optlen — is a value result parameter. It specifies the address of an int. The value of *optlen* should initially contain the size of the buffer pointed to by *optval*. On return, it will contain the actual size of the returned value.

Output Arguments (for *getsockopt* only)

optval specifies the buffer which contains the returned value for the requested socket option(s).

optlen specifies the actual size of the returned value.

Return Values

If the call completes successfully, a value of 0 is returned. An unsuccessful call returns a value of −1. When an error condition exists, the external variable *errno* will contain error details. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

[EACCES]	Unable to disconnect the socket.
[EBADF]	The argument *s* does not contain a valid socket number.
[ECONNABORTED]	The accept connect did not complete. The peer task disconnected and the connection was aborted.
[EDOM]	The acceptance mode is not valid.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOBUFS]	There are no available buffers for optional access control and/or user data.
[ENOPROTOOPT]	No access control information was supplied with the connection request.
[EOPNOTSUPP]	The option is unknown.

4.4.13 shutdown

NAME

shutdown — shutdown all or part of a full duplex logical link.

SYNTAX

```
int     shutdown(s, how)
int     s, how;
```

DESCRIPTION

The *shutdown* call causes all or part of a full duplex connection on the original socket to be shut down.

Input Arguments

s specifies the number for a socket returned by the *socket* or the *accept* call.

how specifies the type of shutdown. The *how* argument can be set to:

0 which disallows further receives or reads.

1 which disallows further sends or writes.

2 which disallows further sends (or writes) and receives (or reads).

Return Value

If the *shutdown* call completes successfully, a value of 0 is returned. If an error occurs, a value of −1 is returned. Additional error detail will be contained in the external variable *errno*. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

[EBADF]	The argument *s* does not contain a valid socket number.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOTCONN]	The specified socket is not connected.

4.4.14 sioctl

NAME

sioctl — control the operations of sockets.

SYNTAX

#include <sioctl.h>

```
int     sioctl(s, request, argp)
int     s, request;
char    *argp; (or int    *argp)
```

DESCRIPTION

The *sioctl* call controls the operations of sockets. The call indicates whether an argument is an input or output argument and the size of the specific argument in bytes.

NOTE

The DECnet–VAXmate function call *sioctl* is not compatible with DECnet–ULTRIX systems. See Appendix G for information on how to transport DECnet–VAXmate programs that use *sioctl*.

Input Arguments

s specifies the number for a socket returned by the *socket* or the *accept* call.

request specifies the I/O control function to be used. The control levels are:

FIONREAD returns the total byte count of all messages waiting to be read. The argument *argp* points to an int.

FIONBIO sets or clears blocking or nonblocking I/O operation. The argument *argp* points to a byte that contains a value of 0 or 1. For blocking I/O, *argp* should point to a value 0. For nonblocking I/O, *argp* should point to a value of 1.

FIORENUM renumbers an assigned socket number to another number. In this way, the original socket number is made available again. The valid range for socket numbers is 0 to 31. The argument *argp* points to an int.

NOTE

The *select* function call cannot accept socket numbers that exceed this range. (See Section 4.4.10 for details.) If you specify a number that is already in use, an error message, EEXIST, is returned.

argp specifies the address of the argument list.

Output Argument

argp specifies the results of the socket operations.

Return Value

If the call completes successfully, a value of 0 is returned with the following additional message:

For FIONREAD, *argp* returns the total byte count of all messages waiting to be read.

If an error occurs, a value of –1 is returned. Additional error detail will be contained in the external variable *errno*. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

[EBADF]	The argument *s* does not contain a valid socket number.
[EEXIST]	The socket number is already in use.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[EOPNOTSUPP]	The socket type does not support the socket I/O operation.

4.4.15 socket

NAME

socket — create an endpoint for communication and return a socket number.

SYNTAX

```
#include <types.h>
#include <socket.h>
#include <dn.h>

int     socket(domain, type, protocol)
int     domain, type, protocol;
```

DESCRIPTION

The *socket* call creates a socket and returns a socket number. A socket is an addressable endpoint of communications within a task. It can be used to transfer data to or from a similar socket in another task. Subsequent function calls on this socket will refer to the associated socket number.

Input Arguments

domain specifies the communications environment as AF_DECnet.

type specifies the type of communication for the socket. For example, SOCK_STREAM. (See Appendix A for a list of defined socket types.)

SOCK_STREAM causes bytes to accumulate until internal DECnet buffers are full. The receiving task does not know how many bytes were sent in each write operation.

NOTE

To prevent a program from hanging on a stream socket, issue the *sioctl* call with the FIONREAD function argument (see Section 4.4.14), and then read only those numbers of bytes returned by the call. You should also perform socket operations in nonblocking I/O mode.

SOCK_SEQPACKET causes bytes to be sent immediately. The receiving task receives those bytes in one "record".

protocol specifies a particular DECnet protocol to be used with the socket. (See Appendix A for a list of supported DECnet layers.)

Return Value

If the call completes successfully, the socket number is returned. This number is used by subsequent system calls on this socket. If an error occurs, a value of –1 is returned. Additional error detail will be contained in the external variable *errno*. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

[EAFNOSUPPORT]	The specified domain is not supported in this version of the system.
[EMFILE]	Too many open sockets.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOBUFS]	No buffer space is available. The socket cannot be created.
[EPROTONOSUPPORT]	The specified protocol is not supported.
[ESOCKTNOSUPPORT]	The specified socket type is not supported in this address family.

4.4.16 sread

NAME

sread — read data from a socket.

DESCRIPTION

The *sread* call is used to read data from your peer. If no messages are available at the socket, the *sread* call waits for a message to arrive unless the socket is nonblocking. In this case, a status of −1 is returned with the external variable *errno* set to EWOULDBLOCK.

If the socket becomes disconnected, queued data can still be received from the broken logical link. However, if you shut down the socket or detach it, queued data cannot be received. When the logical link is not in a connected state, and all data has been read, the *sread* call returns zero bytes.

The *select* call can be used to determine if more data has arrived. (See Section 4.4.10.)

NOTE

The DECnet–VAXmate function call *sread* is not compatible with DECnet–ULTRIX systems. See Appendix G for information on how to transport DECnet–VAXmate programs that use *sread*.

The *sread* call performs the same function as the *recv* call (see Section 4.4.8) with one exception — you cannot set any flags.

SYNTAX

```
int     sread(s, buffer, buflen)
int     s, buflen;
char    *buffer;
```

Input Arguments

s specifies the number for a socket returned by the *socket* or the *accept* call.

buffer specifies the address of the buffer which will contain the received message.

buflen specifies the size of the message buffer.

Output Argument

buffer specifies the buffer which contains the received message.

Return Value

If the call succeeds, the number of read characters is returned.

If the call returns a zero, you have either received a zero length message or the logical link has been disconnected. To determine the state of the logical link, use the *getsockopt* function call with the *DSO_LINKINFO* option (see Section 4.4.12), or the DECnet utility function, *dnet_eof* (see Chapter 5). If the link has been disconnected, then all subsequent receives will return zero bytes.

If an error occurs, a value of –1 is returned. Additional error detail will be specified in the external variable *errno*. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

When reading **normal data**, the following set of error messages can occur:

Blocking I/O

Message	Description
[EBADF]	The argument *s* does not contain a valid socket number.
[ENETUNREACH]	The network is unreachable. The network process is not installed.

Nonblocking I/O

Message	Description
[EBADF]	The argument *s* does not contain a valid socket number.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[EWOULDBLOCK]	The receive operation would block because there is currently no data to receive.

4.4.17 swrite

NAME

swrite — write data to a socket.

SYNTAX

```
int     swrite(s, buffer, buflen)
int     s, buflen;
char    *buffer;
```

DESCRIPTION

The *swrite* call is used to write data to your peer.

If no message space is available at the socket to hold the message to be transmitted, then the *swrite* call will normally block. If the socket has been placed in nonblocking I/O mode, the message will not be sent, and the function will complete with the error EWOULDBLOCK.

The *select* call can be used to determine if more data may be sent over the socket. (See Section 4.4.10.)

NOTE

The DECnet–VAXmate function call *swrite* is not compatible with DECnet–ULTRIX systems. See Appendix G for information on how to transport DECnet–VAXmate programs that use *swrite*.

Input Arguments

s specifies the number for a socket returned by the *socket* or the *accept* call.

buffer specifies the address of the buffer which contains the outgoing message.

buflen specifies the size of the message.

Return Value

If the call succeeds, the number of sent characters is returned. If an error occurs, a value of –1 is returned. Additional error detail will be specified in the external variable *errno*. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

When sending **normal data**, the following set of error messages can occur:

Blocking I/O

Message	Description
[EBADF]	The argument *s* does not contain a valid socket number.
[EMSGSIZE]	The size of the outgoing message is more than 2048 bytes.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOTCONN]	The *swrite* call did not complete and the link was disconnected.
[EPIPE]	The link has been disconnected, aborted, or shut down. No further messages can be sent.

Nonblocking I/O

Message	Description
[EBADF]	The argument *s* does not contain a valid socket number.
[EMSGSIZE]	The size of the outgoing message is more than 2048 bytes.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOTCONN]	The *swrite* call did not complete and the link was disconnected.
[EPIPE]	The link has been disconnected, aborted, or shut down. No further messages can be sent.
[EWOULDBLOCK]	The outbound quota was full, and the message could not be sent.

5
DECnet Utility Functions

DECnet–VAXmate includes C language source files which are used to create a linkable library for DECnet–VAXmate applications. This library provides compatibility with the network socket interface supported by DECnet–ULTRIX.

The DECnet utility functions are contained in the C language source files. Some of these routines are used for accessing the network node database and manipulating the data.

Some of the DECnet utility functions include the DECnet header file <dnetdb.h>. This file provides DECnet definitions used with standard C functions.

5.1 Creating the DECnet–VAXmate Programming Interface Library

The file DNETLIB.SRC contains three types of files: the .C files (C language sources), the .H files (header files that contain definitions for the network interface) and the .ASM files (assembly language sources). You should refer to the appropriate installation guide for a complete list of these files.

In order to interface to the DECnet–VAXmate network process, you should create a library against which to link your DECnet–VAXmate program(s).

Use the following procedure to create a DECnet–VAXmate programming interface library:

1. The Break Source utility, BREAKSRC, allows you to break the source file, DNETLIB.SRC, into separate source files for compilation and assemblies. BREAKSRC is supplied with the DECnet–VAXmate distribution kit. When you run the DECnet–VAXmate Installation Procedure (DIP), you can select to have the DNETLIB.SRC file split into separate files. The BREAKSRC utility will then be run automatically for you. For instructions on how to run DIP, refer to the appropriate installation guide.

To run BREAKSRC, use the following format:

BREAKSRC <input_file_spec> <output_device:\path>

For example:

BREAKSRC A:DNETLIB.SRC C:\DECNET\SRC\

2. Use your C language compiler to compile each C language source module. Use your assembler to assemble each assembly source module.

3. After you produce an object module for each source module, build a library against which to link your DECnet–VAXmate applications programs.

5.1.1 DECnet–VAXmate Programming Considerations

The following programming considerations should be noted when writing and developing your DECnet–VAXmate applications:

1. Using the External Variable *errno* — Most DECnet–VAXmate programming interface functions use the external variable *errno* as a place to return error detail. It is assumed that *errno* has been defined externally to the programming interface as an *int*. It may already be defined in your C language run-time library; if not, your applications program should define it.

2. Checking Software Compatibility — When creating DECnet–VAXmate applications, make sure that you resolve any C language compiler incompatibilities before compiling the C language source modules such as long variable names or certain type definitions.

3. If your C compiler does not do so by default, you should compile the sources so that all data structures are stored without extra space (packed) for alignment of members on "int" boundaries.

4. Using Assembly Source Modules — There are assembly source modules included in DNETLIB.SRC. Before you can successfully call these functions from sources compiled by your C language compiler, you should fulfill any assembly-format requirements such as segment names. (Refer to the header file, *begin.h*, on the distribution kit as an example of specific C language compiler segment naming requirements.)

5. Using Specific Macros — There are references to the macros/functions such as *toupper* and *islower* in some of the C language source modules. It is assumed that your C language compiler has provided a standard macro/function for them. If not, you can simply provide your own macros/functions.

6. If you are not using a C compiler that supports the *signal* handling function (for example, CTRL/C trapping), then you will need either to provide your own *signal* function or to comment out from the code in *dnet_conn* any references to the *signal()* function and the reference to the include file, <signal.h>.

The CTRL/C trapping, provided in *dnet_conn*, allows you to abort a utility which appears to be waiting indefinitely for connections to complete. (This applies only to utilities that use *dnet_conn* for making connections.) The *connect* function in *dnet_conn* is invoked in a nonblocking socket mode.

If CTRL/C trapping code is commented out from *dnet_conn*, there is the potential problem of leaving hung sockets and running out of system resources, should users decide to CTRL/C in the middle of nonblocking connection request attempts.

7. If you are using a compiler that does not define the following variables: *int daylight*, *long timezone* and *char *tzname[2]*, either comment these external declarations out of the header file <time.h> or define them in your C program code.

8. The DECnet–VAXmate programming interface library was based on a 2-segment model: one code segment and one data segment.

9. Using Specific DECnet Function Calls — If you develop code that uses *dnet_getacc*, *dnet_installed* and *dnet_path* function calls, you will be unable to transport that code to a DECnet–ULTRIX system. These function calls are not valid on DECnet–ULTRIX systems.

CAUTION

If a program terminates with any active logical links (sockets), the links remain active. In this way, another program can start and use the same links.

If the logical links are not needed, you must issue the *sclose* function call before a program is terminated with an EXIT, CTRL/Z or CTRL/C. If too many links are left active, an error message, indicating that no more buffers are available or that there are insufficient network resources, is reported. This causes the network to become unusable. To completely deactivate any logical links, run the Network Control Program (NCP) and issue the SET KNOWN LINKS command with STATE OFF. (See the *DECnet–VAXmate User's Guide* for more information on this command.)

5.2 DECnet Utility Function Calls

The following sections describe the DECnet utility function calls for C programs. The calls are summarized in the table below:

Table 5–1: DECnet Utility Function Calls

DECnet Call	Description
bcmp	Compare byte strings.
bcopy	Copy *n* bytes from one specific string to another string.
bzero	Zero *n* bytes in a specific string.
dnet_addr	Convert an ASCII node address string to binary and return a pointer to a *dnnaddr* data structure.
dnet_conn	Connect to the specified target network object on a remote node and send along access control information and/or optional data.
dnet_eof	Test the current state of the connection.
dnet_getacc	Search the incoming access database file, DECACC.DAT, for access control information that is associated with a given user name. The access control information set by the NCP command SET ACCESS is stored in the database file, DECACC.DAT.
dnet_getalias	Return default access control information by node name.
dnet_htoa	Search the node database. If the node name is found, a pointer to the DECnet ASCII node name string is returned. Otherwise, a pointer to the DECnet ASCII node address string is returned. If the function fails to return a valid node name or address string, a pointer to the string "?unknown?" is returned.
dnet_installed	Perform an installation check on the specific software module.
dnet_ntoa	Specify a pointer to the *dnnaddr* data structure which contains the binary node address. If the function completes successfully, a pointer to the ASCII string representation of the DECnet node address is returned.
dnet_path	Return a modified file name which contains the DECnet database device and path name prefixed to the file name.
getnodeadd	Return the address of your local DECnet–DOS node.
getnodeent	Access the network node database and return complete node information given only a node address or node name.
getnodename	Return the ASCII string representation of your local DECnet–DOS node.
nerror	Produce DECnet error messages and output the ASCII text string to *stdout*.
perror	Produce an ULTRIX error message appropriate to the last detected system error, and output the ASCII text string to *stdout*.

5.2.1 bcmp

NAME

bcmp — compare byte strings.

SYNTAX

```
int  bcmp(s1, s2, n)

char      *s1, *s2;
int       n;
```

DESCRIPTION

bcmp compares byte strings to see if they are matching character strings. It is assumed that the strings are of equal length.

Input Arguments

s1 specifies the address of the first character string.

s2 specifies the address of the second character string.

n is the length of the strings.

Return Value

If a match is found, the value of 0 is returned. Otherwise, a nonzero value is returned.

5.2.2 bcopy

NAME

bcopy — copy *n* bytes from one specific string to another string.

SYNTAX

```
int   bcopy(s1, s2, n)

char        *s1,*s2;
int         n;
```

DESCRIPTION

bcopy copies *n* bytes from one specific string to another string.

Input Arguments

s1 is the character pointer to the source string.

s2 is the character pointer to the destination string.

n specifies the number of bytes to be copied.

Return Value

The number of bytes copied from the source string to the destination string is returned.

5.2.3 bzero

NAME

bzero — zeroes *n* bytes in a specified string.

SYNTAX

```
int   bzero(s1, n)

char        *s1;
int         n;
```

DESCRIPTION

bzero zeroes *n* bytes in a specified string.

Input Arguments

**s1*	is the character pointer to the specified string.
n	specifies the number of bytes to be zeroed.

Return Value

The number of bytes zeroed in the specified string is returned.

5.2.4 dnet_addr

NAME

dnet_addr — convert an ASCII node address string to binary and return a pointer to a *dn_naddr* data structure.

SYNTAX

struct dn_naddr *dnet_addr(cp)

char *cp;

DESCRIPTION

In area based networks, a DECnet node address includes an area number and a node number. The function call *dnet_addr* converts an ASCII node address string to binary and returns a pointer to a *dn_naddr* data structure. This information is required for the *sockaddr_dn* data structure. (See Appendix B on how these data structures are formatted.)

Input Argument

cp is the character pointer to the ASCII node address string. The DECnet node address is specified as *a.n*

where

a is the area number

n is the node number

Return Argument

dn_naddr specifies the node address data structure. A user retrieves data from the fields filled in by this function call. The fields are:

a_len specifies the length of the returned node address.

a_add specifies the node address.

If the call succeeds, a pointer to a *dn_naddr* data structure is returned. Otherwise, a null value is returned.

NOTE

If you plan to call this function again before you are finished using the data, you must copy the data into a local buffer.

5.2.5 dnet_conn

NAME

dnet_conn — connect to the specified target network object on a remote node and send along access control and/or optional user data.

SYNTAX

int dnet_conn(node, object, sock_type, out_data, out_len, in_data, in_len)

```
char      *node;
char      *object;
int       sock_type;
u_char    *out_data, *in_data,
int       out_len, *in_len;
```

DESCRIPTION

dnet_conn establishes a connection to the specified target DECnet object on a remote node. If no access control information is supplied as part of the node input argument, the default access control information (if found in the access control database) will be sent. Optional data can also be passed with the function.

dnet_conn supports password prompting based on the input node specification string. You are asked to supply a password whenever:

- The password field in the node specification is either a question mark (?) or an asterisk (*).
- The user field is present but the password field is missing.

The following example node specifications will cause prompting for passwords:

```
dnet_conn("boston/revere",...)
dnet_conn("boston/revere/?",...)
dnet_conn("boston/revere/*",...)
```

dnet_conn supports outgoing proxy log-in access. Outgoing proxy allows the local node to initiate proxy log-in access to the remote node, but does not allow proxy log-in access from the remote node to the local node. Before you are permitted to use proxy log-in, the following must take place:

- Proxy log-in access **must** be supported at the remote node.
- The Executor (local) node must have a user name set up in its node database. This user name is passed in outgoing connection requests and may be used for proxy log-in access. To set up access control information, refer to the discussion of the NCP utility in the *DECnet-VAXmate User's Guide*.

If access control information is not explicitly supplied with an input node name, *dnet_conn* will check the node database for implicit access control information. If implicit access control does not exist, proxy login will take place with the Executor node's user name and passed in the outgoing connection request.

The following examples show the use of proxy log-in access with the *dnet_conn* function call:

The Executor (local) node has set up TASHA as the user name in the local node's database. The remote node BOSTON is stored as an entry in the local node's database without any access control information.

1. No proxy, null access control information is explicitly specified:

   dnet_conn("BOSTON//", . . .)

2. Proxy will be passed, no explicit access control information, no implicit access control information for node BOSTON in the database:

   dnet_conn("BOSTON", . . .)

The Executor (local) node has set up TASHA as the user name. The remote node BOSTON is stored as an entry in the local node's database with TASHA as the user name/access control information.

1. No proxy, null access control information is explicitly specified:

   dnet_conn("BOSTON//", . . .)

2. No proxy, the implicit access control information for node BOSTON will be used:

   dnet_conn("BOSTON", . . .)

A target task can return a 1- to 16-byte optional data message when it accepts or rejects the connection request.

When a program which uses *dnet_conn* fails to complete a connection request, *nerror* can subsequently be called in order to display the DECnet error message.

Input Arguments

node specifies the address of the string which contains the remote node name or address and any access control information. Node names are always converted to uppercase before being processed by this function. Access control information is passed as supplied **with** regard to case.

To pass access control information, the node string can take one of the following formats:

"*node_name/user/password/account*"

or

"*area.node_number/user/password/account*" To pass null access control information, the node string can take one of the following formats:

"*node_name//*"

or

"*area.node_number//*"

object specifies the address of the string which contains the specified target DECnet object. The object string can be specified in one of the following ways:

1. To access a target object by supplying an object name. You should note that the object name is passed as supplied, **with** regard to case.

   "*target_object_name*"

2. To access a target object by object number.

   "*#target_object_number*"

sock_type is 0 when creating a sequenced socket packet and 1 when creating a stream socket.

out_data specifies the address of the outgoing optional user data buffer. If not required, supply a null pointer.

out_len specifies the size of the optional outgoing message. The message length can be up to 16 bytes. If not required, supply a null value.

in_data specifies the address of the buffer which will store the optional incoming message. If not required, supply a null pointer.

in_len specifies the address of the location in which the value will be stored. It should initially contain the size of the buffer pointed to by *in_data*. On return, it will contain the actual size of the optional incoming message. If not required, supply a null pointer.

Return Value

If the function completes successfully, the socket number is returned. If an error occurs, a value of –1 is returned. Additional error detail will be contained in the external variable *errno*. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

[E2BIG]	An argument is too long.
[EACCES]	Access control information was rejected.
[EADDRNOTAVAIL]	The node name is undefined.
[EDESTADDRREQ]	A specific destination address is required.
[ENAMETOOLONG]	The node name is invalid.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ESRCH]	The object is unknown.

NOTE

Additional errors may be returned by the *socket*, *setsockopt*, and the *connect* function calls which are called from within *dnet_conn*.

5.2.6 dnet_eof

NAME

dnet_eof — test the current state of the connection.

SYNTAX

int dnet_eof(sock)

int sock;

DESCRIPTION

dnet_eof tests the current state of the connection.

Input Argument

sock specifies the socket whose connection is to be checked.

Return Value

If the connection is determined to be active (either a running or a connecting state), a value of 0 is returned. If the connection is inactive, a value of 1 is returned.

NOTE

If a bad socket number is supplied with *dnet_eof*, a value of 1 is returned. However, this value is not a true indication of an invalid socket number.

5.2.7 dnet_getacc

NAME

dnet_getacc — searches the incoming access database for access control information that is associated with a given user name.

SYNTAX

```
struct  dnet_accent  *dnet_getacc(nacc)
struct  dnet_accent  *nacc;
```

DESCRIPTION

dnet_getacc searches the incoming access database file, DECACC.DAT, for access control information that is associated with a given user name. The access control information set by the NCP command SET ACCESS is stored in the database file, DECACC.DAT.

NOTE

If you develop code that uses *dnet_getacc*, you will be unable to transport that code to a DECnet-ULTRIX system. This call is not valid on DECnet-ULTRIX systems.

Input Argument

nacc is a character pointer to an access control information block containing the user name to be matched. The string consists of 1 to 39 alphabetic characters. This string must be identical to the user string set up by the NCP command SET ACCESS. (Use the NCP command SHOW KNOWN ACCESS to display the string.) Refer to the *DECnet-VAXmate User's Guide* on using these commands.

Return Argument

nacc specifies the access control information block data structure *dnet_accent*. A user retrieves information filled in by this function call. (See Appendix B on how *dnet_accent* is formatted.)

The following data fields can be filled in by this function call:

acc_status is used internally by this function call.

acc_type specifies the type of privilege associated with a user name. The four access types are: 0 for no access rights, 1 for read only access, 2 for write only access, and 3 for read and write access.

acc_user specifies the user name. It consists of a 1- to 39-alphabetic character string terminated by a null character.

acc_pass specifies the password associated with a user name. It consists of a 1- to 39-alphabetic character string terminated by a null character.

If there is a match, a character pointer to the access control information block associated with the user string is returned. The access type is always returned along with any user and password information.

A value of 0 is returned if there is no match or if the DECnet database path cannot be found for DECACC.DAT using the function *dnet_path*. (See Section 5.2.12.)

NOTE

If you plan to use this function again before you are finished using the data, you must copy the data into a local structure.

5.2.8 dnet_getalias

NAME

dnet_getalias — returns default access control information by node name.

SYNTAX

```
char        *dnet_getalias(node)
char        *node;
```

DESCRIPTION

dnet_getalias retrieves any default access control information associated with a specific node.

Input Argument

node is a character pointer to a node name. The node name is forced to uppercase and then processed by the function.

Return Argument

If the node has default access control information associated with it, the node name followed by the access data is retrieved. The extended node name string is returned as *node_name/user/password/account*. This data was set up in the permanent databases, DECNODE.DAT and DECALIAS.DAT, using the Network Control Program (NCP).

If no access control data can be found, a null pointer is returned.

NOTE

If you plan to call this function again before you are finished using the data, you must copy the data into a local buffer.

5.2.9 dnet_htoa

NAME

dnet_htoa — search the node database. If the node name is found, a pointer to the DECnet ASCII node name string is returned. Otherwise, a pointer to the DECnet ASCII node address string is returned. If the function fails, a pointer to the string "?unknown?" is returned.

SYNTAX

```
char        *dnet_htoa(add)
struct      dn_naddr *add;
```

DESCRIPTION

dnet_htoa searches the node database. If the node name is found, a pointer to the DECnet ASCII node name string is returned. If the node name is not found, a pointer to the DECnet ASCII node address string is returned.

Input Argument

add specifies a pointer to a structure of the type *dn_naddr*, which contains the node address. The format is *area.number*. (Refer to Appendix B on how the *dn_naddr* data structure is formatted.)

Return Value

If the node name is found, a pointer to the DECnet ASCII node name string is returned. Otherwise, a pointer to the DECnet ASCII node address string is returned.

If the function call fails to return a valid node name or address string, a pointer to the string "?unknown?" is returned.

NOTE

If you plan to call this function again before you are finished using the data, you must copy the data into a local structure.

5.2.10 dnet_installed

NAME

dnet_installed — perform an installation check on a specific software module.

SYNTAX

```
int     dnet_installed(vector, tla)
short   vector;
char    *tla;
```

DESCRIPTION

dnet_installed performs an installation check on a specific software module. You supply the interrupt vector number and the 3-letter acronym for the software module.

If the software module is installed, the 2-byte software version number is returned. The low byte is the major version number. The high byte is the minor version number. Otherwise, a value of –1 is returned.

NOTE

If you develop code that uses *dnet_installed*, you will be unable to transport that code to a DECnet-ULTRIX system. This function call is not valid on DECnet-ULTRIX systems.

Input Arguments

vector specifies the interrupt vector number for the software module. Appendix A lists the interrupt vector numbers for the defined software modules.

tla is the 3-letter acronym for the software module. The acronym is passed as supplied, with regard to case. For example, DNP is the acronym for the DECnet Network Process. It must be passed as uppercase. See Appendix A for a list of defined software modules.

Return Value

If the installation check successfully completes, the 2-byte software version number is returned. Otherwise, a value of –1 is returned.

5.2.11 dnet_ntoa

NAME

dnet_ntoa — convert a DECnet node address from binary form to ASCII form.

SYNTAX

```
char        *dnet_ntoa(add)
struct      dn_naddr *add;
```

DESCRIPTION

dnet_ntoa converts a DECnet node address from binary form to ASCII form.

Input Argument

add specifies a pointer to a structure of the type *dn_naddr*, which contains the binary node address. (See Appendix B on how the *dn_naddr* data structure is formatted.)

Return Value

If the function completes successfully, a pointer to the ASCII string representation of the DECnet node address is returned. The format is *area.number*.

If the function call fails to return a valid node name or address string, a pointer to the string "?unknown?" is returned.

NOTE

If you plan to call this function again before you are finished using the data, you must copy the data into a local structure.

5.2.12 dnet_path

NAME

dnet_path — return a modified file name that contains the DECnet database device and path name prefixed to the specified input file name.

SYNTAX

```
char    *dnet_path(file_name)
char    *file_name;
```

DESCRIPTION

Given a character string pointer to a file name, *dnet_path* returns a modified file name that contains the DECnet database device and path name prefixed to the specified input file. It is recommended that all user-created DECnet–VAXmate database files be located in the same DECnet directory.

For example:

Call:

```
new_file_name = dnet_path("NCPHELP.BIN");
```

Returns:

```
new_file_name = "c:\decnet\NCPHELP.BIN"
```

The DECnet database device and path name should be specified as input arguments to the DLL and/or DNP command lines in your AUTOEXEC.BAT file. To do this, edit your AUTOEXEC.BAT file using EDLIN or a similar text editor.

IMPORTANT

The *dnet_path* function call differs from Version 1.0. You can no longer set the DECnet database device and path names using the NCP command SET EXECUTOR. However, you can still display the database path specification. To do this, issue the NCP command SHOW EXECUTOR CHARACTERISTICS. For information on using this command, refer to the *DECnet–VAXmate User's Guide*.

If you develop code that uses *dnet_path*, you will be unable to transport that code to a DECnet-ULTRIX system. This function call is not valid on DECnet-ULTRIX systems.

When the system is rebooted, DNP and/or DLL will use its command line argument or default to the current device:\decnet as the path specification.

To change the DECnet database path, you will need to specify a new path specification as command line input the next time that DLL and/or DNP are installed.

DECnet–VAXmate Version 1.0 supports Ethernet and asynchronous DDCMP configurations. The following examples illustrate acceptable ways for specifying the DECnet database path.

If you have an Ethernet setup.

Example 1: The Scheduler (SCH), Data Link layer (DLL) and DECnet Network Process (DNP) files are to be installed. The DLL and DNP command lines include a defined DECnet database path. The DLL path is used.

```
.
.
.
SCH
DLL c:\decnet\v11
DNP c:\decnet\v11
```

Example 2: SCH, DLL and DNP files are to be installed. Only the DLL command line includes a defined DECnet database path. A database path is not defined for DNP. The database path set for DLL will also be used by DNP.

```
.
.
.
SCH
DLL c:\decnet\v11
DNP
```

Example 3: SCH, DLL and DNP files are to be installed. The DLL and DNP command lines do not include defined database paths. For this setup, the default DECnet database path (current device:\decnet) set for DLL will also be used by DNP.

```
.
.
.
SCH
DLL
DNP
```

If you have an asynchronous DDCMP setup.

Example 1: SCH and DNPDCP files are to be installed. The DNPDCP command line includes a defined DECnet database path. The DLL file is not required for asynchronous operations.

```
.
.
.
SCH
DNPDCP c:\decnet\v11
```

Example 2: SCH, and DNPDCP files are to be installed. A DECnet database path is not specified for DNPDCP. Therefore, the default database path (current drive:\decnet) will be used by DNPDCP .

```
.
.
.
SCH
DNPDCP
```

Return Value

If the call completes successfully, a pointer to a modified file name that contains the DECnet database device and path name prefixed to the file is returned.

5.2.13 getnodeadd

NAME

getnodeadd — return the address of your local DECnet–VAXmate node.

SYNTAX

struct dn_naddr *getnodeadd( );

DESCRIPTION

getnodeadd returns a pointer to a *dn_naddr* data structure which contains the DECnet node address of the local DECnet–VAXmate node.

Return Argument

dn_naddr specifies the node address data structure. A user retrieves data from the fields filled in by this function call. The fields are:

a_len specifies the length of the returned DECnet-VAXmate node address.

a_add specifies the DECnet–VAXmate node address.

If the call succeeds, a pointer to a *dn_naddr* data structure is returned. If an error occurs, a null value is returned.

NOTE

The *dn_naddr* data structure will be reused by additional calls. To keep this information, you must copy the data into a local structure.

5.2.14 getnodeent

NAME

getnodeent, getnodebyaddr, getnodebyname, setnodeent, endnodeent — access the network node database and return complete node information given only a node address or node name.

SYNTAX

```
#include <dnetdb.h>

struct    nodeent    *getnodeent( )

struct    nodeent    *getnodebyname(name)
char      *name;

struct    nodeent    *getnodebyaddr(addr, len, type)
char      *addr;
int       len, type;

setnodeent(stayopen)

int       stayopen;

endnodeent( )
```

DESCRIPTION

These functions access the network node database and return complete node information given only a node address or node name. Each *getnodeent*, *getnodebyname* and *getnodebyaddr* function returns a pointer to a single static *nodeent* structure. This structure contains the broken out fields of an entry in the network node database.

getnodeent returns a pointer to the next entry of the database.

setnodeent positions you at the beginning of the database. If the *stayopen* flag is set to nonzero, the node database is not closed after each call to *getnodeent* (either directly, or indirectly through one of the other "getnode" calls).

endnodeent closes the database file.

getnodebyname and *getnodebyaddr* sequentially search from the beginning of the database until a matching node name or node address is found or until the end of the database is encountered. Node names are stored in the network node database as uppercase. Therefore, comparisons of node name strings to node names stored in the database are forced to be uppercase. Node addresses are always arranged in ascending numeric order.

Input Arguments

name specifies the address of the buffer containing the DECnet node name string.

addr specifies the address of the buffer containing the DECnet node address string.

len is the length of the node's address string in bytes.

type is the address type AF_DECnet.

stayopen specifies a call dependent argument. If set to nonzero, the network node database is kept open for subsequent "getnode" calls.

Return Value

If the function, other than *setnodeent* and *endnodeent*, completes successfully, the address for the *nodeent* structure is returned.

nodeent specifies the node address database structure. A user retrieves data from the fields filled in by this function call. The following data fields can be modified:

- **n_name* points to a string which is the name of the DECnet node.
- *n_addrtype* specifies the address type AF_DECnet.
- *n_addr* specifies the DECnet network address for the node.

If an error or an EOF occurs, a null pointer is returned.

If *setnodeent* completes successfully, a value of 0 is returned. Otherwise, a value of –1 is returned.

NOTE

The *nodeent* structure will be reused by additional calls. To keep this information, you must copy the data into a local structure.

5.2.15 getnodename

NAME

getnodename — return the DECnet node name of your local DECnet–VAXmate node.

SYNTAX

char *getnodename( );

DESCRIPTION

getnodename returns the ASCII string representation of your local DECnet–VAXmate node name.

Return Value

If the function call is successful, your local DECnet node name is returned. Otherwise, a null pointer is returned.

5.2.16 nerror

NAME

nerror — produce DECnet system error messages.

SYNTAX

nerror(s)

char *s;

DESCRIPTION

nerror produces DECnet error messages by mapping standard *errno* values to their equivalent DECnet error messages. First the characters pointed to by *s* are output to *stdout*, followed by a colon, and then the resulting DECnet error text. The error number is taken from the external variable, *errno*, which is set when an error occurs.

If a program makes a call to *dnet_conn* which fails, *nerror* should be subsequently called in order to display the DECnet error text.

Input Argument

**s* is the character pointer to the ASCII text string to be displayed before the DECnet error text is displayed.

Output Argument

The characters pointed to by *s* are output to *stdout*, followed by the colon, and then the resulting DECnet error text. You should refer to Section 5.2.5 for the list of DECnet error messages returned by *dnet_conn*.

NOTE

This call is not ULTRIX compatible. For the ULTRIX call, the log is output to *stderr*.

5.2.17 perror

NAME

perror — produce a standard ULTRIX error message appropriate to the last detected system error.

SYNTAX

perror(cp)

char *cp;

DESCRIPTION

perror produces a standard ULTRIX error message appropriate to the last detected system error. First the character string is output, followed by a colon, and then the standard ULTRIX error message indexed by *errno*.

For example,

```
perror("Last error was ")
```

Input Argument

**cp* is the character pointer to the ASCII text string to be displayed before the ULTRIX error text is displayed.

NOTE

The log from *perror* is output to *stdout*. This call is not ULTRIX compatible. For the ULTRIX call, the log is output to *stderr*.

6
Assembly Language

Application programs written in assembly language can establish logical links and exchange data over the network. Nontransparent task-to-task communication requires specific network process interface calls. The implementation of these programming calls is discussed in this chapter.

6.1 DECnet–VAXmate Network Process

The DECnet–VAXmate network process is a terminate and stay resident task which is loaded into memory. It remains in memory once it is run. The DECnet–VAXmate network process supports blocking and nonblocking synchronous and asynchronous I/O.

6.2 DECnet Network Process Installation Check

Before accessing the DECnet network process, you should check to see if the network process has been installed. You must first issue the MS–DOS interrupt function call 35H (Get Interrupt Vector) with the DNP vector number as input. This function call returns a pointer to the DNP interrupt entry point.

On entry, the AH register contains the hexadecimal code, 35H. The AL register contains a hexadecimal interrupt number. For example, 6EH is the interrupt number for the DECnet network process. On return, the ES:BX register contains the CS:IP interrupt vector for the specified interrupt.

The interrupt function call 35H is described here:

On Entry	8086/8088 Register Contents
AH	35H (hexadecimal function code)
AL	6E (interrupt vector number for the DECnet network process)

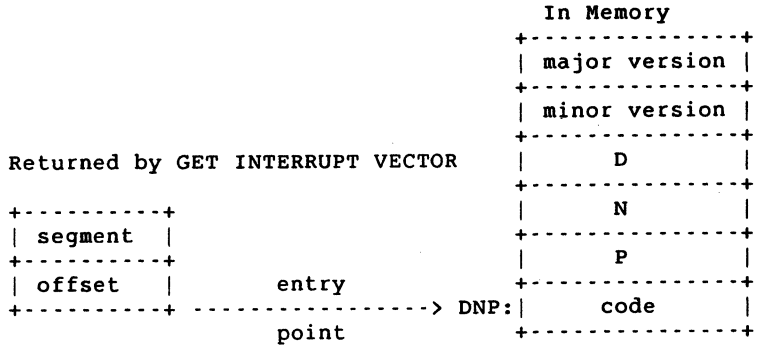
On Return	8086/8088 Register Contents
ES:BX	pointer to the interrupt handling routine

Figure 6–1: MS–DOS Interrupt Function Call 35H, Get Interrupt Vector

Using the interrupt vector that identifies the DECnet network process (DNP) module, you can examine the contents of the five bytes that precede the DNP entry point. Use the following procedure:

- Check the first 3 bytes for the 3-letter acronym for the software module. In this case, the acronym is DNP.
- Examine the 2-byte software version number. The lo byte is the major version number, and the high byte is the minor version number.

The installation check procedure is described here:



6.3 Using the I/O Control Block

All I/O to the network process is handled by way of DECnet–VAXmate network process interface calls. A data structure called the I/O Control Block (IOCB) transfers data to or from the network process. The address of the IOCB data structure is passed directly to the network process.

To issue a network process interface call, you must first create an IOCB for that call. Refer to the individual call descriptions (Sections 6.7.1 to 6.7.20) on how to set up the IOCB. To pass information to the DECnet network process, you must issue the interrupt function call 6EH. Use this call for the following I/O requests to the network process:

- Set up data in an IOCB.
- Move the address of the IOCB into the DS:DX register.
- Move the DECnet network process code, DE, into the AH register.
- Move a value of 1 into the AL register as the function code for an IOCB function request.
- Issue the interrupt function call 6EH.

On return, status information is returned in the IOCB for the network process interface call addressed by this interrupt function. If the call 6EH completes successfully, the IOCB's data member, *io_status*, returns a value of 0. You should refer to the individual call descriptions for details on any returned data.

In all I/O modes, if the call 6EH is unsuccessful, *io_status* returns a –1 value. In asynchronous mode, if the call does not complete, a value of –2 is returned. Another IOCB data member, *io_errno* returns additional error detail. See Appendix C for a list of error conditions.

The function call 6EH is described here:

On Entry	8086/8088 Register Contents
AH	DE (DECnet network process hexadecimal code)
DS:DX	IOCB address
AL	1 (IOCB Function request)

Figure 6–2: Interrupt Function Call 6EH, IOCB Request

6.3.1 I/O Control Block Structure

The I/O Control Block consists of a header substructure and a parameter list. The members of the IOCB header are listed in the following table:

Table 6–1: IOCB Header Data Members

Member	Size	Data Contents
io_fcode	1 byte	network process call specific function code (see Section 6.7)
io_socket	2 bytes	socket number
io_flags	2 bytes	flag option
io_status	2 bytes	returned status value
io_errno	2 bytes	returned error value
io_psize	2 bytes	parameter list size or buffer size

The IOCB parameter list depends on the particular network process interface call. It may contain one of the following members:

Table 6–2: IOCB Parameter List Members

Member	Size	Data Contents
attach_dn	12 bytes	socket creation data
io_buffer	4 bytes	data buffer offset
listen_dn	2 bytes	maximum number of incoming connects
localinfo_dn	20 bytes	local node information
select_dn	16 bytes	socket I/O descriptors
shutdown_dn	2 bytes	type of logical link shutdown
sioctl_dn	8 bytes	socket I/O control
sockaddr_dn	26 bytes	socket definitions
sockopt_dn	12 bytes	socket options

The members of an IOCB data structure are illustrated here:

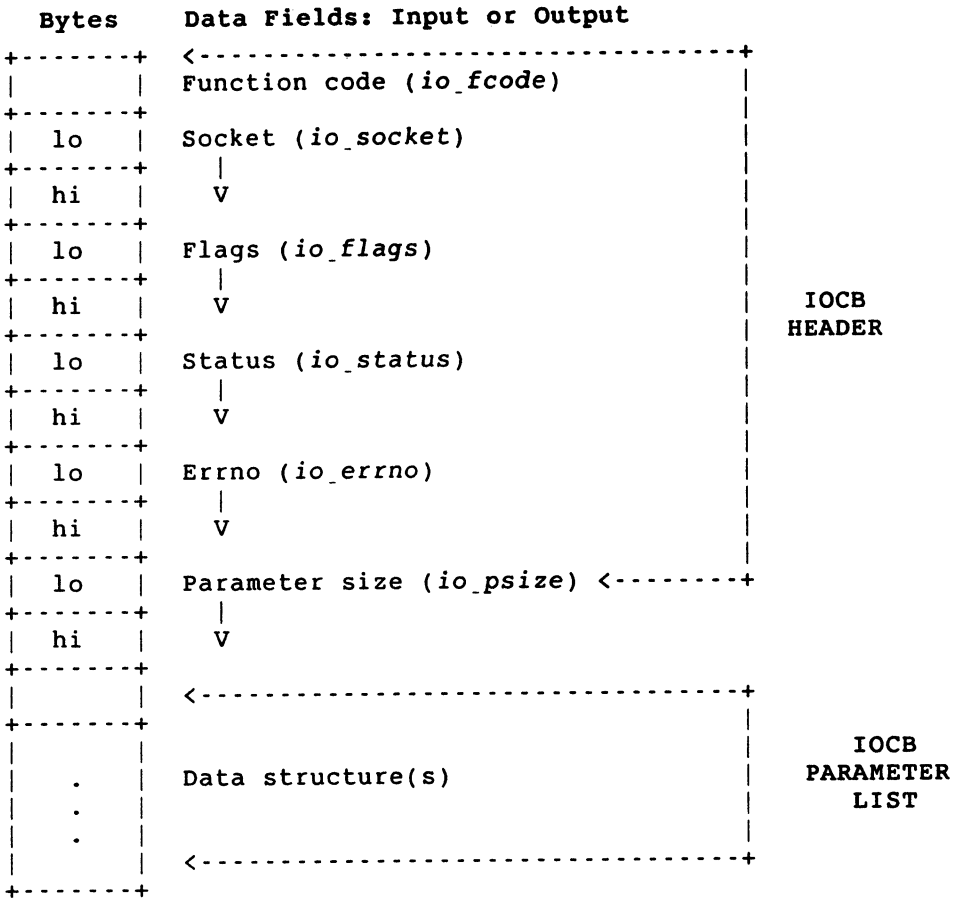


Figure 6–3: An IOCB Data Structure

6.4 Using the Callback I/O Control Block

To request a callback routine when specific calls complete, you must set up a data structure called the Callback I/O Control Block (CIOCB). The address of the CIOCB data structure is passed directly to the DECnet network process. To issue a DECnet–VAXmate call with a callback routine, you must first create a CIOCB for that call. Refer to individual call descriptions on how to set up the CIOCB. To pass information to the DECnet network process, issue the interrupt function call 6EH. (Refer to Section 6.3 for a description of the function call 6EH.)

- Set up data in a CIOCB.
- Move the address of the CIOCB into the DS:DX register.
- Move the DECnet network process code, DE, into the AH register.
- Move a value of 1 into the AL register as the function code for a CIOCB function request.

6.4.1 Callback I/O Control Block Structure

The Callback I/O Control Block consists of a header substructure, a parameter list, and the address of the callback routine. The CIOCB uses the same header substructure and parameter list as the IOCB. Their data members are listed in Tables 6–1 and 6–2. The additional data member is the address of the callback routine which requires 4 bytes.

Its location along with the other members of the CIOCB are illustrated here:

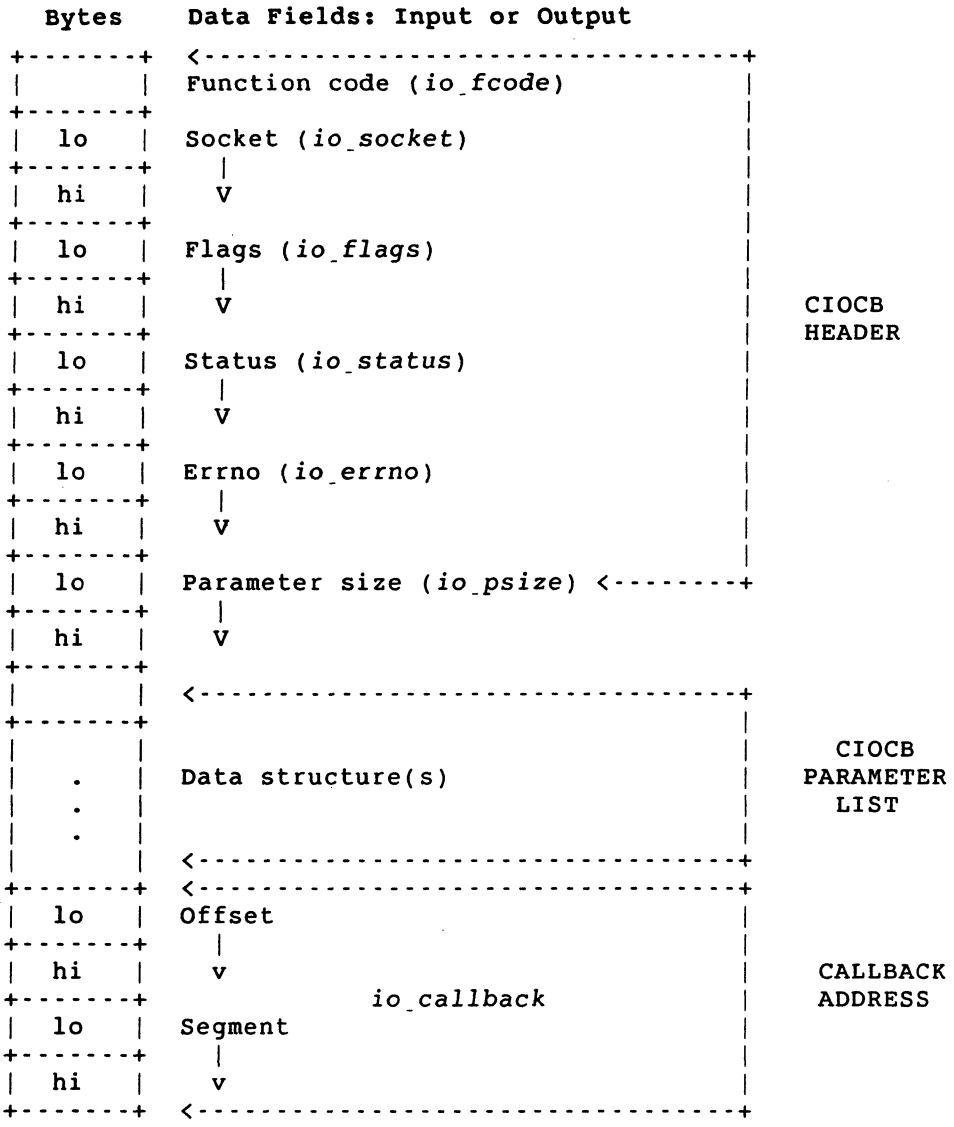


Figure 6–4: A CIOCB Data Structure

6.5 Synchronous I/O and Asynchronous I/O

DECnet–VAXmate supports three ways of handling network I/O: blocking synchronous, nonblocking synchronous, and asynchronous I/O. Each method is independent of the others.

When a DECnet–VAXmate call is issued in **blocking synchronous** mode, the software issuing the call does not regain control until the call completes or has failed. When **nonblocking synchronous** I/O is set, socket operations return to the calling program after the operation has been started, but not necessarily completed. Some operations must be restarted. If on the other hand, the operation can be performed immediately, it will either complete or fail. Should it fail, an error reason such as EWOULDBLOCK or ENOBUFS will be reported.

When **asynchronous** I/O is used, control returns to the calling program immediately after the DECnet network process records the call request. The network process may complete the request immediately or wait for a later time. There are two ways for you to check the status of an asynchronous DECnet–VAXmate call: use a callback routine and/or poll for status.

When using asynchronous I/O, the calling program can request that a routine be called upon completion. This routine is referred to as the **callback routine**. Callback routines can only be implemented with the asynchronous form of a function call. Asynchronous callbacks are valid for the following seven calls. It is also valid to issue these calls asynchronously without a callback as well as synchronously. The calls are:

- ACCEPT (Section 6.7.2)
- CONNECT (Section 6.7.6)
- RCVD (Section 6.7.12)
- RCVOOB (Section 6.7.13)
- SELECT (Section 6.7.14)
- SEND (Section 6.7.15)
- SENDOOB (Section 6.7.16)

NOTE

Information pertaining to the asynchronous form of function calls is indicated by the caption **For Asynchronous Mode**.

The function calls not appearing in this list can be issued asynchronously without a callback routine. Due to the way that the network process handles these calls, they will still complete immediately.

If your program does not implement callback routines, you should poll for status. To do this, examine the *io__status* field in the IOCB. A value of either –2, 0 or –1 may be returned. A value of –2 indicates pending or no change in status. When the call successfully completes, a value of 0 is returned in the *io__status* field. For some calls, you will be able to retrieve data from a data structure which is filled in by the call.

If the call is unsuccessful, *io_status* returns a value of –1. Additional error reason will be contained in the *io_errno* field. The error messages are listed in the DIAGNOSTICS section under each call description.

You can use callbacks and also poll for status. When you do this, the following status values can be returned:

Status Value	Meaning
–2	No change in status. The callback routine has not been called.
0	The call has completed. The callback routine was already called.
–1	The callback was already called, but there was an error in completing the call.

6.5.1 Using a Callback Routine

When you write a program that uses a callback routine, follow these guidelines:

- There is a valid stack, but not the user's stack.
- The address of the IOCB is in the DS:DX register.
- The address of the IOCB is on the stack.
- You preserve all registers except for the AX, BC, CX and DX registers.
- You must be sure to do a far return.
- When you implement callback routines, you cannot do any MS–DOS function calls or synchronous network I/O inside the callback routine. You can still do asynchronous network I/O.

6.5.2 Setting the *io_flags* Field

You can set more than one bit in the *io_flags* field which is included in the I/O Control Block. Bits for asynchronous I/O and callback routines can be used in conjunction with other bits for "peeking" at messages and for sending and receiving a multi-part message. How these bits are used with a specific function call appears in the individual call descriptions. Additional information about these flag options appears in Appendix A.

6.6 Using Socket Numbers with DECnet Network Process Interface Calls

The network process uses socket numbers to identify a particular I/O session. A socket number must be assigned by the user before proceeding with further network I/O. To assign a socket number, issue the ATTACH function call. (See Section 6.7.3 for a call description.) The assigned socket number is then used in the IOCB for subsequent network calls. The DETACH function call instructs DNP to deallocate any network resources associated with the attached socket and to make the socket and resources available again for network I/O.

6.7 Network Process Interface Calls

The following sections describe the synchronous form of the network process interface calls used for assembly language programs. An IOCB is passed to the network process with each call. Each IOCB contains a header and a parameter list. The members of the IOCB header are detailed with each call. For some calls, the IOCB also includes a data structure or a buffer address. The type of data structure is determined by the specific call. The members of the data structures are detailed in a similar manner.

Information pertaining to the asynchronous form of certain function calls is indicated by the caption **For Asynchronous Mode**. Refer to Section 6.5 for a list of function calls that have an asynchronous and a synchronous form.

The assembly language network process interface calls are summarized in the following table:

Table 6–3: Assembly Language Network Process Interface Calls

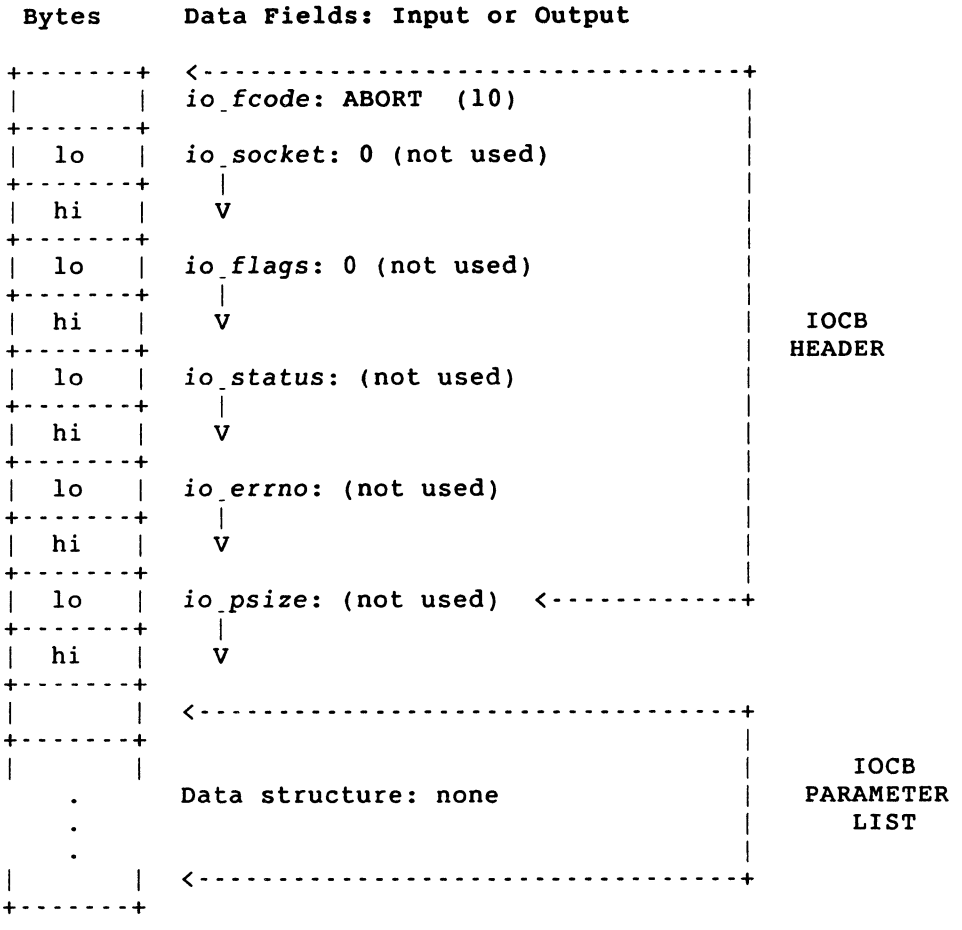
Decimal Value	Function	Description
10	ABORT	Disconnect all logical links (if any) and detach the sockets that do not have the option SO_KEEPALIVE set.
5	ACCEPT	Accept an incoming connection request on a socket, and return a socket number.
0	ATTACH	Create a socket and attach a socket number.
2	BIND	Assign an object name or number to a socket.
4	CONNECT	Initiate a connection request on a socket.
1	DETACH	Disconnect all associated active logical links, and detach the specified socket and any associated sockets, only if the option SO_KEEPALIVE is not set.
6	DISCONNECT	Disconnect from the peer socket, and terminate the logical link connection.
26	GETSOCKOPT	Get the options associated with sockets.
3	LISTEN	Listen for pending connections on a socket.
22	LOCALINFO	Retrieve network information for the local node.
16	PEERADDR	Retrieve information about your peer socket.
8	RCVD	Receive data on a specified socket.
13	RCVOOB	Receive out-of-band messages on a specified socket.
23	SELECT	Check the I/O status of the network sockets.
9	SEND	Send data on a specified socket.
14	SENDOOB	Send out-of-band messages on a specified socket.
25	SETSOCKOPT	Set options associated with sockets.
7	SHUTDOWN	Shut down part or all of a full duplex logical link connection.
24	SIOCTL	Control the operations of open sockets.
15	SOCKADDR	Retrieve information set by the *BIND* call for the specified socket.

6.7.1 ABORT

NAME

ABORT – disconnect all logical links (if any) and detach sockets that do not have the option SO_KEEPALIVE set.

IOCB Data Members



DESCRIPTION

The *ABORT* call disconnects all logical links (if any) and detaches the sockets that do not have the option SO_KEEPALIVE set. Refer to Section 6.7.17 for a discussion of SO_KEEPALIVE.

NOTE

Before you can terminate a connection over a socket with the option SO_KEEPALIVE set, you must first issue a *SETSOCKOPT* call with SO_KEEPALIVE turned off. To turn off SO_KEEPALIVE, you must precede SO_KEEPALIVE with the NOT operator. (NOT SO_KEEPALIVE is the default condition.)

You are then able to issue the *ABORT* call. The logical links (if any) are disconnected, and the socket is detached. However, if you issue *ABORT* without turning off SO_KEEPALIVE, the socket remains attached, and the links (if any) stay active.

Input Data

io_fcode specifies *ABORT* as the function code. It has a decimal value of 10.

io_socket specifies the socket number created by the *ACCEPT* or the *ATTACH* call. It is not used with the *ABORT* call. All logical links are disconnected by the *ABORT* call.

io_flags defines specific flag options. You must set this data member to 0. It is not used with the *ABORT* call.

io_psize specifies the size of a data structure. This data member is not used with the *ABORT* call.

Output Data

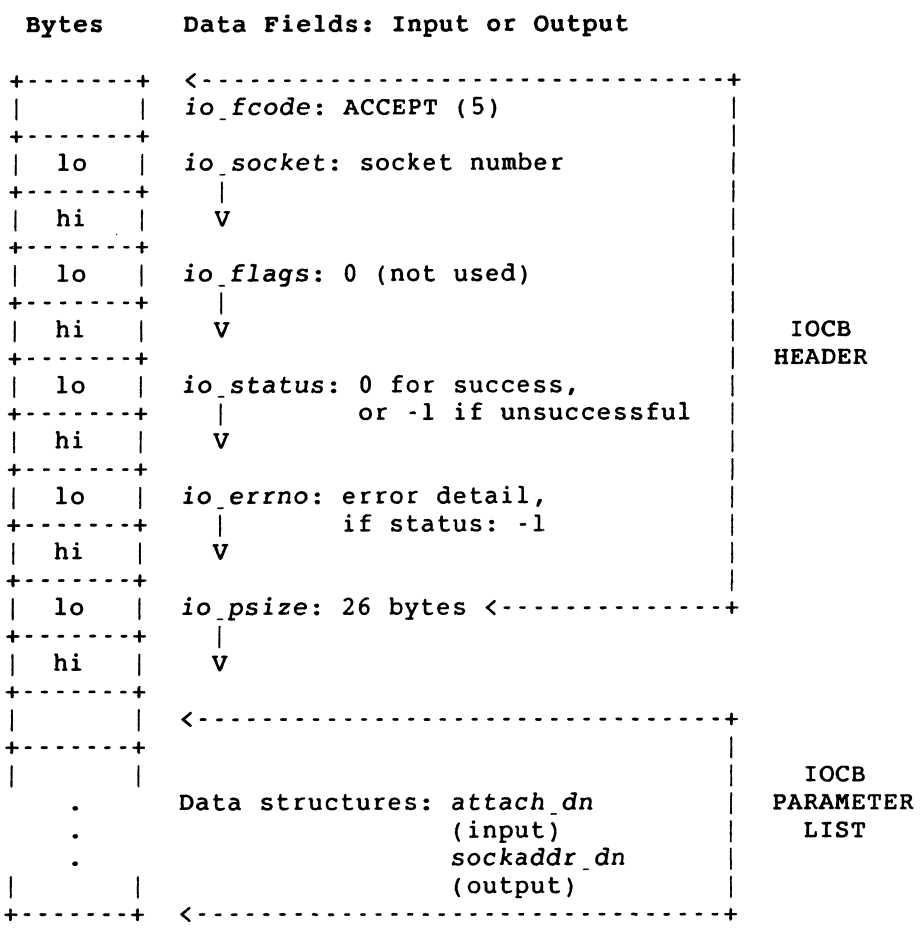
There is no output data for the *ABORT* call.

6.7.2 ACCEPT

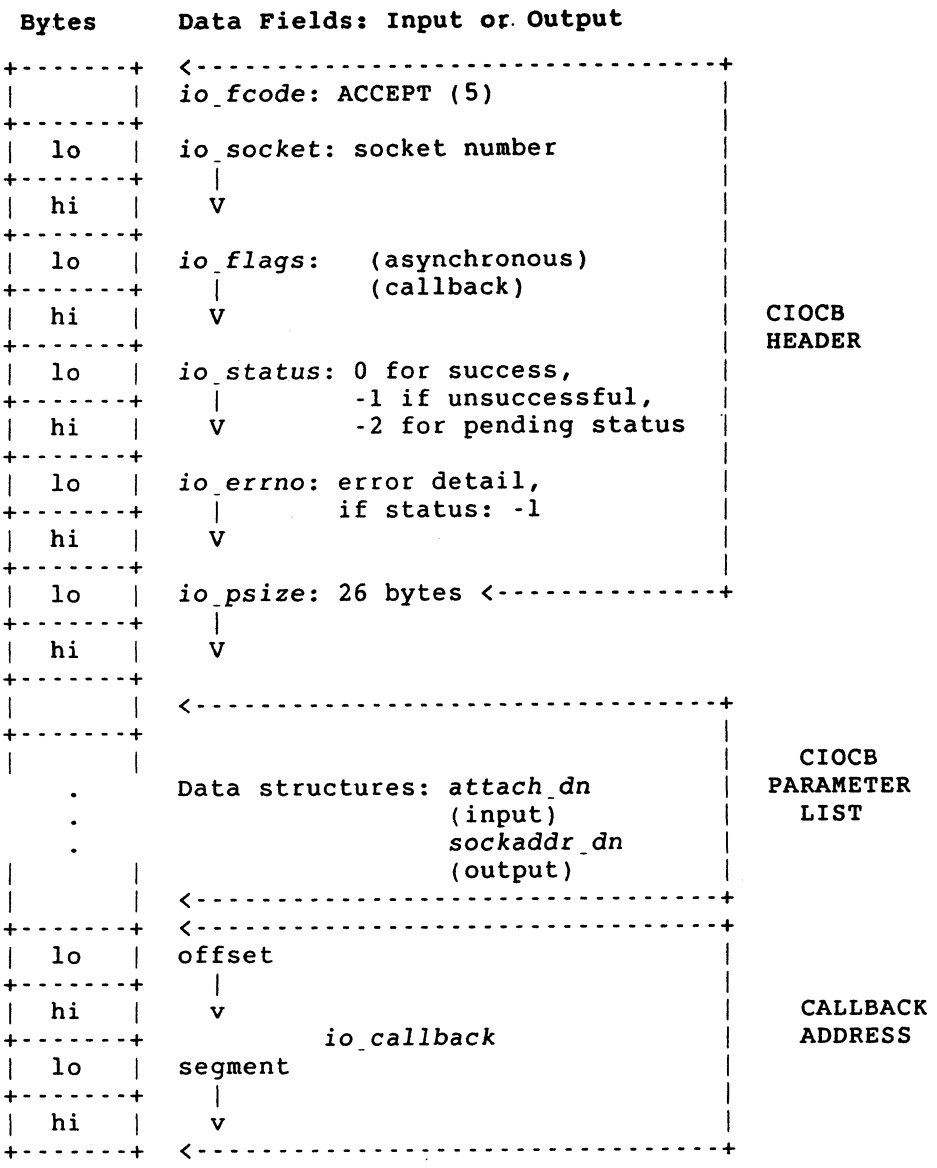
NAME

ACCEPT – accept an incoming connection request on a socket and return a socket number.

IOCB Data Members



CIOCB Data Members – For Asynchronous Mode



DESCRIPTION

The *ACCEPT* call extracts the first connection request on the queue of pending connections and creates a new socket with a new number having the same properties as the original listening socket. The original socket remains open.

If the socket is set to nonblocking I/O and there are no queued connection requests, *io_status* will return a -1, and *errno* will contain EWOULDBLOCK.

There can be two modes of accepting an incoming connection. They are immediate and deferred modes. These modes of acceptance are set using the *SETSOCKOPT* call. When immediate mode is in effect, the acceptance of the connection request takes place immediately. The deferred mode indicates that the server task completes the *ACCEPT* call without fully completing the connection to the client task. In this case, the server task can examine the access control or optional user data before it decides to accept or reject the connection request. The server task must then issue the *SETSOCKOPT* call with the appropriate reject or accept option.

DESCRIPTION - For Asynchronous Mode

The *ACCEPT* call extracts the first connection request on the queue of pending connections and creates a new socket with a new number having the same properties as the original listening socket. The original socket remains open. When the asynchronous form of the *ACCEPT* call is used, it is possible that the call may not complete. If so, there are two ways for you to check the status of the call:

- A callback routine (if implemented) will be called upon completion of the call.
- If there is no callback routine, you should poll for status. To do this, examine the *io_status* field. A value of -2, 0 or -1 may be returned. A value of -2 indicates pending (no change) status. When the *ACCEPT* call successfully completes, a value of 0 is returned in the *io_status* field. In addition, you can retrieve data from the *sockaddr_dn* data structure which is filled in by the *ACCEPT* call.

  If the call is unsuccessful, *io_status* returns a value of -1. The error reason will also be contained in *io_errno*. For a description of these error messages, see the DIAGNOSTICS section.

There can be two modes of accepting an incoming connection. They are immediate and deferred modes. These modes of acceptance are set using the *SETSOCKOPT* call. When immediate mode is in effect, the acceptance of the connection request takes place immediately. The deferred mode indicates that the server task completes the *ACCEPT* call without fully completing the connection to the client task.

In this case, the server task can examine the access control or optional user data before it decides to accept or reject the connection request. The server task can then issue the *SETSOCKOPT* call with the appropriate reject or accept option.

Input Data

io_fcode specifies *ACCEPT* as the function code. It has a decimal value of 5.

io_socket specifies the socket number to be assigned. If 0 is specified, the network process assigns a socket number for you. If a nonzero value is specified, the network process assigns that value. The assigned socket number, returned to *io_socket*, is used in subsequent send and receive calls. If the value is already in use, an error message, EBADF, is returned in *io_errno*.

io_flags defines specific flag options. You must set this data member to 0. It is not used with the synchronous form of the *ACCEPT* call.

For Asynchronous Mode: You can set *io_flags* to MSG_ASYNC and MSG_CALLBACK. MSG_ASYNC processes the asynchronous I/O form of the *ACCEPT* function call. MSG_CALLBACK allows the network to issue a callback routine when the *ACCEPT* call completes. The hexadecimal value for each flag option is listed in Appendix A.

io_psize specifies the larger size of the 2 data structures, *attach_dn* and *sockaddr_dn* (26 bytes). The 2 data structures overlay each other: *attach_dn* is used for input and *sockaddr_dn* is used for output.

attach_dn specifies the attach function data structure. The structure contains the following data fields:

att_socket defines the number for a socket which was created with the *ATTACH* call, bound to a name by the *BIND* call, and set to listen for incoming connections by the *LISTEN* call.

att_domain specifies the communications domain as AF_DECnet.

att_type specifies the socket type (for example, SOCK_STREAM). See Appendix A for a list of defined socket types.

att_protocol specifies the DECnet option for the socket (for example, DNPROTO_NSP). See Appendix A for details.

att_srp specifies the socket recovery period. This data member is not used with the *ATTACH* call.

att_supreq specifies the support requirements. This data member is not used with the *ATTACH* call.

io_callback **For Asynchronous Mode:** If the *MSG_CALLBACK* bit is set in *io_flags*, then *io_callback* specifies the 4 byte address of the function to be called by the network process when the function completes. (See Appendix B on how the data type *exptr* is formatted.)

Output Data

io_status 0 upon successful completion. If an error occurs, *io_status* returns a −1. The *io_errno* field will also contain additional error detail.

For Asynchronous Mode: If the call's status is still pending, a value of −2 is returned. The *io_status* field will return a 0 upon successful completion. If an error occurs, *io_status* will return a −1. The *io_errno* field will also contain additional error detail.

io_errno additional error detail if *io_status* returns a –1. (See the DIAGNOSTICS section for a list of error conditions.)

sockaddr_dn specifies the socket address data structure. A user retrieves data from the fields filled in by this function call. (See Appendix B on how *sockaddr_dn* is formatted.)

For Asynchronous Mode: A user cannot retrieve data from the socket address data structure until *io_status* has changed from a pending condition (–2) to successful completion (0).

The following data fields are filled in by this function call:

sdn_family is the address family AF_DECnet.

sdn_objnum is the object number for the client node. It can be a number 0 to 255.

sdn_objnamel is the size of the object name.

sdn_objname is the object name of the client network task. It can be up to a 16-byte array. It is used only when *sdn_objnum* equals 0.

sdn_add is the node address structure for the client task. (See Appendix B on how *dn_naddr* is formatted.)

Data Structure Type Summary

sdn_family	2 bytes
sdn_objnum	1 byte
sdn_objnamel	2 bytes
sdn_objname	16-byte array
sdn_add	4 bytes

att_socket	2 bytes
att_domain	2 bytes
att_type	2 bytes
att_protocol	2 bytes
att_srp	2 bytes
att_supreq	2 bytes

DIAGNOSTICS

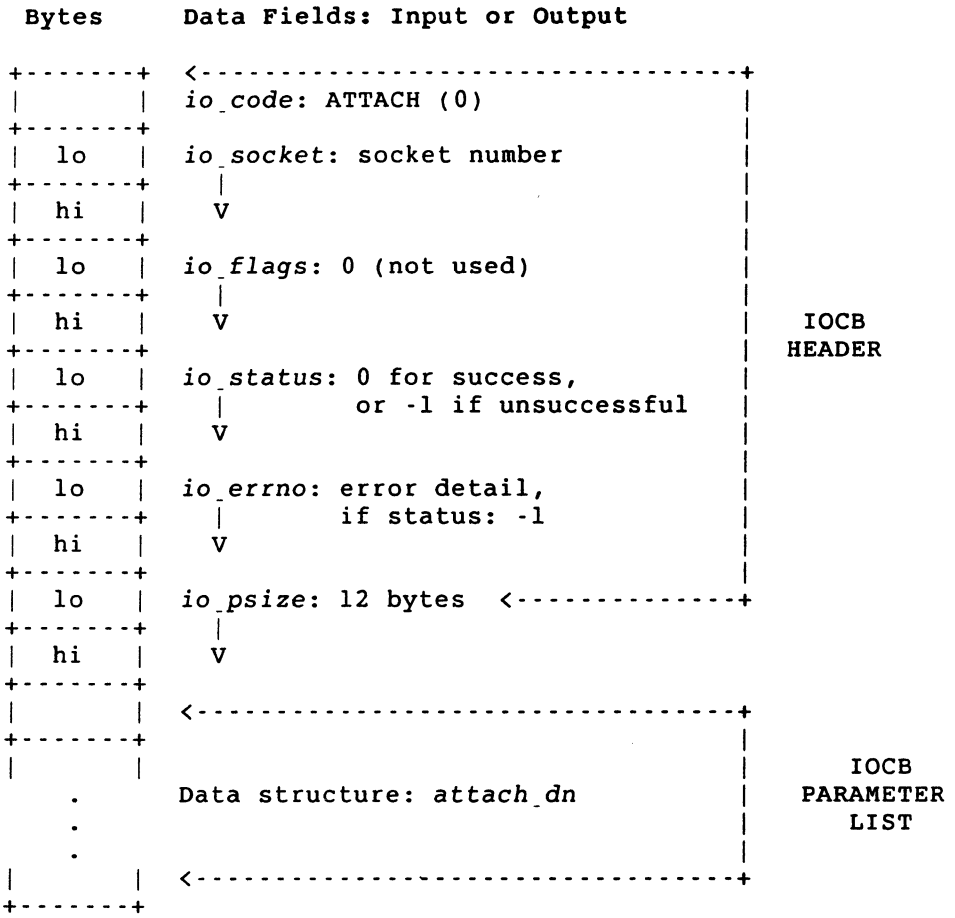
[EBADF]	The argument *io_socket* does not contain a valid socket number.
[ECONNABORTED]	The client task disconnected before the *ACCEPT* call completed.
[EEXIST]	The socket number is already in use.
[EMFILE]	There are no more available sockets.
[EWOULDBLOCK]	The socket is marked for nonblocking and no connections are waiting to be accepted. EWOULDBLOCK is not a valid error message for the asynchronous form of the *ACCEPT* call.

6.7.3 ATTACH

NAME

ATTACH – create a socket and attach a socket number.

IOCB Data Members



DESCRIPTION

The *ATTACH* call creates a socket and attaches a socket number.

Input Data

io_fcode specifies *ATTACH* as the function code. It has a decimal value of 0.

io_socket specifies the socket number to be assigned. If 0 is specified, the network process assigns a socket number for you, and returns this number to *io_socket*. If a nonzero value is specified, the network process assigns that value. The assigned socket number is used in subsequent network calls.

io_flags defines specific flag options. You must set this data member to 0. It is not used with the *ATTACH* call.

io_psize specifies the size of the data structure *attach_dn* as 12 bytes.

attach_dn specifies the attach function data structure. The structure contains the following data fields:

- *att_socket* specifies the socket number. This data field is ignored by the *ATTACH* call.
- *att_domain* specifies the communications domain as AF_DECnet.
- *att_type* specifies the socket type (for example, SOCK_STREAM). See Appendix A for a list of defined socket types.
- *att_protocol* specifies the DECnet option for the socket (for example, DNPROTO_NSP). See Appendix A for details.
- *att_srp* specifies the socket recovery period. This data member is not used with the *ATTACH* call.
- *att_supreq* specifies the support requirements. This data member is not used with the *ATTACH* call.

Output Data

io_status 0 upon successful completion. If an error occurs, *io_status* returns a −1. The *io_errno* field will also contain additional error detail.

io_errno additional error detail if *io_status* returns a −1. (See the DIAGNOSTICS section for a list of error conditions.)

Data Structure Type Summary

att_socket	2 bytes
att_domain	2 bytes
att_type	2 bytes
att_protocol	2 bytes
att_srp	2 bytes
att_supreq	2 bytes

DIAGNOSTICS

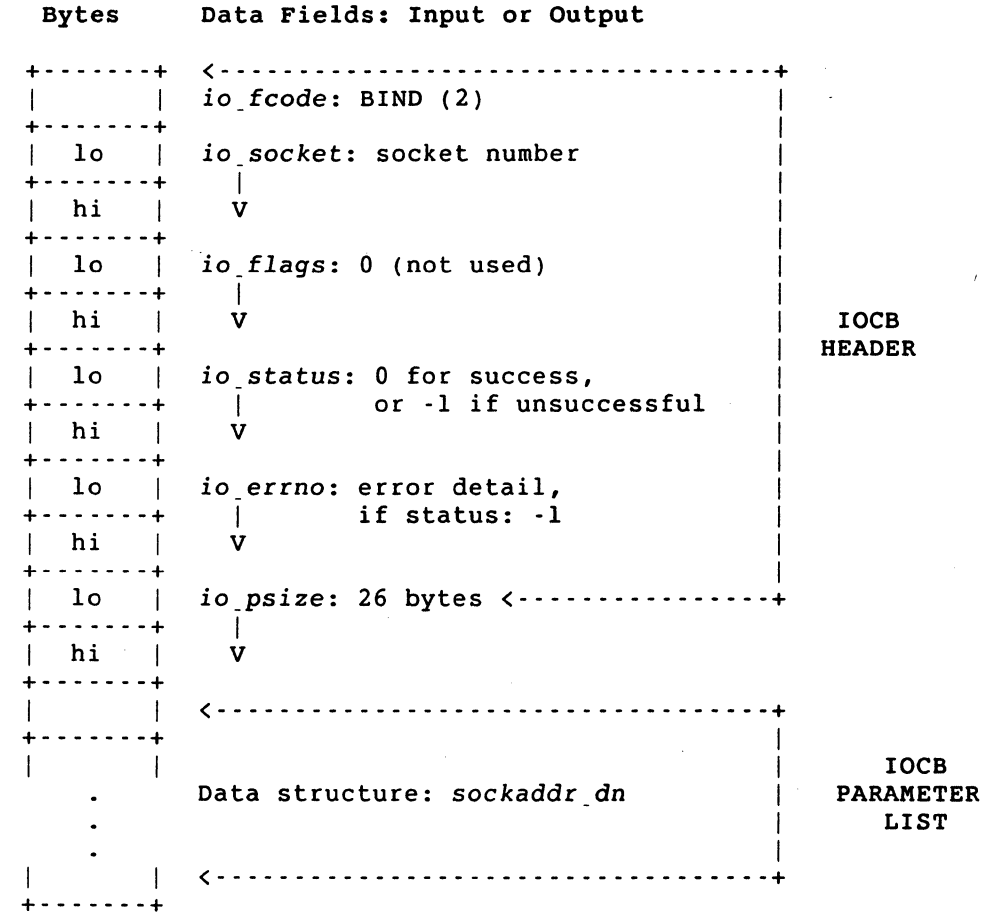
[EEXIST]	The socket number is already in use.
[EINVAL]	The argument *io_socket* does not contain a valid socket number. (You cannot assign –1 or –2 as socket numbers.)
[EMFILE]	There are no more available sockets.
[ENOBUFS]	No buffer space is available. The socket cannot be created. There are no more available logical links.

6.7.4 BIND

NAME

BIND – assign an object name or number to a socket.

IOCB Data Members



DESCRIPTION

The *BIND* call assigns an object name or number to a socket. When a socket is first created with the *ATTACH* call, it exists in a name space but has no assigned name. The *BIND* call is used primarily by server tasks. The object name is required before a server task can listen for incoming connection requests using the *LISTEN* call. It can also be used by client tasks to identify themselves to server tasks. See also *ACCEPT* (Section 6.7.2), *CONNECT* (Section 6.7.6), *PEERADDR* (Section 6.7.11) and *SOCKADDR* (Section 6.7.20).

NOTE

The VAX/VMS proxy access by user name is made possible, if the client task uses the *BIND* call specifying a user name as the object name. You should refer to the SO__REUSEADDR option of the *SETSOCKOPT* call if you wish to make more than one proxy connection with the same name.

Input Data

io__fcode specifies *BIND* as the function code. It has a decimal value of 2.

io__socket specifies the number for a socket which has been created by the *ATTACH* call.

io__flags defines specific flag options. You must set this data member to 0. It is not used with the *BIND* call.

io__psize specifies the size of the data structure *sockaddr__dn* as 26 bytes.

sockaddr__dn specifies the socket address data structure. A user fills in the data for each field. The same data members must be used with the *ACCEPT* call.

The following data fields can be modified:

- *sdn__family* specifies the address family as AF__DECnet.
- *sdn__flags* specifies the object flag option. It must be set to 0.
- *sdn__objnum* defines the object number for this network task. It can be a number from 0 to 255.
- *sdn__objnamel* is the size of the object name.
- *sdn__objname* defines the object name of this network task. It can be up to a 16-byte array. It is used only when *sdn__objnum* equals 0.
- *sdn__add* specifies the node address structure for this network task. This data member is ignored.

Output Data

io__status returns a 0 upon successful completion. If an error occurs, *io__status* returns a −1. *io__errno* will also contain additional error detail.

io__errno returns additional error detail if *io__status* returns a −1. (See the DIAGNOSTICS section for a list of error conditions.)

Data Structure Type Summary

sdn_family	2 bytes
sdn_flags	1 byte
sdn_objnum	1 byte
sdn_objnamel	2 bytes
sdn_objname	16-byte array
sdn_add	4 bytes

DIAGNOSTICS

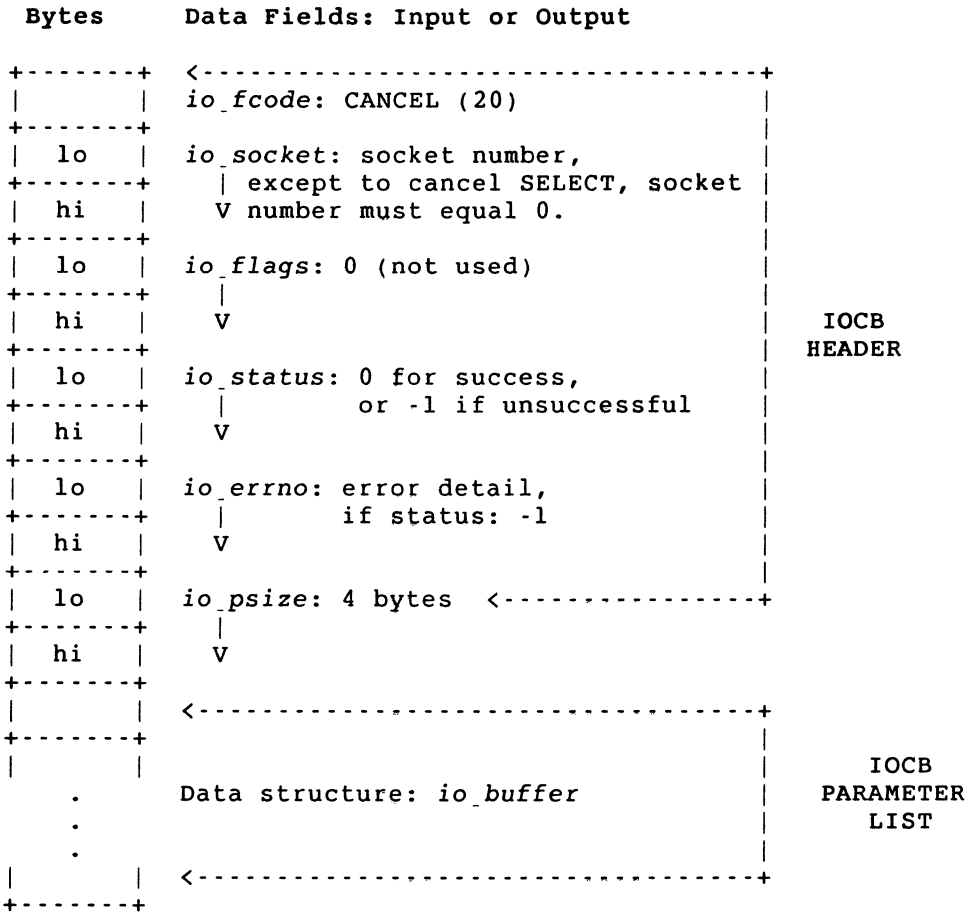
[EADDRINUSE]	The specified name is already used by another socket.
[EBADF]	The argument *io_socket* does not contain a valid socket number.
[EINVAL]	An invalid length for the object name was specified.

6.7.5 CANCEL

NAME

CANCEL – cancel a previous asynchronous function call request.

IOCB Data Members



DESCRIPTION

The *CANCEL* call allows a network process to cancel a previous asynchronous I/O request. This function call is used with the asynchronous form of the *ACCEPT*, *RCVD*, *RCVOOB*, *SELECT* and *SEND* function calls. To cancel a previous function request, you must specify the socket number for that call. If you want to cancel a *SELECT* call, the socket number must be set to 0. Otherwise, the call will not be cancelled.

There are two return values for any cancel operation:

- When a function call is cancelled, the *CANCEL* function always returns success. It does not matter whether the previous request existed, was already completed, or was still in progress.
- To determine if the previous request was found and successfully cancelled, you must examine *io_errno* for that particular call. An error code of EINTR will indicate successful cancellation. (See the DIAGNOSTICS section for each applicable function call.)

Input Data

io_code specifies *CANCEL* as the function code. It has a decimal value of 20.

io_socket specifies the socket number that must match the socket number for the call to be cancelled.

io_flags defines specific flag options. You must set this data member to 0. It is not used with the *CANCEL* call.

io_psize specifies the size of a data structure. This data member is not used with the *CANCEL* call.

Output Data

io_status returns a 0 upon successful completion.

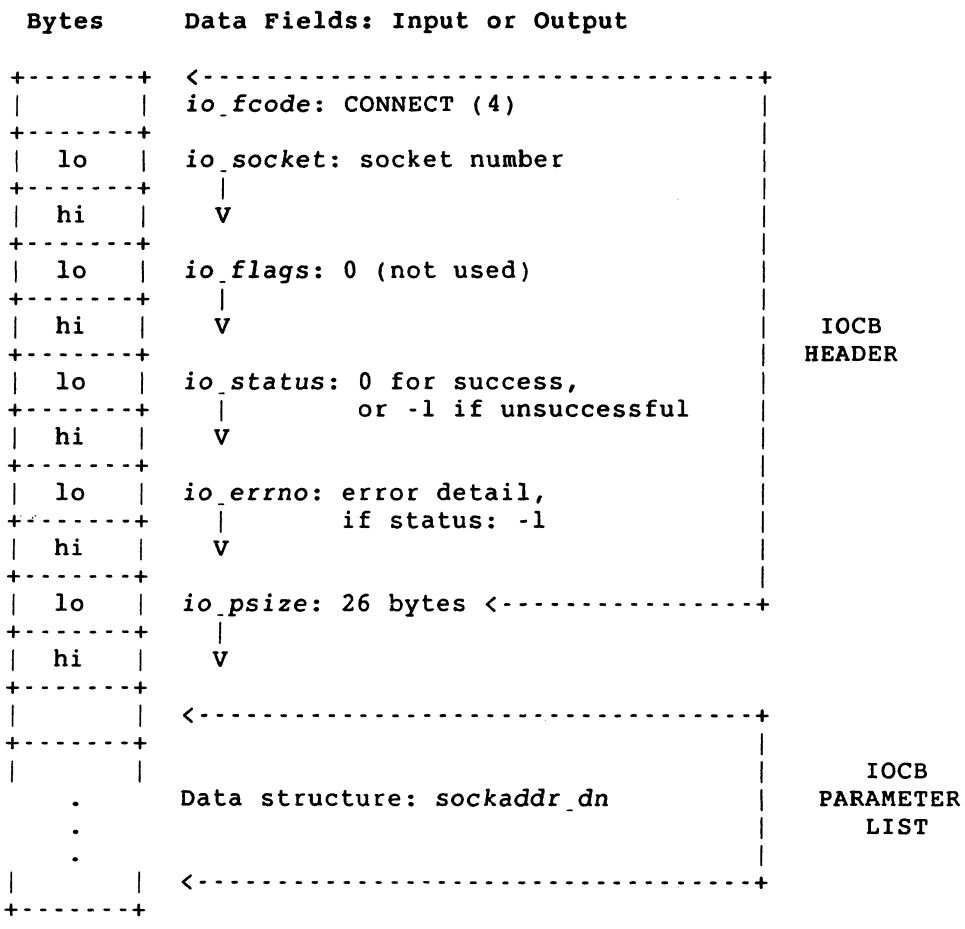
io_errno specifies additional error detail. This data member is not used with the *CANCEL* function.

6.7.6 CONNECT

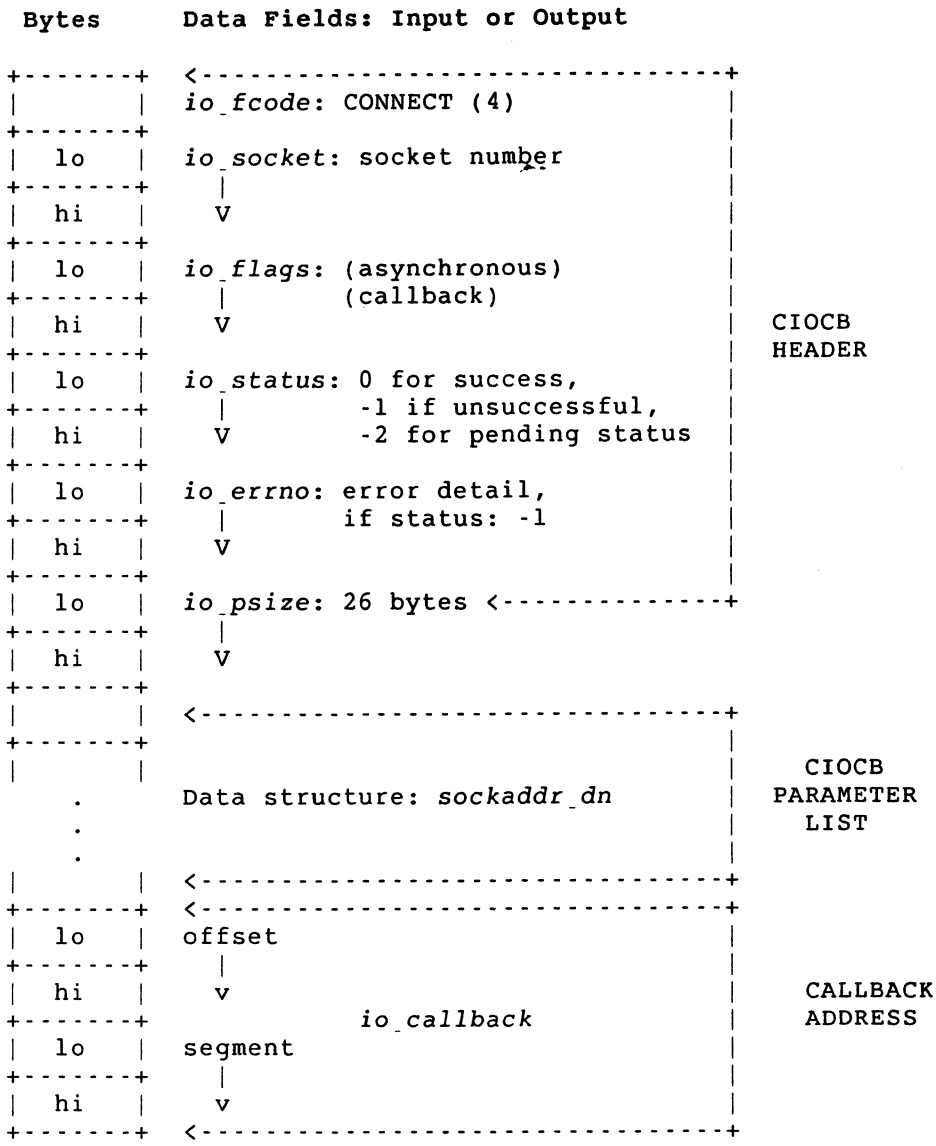
NAME

CONNECT – initiate a connection request on a socket.

IOCB Data Members



CIOCB Data Members – For Asynchronous Mode



DESCRIPTION

The *CONNECT* call issues a connection request to another socket. Optional data as well as access control information (if any) are passed to the peer task as a result of this function call. This data must be previously set by the *SETSOCKOPT* call. If subsequent *CONNECT* calls are issued on the same socket, a task must reissue the *SETSOCKOPT* call to set up new optional data and/or access control information.

NOTE

Subsequent connection requests cannot be made on the same socket until it has been disconnected.

If nonblocking I/O mode is set and the *CONNECT* call is issued, the call returns immediately with an error status, EINPROGRESS.

DESCRIPTION – For Asynchronous Mode

The *CONNECT* call issues a connection request to another socket. Optional data as well as access control information (if any) are passed to the peer task as a result of this function call. This data must be previously set by the *SETSOCKOPT* call. If subsequent *CONNECT* calls are issued on the same socket, a task must reissue the *SETSOCKOPT* call to set up new optional data and/or access control information.

When the asynchronous form of the *CONNECT* call is used, it is possible that the call may not complete. If so, there are two ways for you to check the status of the call:

- A callback routine (if implemented) will be called upon completion of the call.
- If there is no callback routine, you should poll for status. To do this, examine the *io_status* field. A value of –2, 0 or –1 may be returned. A value of –2 indicates pending (no change) status. When the *CONNECT* call successfully completes, a value of 0 is returned in the *io_status* field.

  If the call is unsuccessful, *io_status* returns a value of –1. Additional error detail will also be contained in *io_errno*. For a description of these error messages, see the DIAGNOSTICS section.

Input Data

io_fcode specifies *CONNECT* as the function code. It has a decimal value of 4.

io_socket specifies the number for the socket which has been created by the *ATTACH* call. This socket number is used for establishing a connection between the user tasks. It is also used with subsequent send and receive function calls.

io_flags defines specific flag options. You must set this data member to 0. It is not used with the synchronous form of the *CONNECT* call.

For Asynchronous Mode: You can set *io_flags* to MSG_ASYNC and MSG_CALLBACK. MSG_ASYNC processes the asynchronous I/O form of the *CONNECT* function call. MSG_CALLBACK allows the network to issue a callback routine when the *CONNECT* call completes. The hexadecimal value for each flag option is listed in Appendix A.

io_psize specifies the size of the data structure *sockaddr_dn* as 26 bytes.

sockaddr_dn specifies the socket address data structure. A user fills in the data for each field. (See Appendix B on how *sockaddr_dn* is formatted.)

The following data fields can be modified:

sdn_family specifies the address family as AF_DECnet.

sdn_flags specifies the object flag option. It must be set to 0.

sdn_objnum defines the object number for the server task. It can be a number from 0 to 255.

sdn_objnamel is the size of the object name.

sdn_objname defines the object name of the server network task. It can be up to a 16-byte array. It is only used when *sdn_objnum* equals 0.

sdn_add specifies the node address structure for the server task. (See Appendix B on how *dn_naddr* is formatted.)

io_callback **For Asynchronous Mode:** If the *MSG_CALLBACK* bit is set in *io_flags*, then *io_callback* specifies the 4 byte address of the function to be called by the network process when the function completes. (See Appendix B on how the data type *exptr* is formatted.)

Output Data

io_status returns a 0 upon successful completion. If an error occurs, *io_status* returns a −1. The *io_errno* field will also contain additional error detail.

If the socket is set to nonblocking I/O, and you issue a *CONNECT*, the function returns a value of −1 and the error message, EINPROGRESS.

For Asynchronous Mode: If the call's status is still pending, a value of −2 is returned. The *io_status* field will return a 0 upon successful completion. If an error occurs, *io_status* will return a −1. *io_errno* will also contain additional error detail.

io_errno returns additional error detail if *io_status* returns a −1. (See the DIAGNOSTICS section for a list of error conditions.)

Data Structure Type Summary

sdn_family 2 bytes

sdn_flags 1 byte

sdn_objnum 1 byte

sdn_objnamel 2 bytes

sdn_objname 16-byte array

sdn_add 4 bytes

DIAGNOSTICS

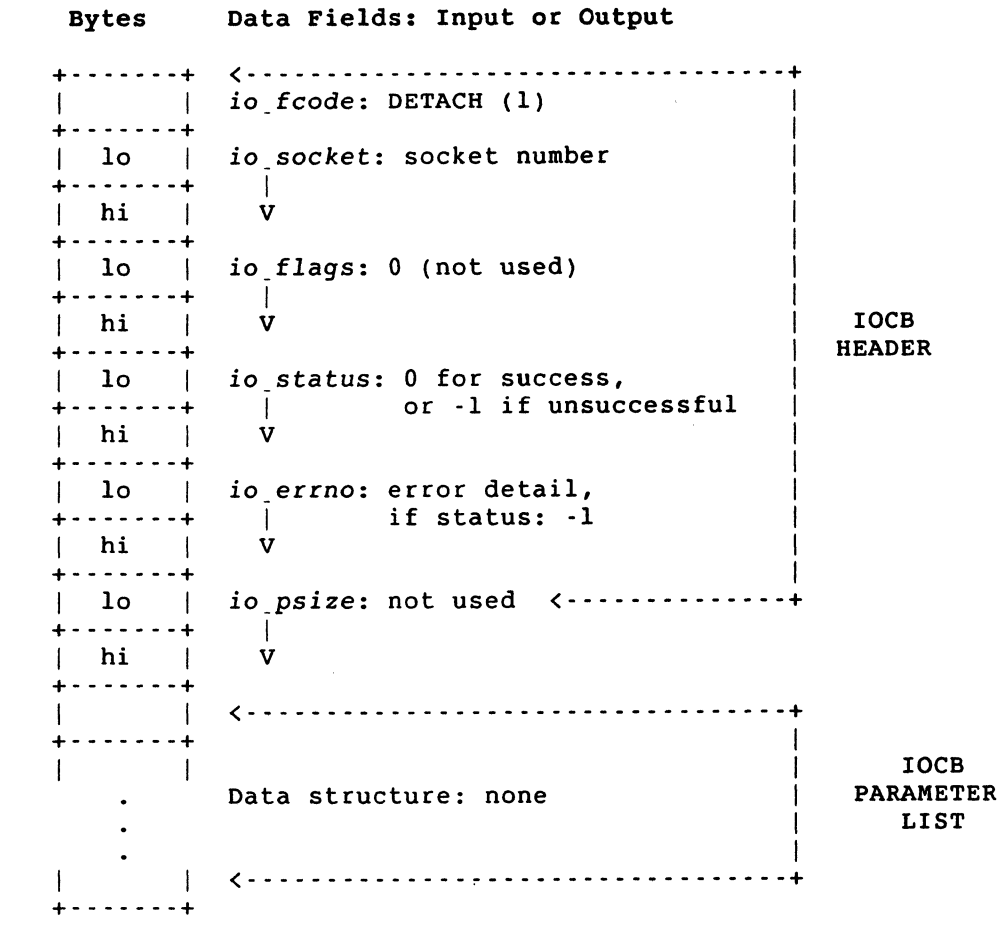
[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this particular socket.
[EBADF]	The argument *io_socket* does not contain a valid socket number.
[EBUSY]	The socket is not in idle state. The socket is in the process of being connected or disconnected; the socket is a connected or listening socket.
[ECONNABORTED]	The peer task has disconnected and the connection was aborted.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ECONNRESET]	The remote task has failed.
[EHOSTUNREACH]	The remote node is unreachable.
[EINPROGRESS]	The connection request is now in progress.
[EINVAL]	The object name of the server task is too long.
[ENETDOWN]	The network is down.
[ENETUNREACH]	The network cannot be reached from this host.
[ERANGE]	The object number of the server task is invalid. The valid range is from 0 to 255.
[ESRCH]	The server object does not exist on the remote node.
[ETIMEDOUT]	Connection establishment was timed out before a connection was established.
[ETOOMANYREFS]	The remote node has accepted the maximum number of connection requests.

6.7.7 DETACH

NAME

DETACH – disconnect all associated active logical links, and detach the specified socket and any associated sockets, only if the option SO__KEEPALIVE is not set.

IOCB Data Members



DESCRIPTION

The *DETACH* call disconnects all associated active logical links, and detaches the specified socket if it does not have the option SO__KEEPALIVE set.

NOTE

Before you can terminate a connection over a socket with the option SO_KEEPALIVE set, you must first issue a *SETSOCKOPT* call with SO_KEEPALIVE turned off. To turn off SO_KEEPALIVE, you must precede SO_KEEPALIVE with the NOT operator. (NOT SO_KEEPALIVE is the default condition.)

You are then able to issue the *DETACH* call. The logical links (if any) are disconnected, and the socket is detached. However, if you issue *DETACH* without turning off SO_KEEPALIVE, the socket remains attached, and the links (if any) stay active.

Input Data

io_fcode specifies *DETACH* as the function code. It has a decimal value of 1.

io_socket specifies the socket number created by the *ACCEPT* or *ATTACH* call.

io_flags defines specific flag options. You must set this data member to 0. It is not used with the *DETACH* call.

io_psize specifies the size of a data structure. This data member is not used with the *DETACH* call.

Output Data

io_status returns a 0 upon successful completion. If an error occurs, *io_status* returns a −1. The *io_errno* field will also contain additional error detail.

io_errno returns additional error detail if *io_status* returns a −1. (See the DIAGNOSTICS section for a possible error condition.)

DIAGNOSTICS

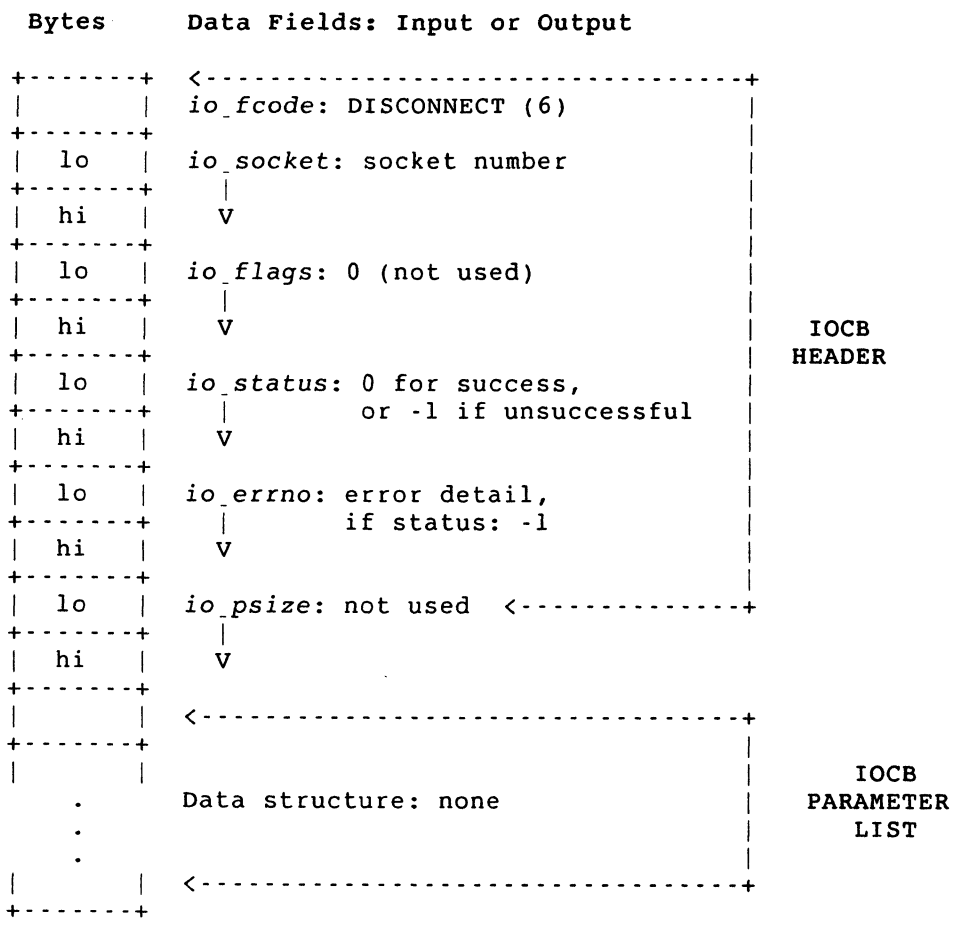
[EBADF] The argument *io_socket* does not contain a valid socket number.

6.7.8 DISCONNECT

NAME

DISCONNECT – disconnect socket from the peer socket, and terminate the logical link connection.

IOCB Data Members



DESCRIPTION

The *DISCONNECT* call disconnects the socket from the peer socket, and terminates the logical link.

NOTE

Before you can terminate a connection over a socket set with the option SO__KEEPALIVE, you must first issue a *SETSOCKOPT* call with the SO__KEEPALIVE option turned off. That is, precede the SO__KEEPALIVE with the NOT operator. Then issue the *DISCONNECT* function call, and the connection will then be completely broken.

The effect of *DISCONNECT* on unsent data queued for a remote task depends on the linger option set with the *SETSOCKOPT* function call. (See Section 6.7.17.) If SO__LINGER is set, control is returned to the task, but the link is not disconnected until the unqueued data is sent. If SO__DONTLINGER is set, control is returned to the task, and any unqueued data is lost.

Input Data

io__fcode specifies *DISCONNECT* as the function code. It has a decimal value of 6.

io__socket specifies the socket number created by the *ACCEPT* or *ATTACH* call.

io__flags defines specific flag options. You must set this data member to 0. It is not used with the *DISCONNECT* call.

io__psize specifies the size of a data structure. This data member is not used with the *DISCONNECT* call.

Output Data

io__status returns a 0 upon successful completion. If an error occurs, *io__status* returns a −1. The *io__errno* field will also contain additional error detail.

io__errno returns additional error detail if *io__status* returns a −1. (See the DIAGNOSTICS section for a possible error condition.)

DIAGNOSTICS

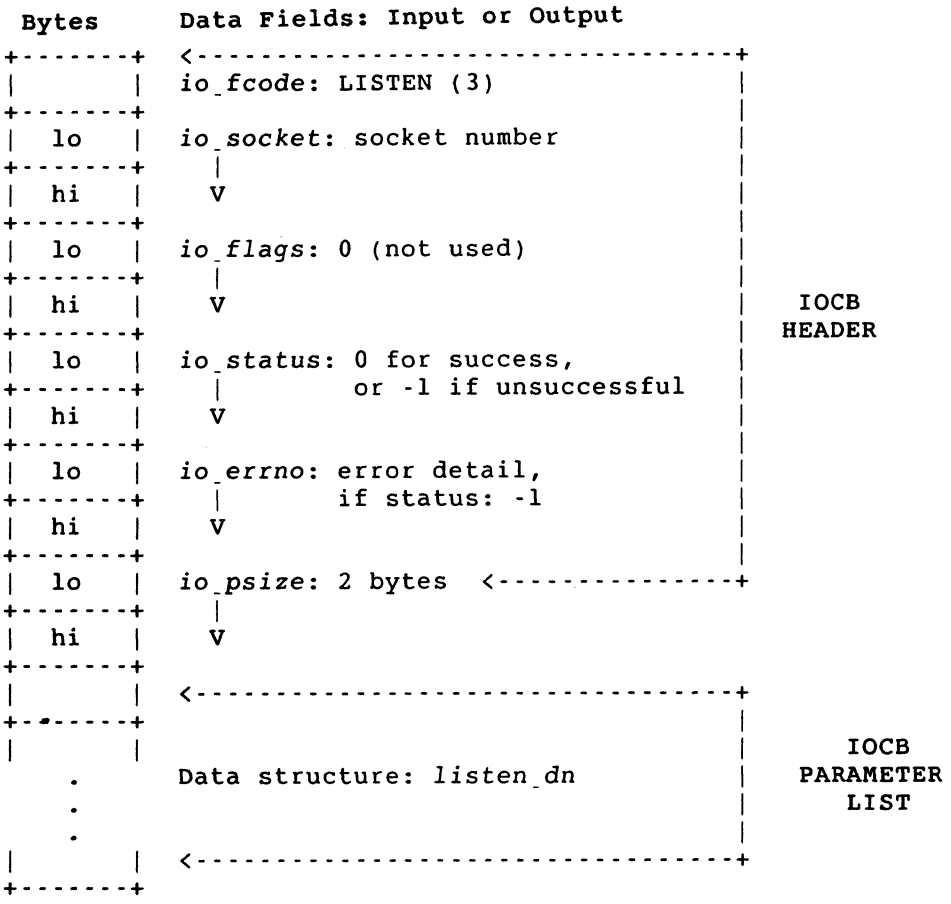
[EBADF] The argument *io__socket* does not contain a valid socket number.

6.7.9 LISTEN

NAME

LISTEN – listen for pending connections on a socket.

IOCB Data Members



DESCRIPTION

The *LISTEN* call declares your socket as a server which is available for client connections. The server must use a bound name or number in order to listen for incoming client connections. This call must be issued before an incoming connection can be accepted or rejected. See also the *ACCEPT* (Section 6.7.2) and the *SELECT* (Section 6.7.14) calls.

If you detach a listening socket while the socket is receiving client connections, then all links associated with the listening socket immediately abort and all outstanding data is lost.

Input Data

io_fcode specifies *LISTEN* as the function code. It has a decimal value of 3.

io_socket specifies a number for a socket which has been created by the *ATTACH* call and bound to a name by the *BIND* call.

io_flags defines specific flag options. You must set this data member to 0. It is not used with the *LISTEN* call.

io_psize specifies the size of the data structure *listen_dn* as 2 bytes.

listen_dn specifies the listen data structure. (See Appendix B on how *listen_dn* is formatted.) The structure contains the following data field:

lsn_backlog defines the total maximum number of unaccepted incoming connects which are allowed on this particular socket. The maximum allowable number of incoming connects is 5. If a connection request arrives when the queue is full, the client task will receive an error with an indication of ECONNREFUSED.

Output Data

io_status returns a 0 upon successful completion. If an error occurs, *io_status* returns a −1. The *io_errno* field will also contain additional error detail.

io_errno returns additional error detail if *io_status* returns a −1. (See the DIAGNOSTICS section for a list of error conditions.)

Data Structure Type Summary

lsn_backlog 2 bytes

DIAGNOSTICS

[EBADF] The argument *io_socket* does not contain a valid socket number.

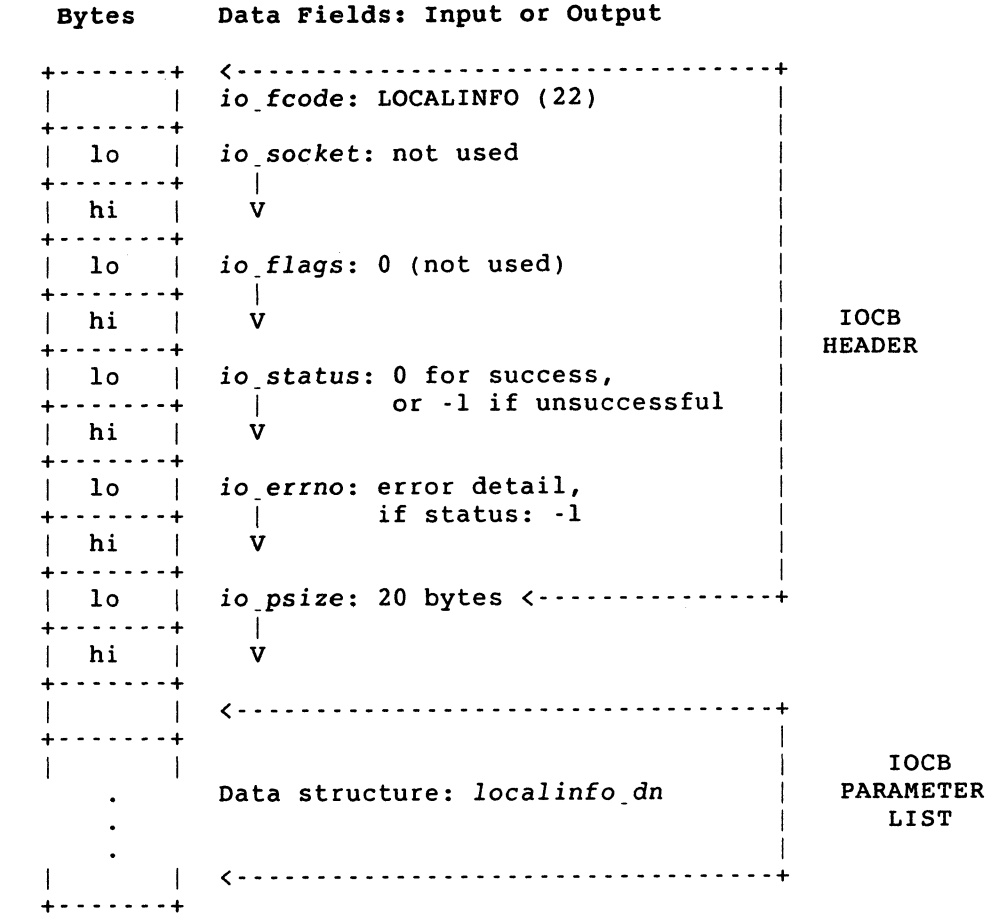
[EOPNOTSUPP] The specified socket type does not support the listen operation.

6.7.10 LOCALINFO

NAME

LOCALINFO – retrieve network information for the local node.

IOCB Data Members



DESCRIPTION

The *LOCALINFO* call retrieves network information for the local node. It returns the software version number for the network process, the local node name and address, and the maximum possible segment buffer size which can be used on a logical link. It also returns the number of sockets available for data exchange and the current DECnet database device and path.

Input Data

io_fcode specifies *LOCALINFO* as the function code. It has a decimal value of 22.

io_socket specifies the socket number. It is not used with the *LOCALINFO* call.

io_flags defines specific flag options. You must set this data member to 0. It is not used with the *LOCALINFO* call.

io_psize specifies the size of the data structure *localinfo_dn* as 20 bytes.

Output Data

io_status returns a 0 upon successful completion. If an error occurs, *io_status* returns a −1. The *io_errno* field will also contain additional error detail.

io_errno returns additional error detail if *io_status* returns a −1. (See Appendix C for a list of error conditions.)

localinfo_dn specifies the local node information data structure. A user retrieves data from the fields filled in by this function call. (See Appendix B on how *localinfo_dn* is formatted.)

The following data fields can be filled in by this function call:

lcl_version is the software version number for the network process.

lcl_nodename is the node name for the local node.

lcl_nodeaddr is the node address for the local node.

lcl_segsize is the buffer segment size to be used on the logical link.

lcl_sockets is the number of sockets available for data exchange.

lcl_decnet_device is the DECnet database device.

lcl_decnet_path specifies the address of the buffer that contains the DECnet database path specification which includes the device name.

Data Structure Type Summary

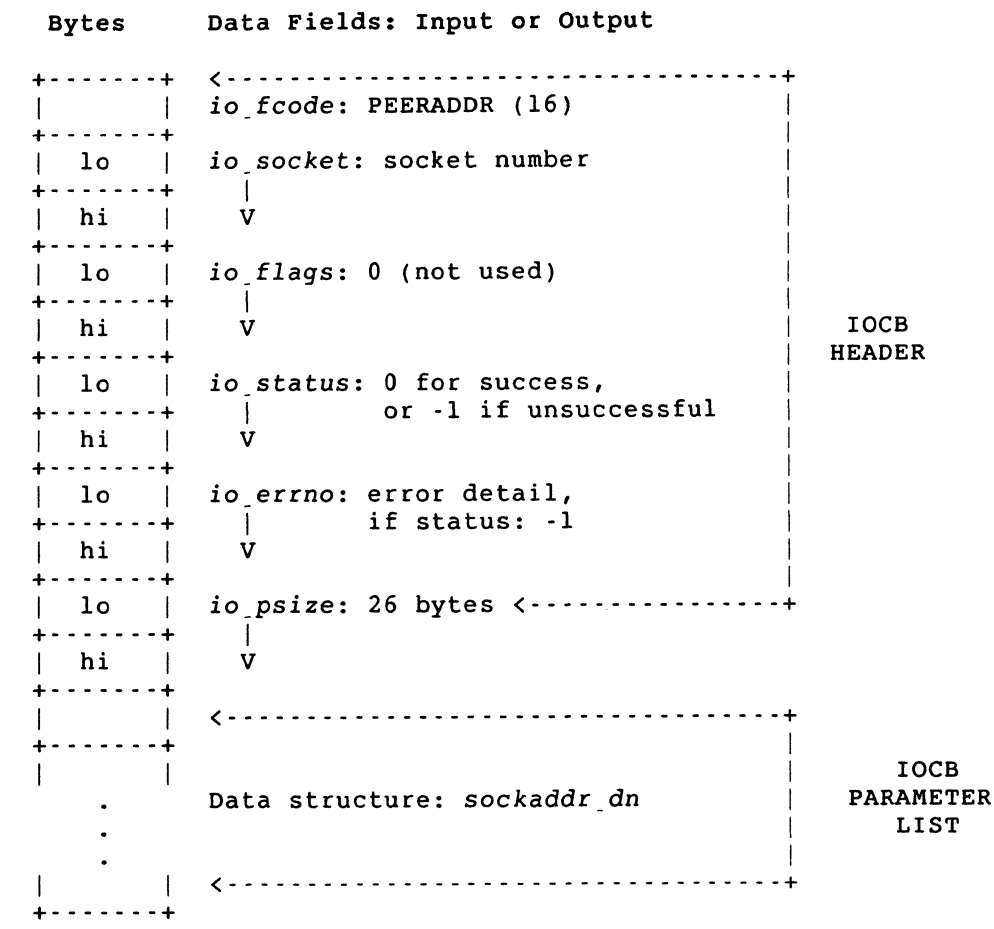
lcl_version	3-byte array
lcl_nodename	7-byte array
lcl_nodeaddr	2 bytes
lcl_segsize	2 bytes
lcl_sockets	1 byte
lcl_decnet_device	1 byte
lcl_decnet_path	4 bytes

6.7.11 PEERADDR

NAME

PEERADDR – get information about your peer socket.

IOCB Data Members



DESCRIPTION

The *PEERADDR* call returns information about your peer socket.

Input Data

io_fcode specifies *PEERADDR* as the function code. It has a decimal value of 16.

io_socket specifies the number for the socket which has been created by the *ACCEPT* or *ATTACH* call.

io_flags defines specific flag options. You must set this data member to 0. It is not used with the *PEERADDR* call.

io_psize specifies the size of the data structure *sockaddr_dn* as 26 bytes.

Output Data

io_status returns a 0 upon successful completion. If an error occurs, *io_status* returns a −1. The *io_errno* field will also contain additional error detail.

io_errno returns additional error detail if *io_status* returns a −1. (See the DIAGNOSTICS section for a possible error condition.)

sockaddr_dn specifies the socket address data structure. A user retrieves data from the fields filled in by this function call. (See Appendix B on how *sockaddr_dn* is formatted.)

The following data fields can be filled in by this function call:

sdn_family is the address family AF_DECnet.

sdn_objnum is the object number for the peer task. It can be a number from 0 to 255.

sdn_objnamel is the size of the object name for the peer task.

sdn_objname is the name of the peer network task. It can be up to a 16-element array. It is only used when *sdn_objnum* equals 0.

sdn_add is the node address structure for the peer task. (See Appendix B on how *dn_naddr* is formatted.)

Data Structure Type Summary

sdn_family	2 bytes
sdn_objnum	1 byte
sdn_objnamel	2 bytes
sdn_objname	16-byte array
sdn_add	4 bytes

DIAGNOSTICS

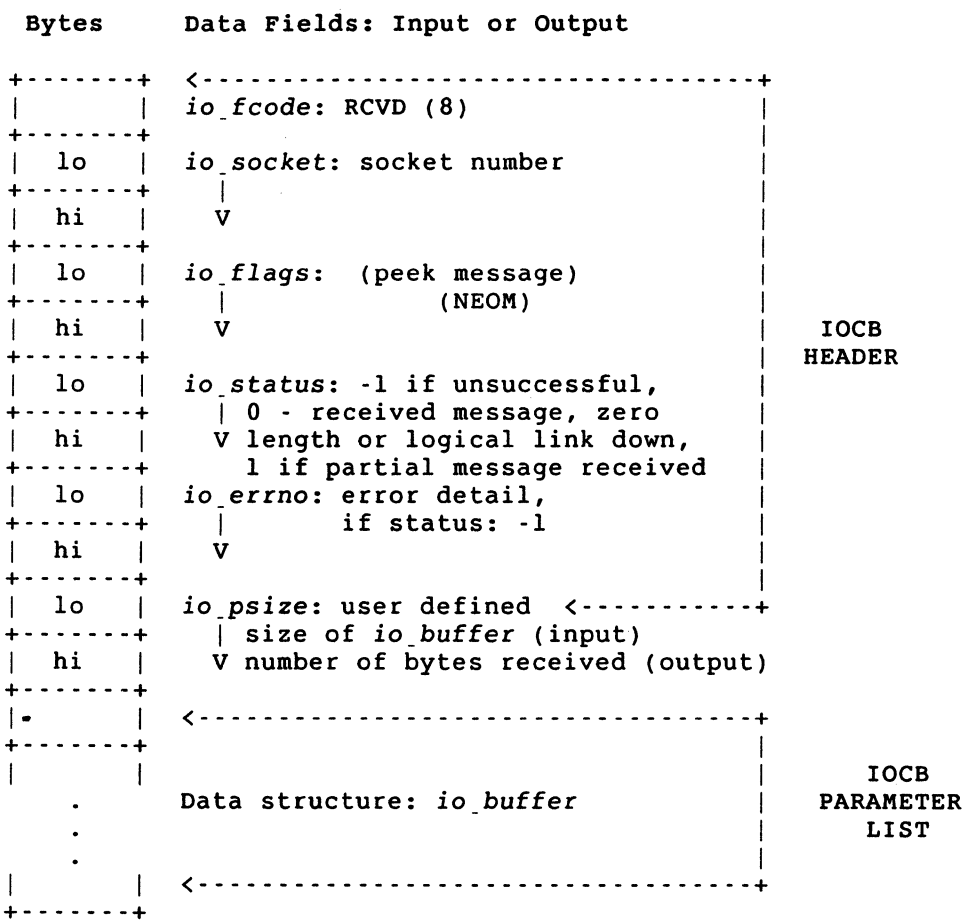
[EBADF] The argument *io_socket* does not contain a valid socket number.

6.7.12 RCVD

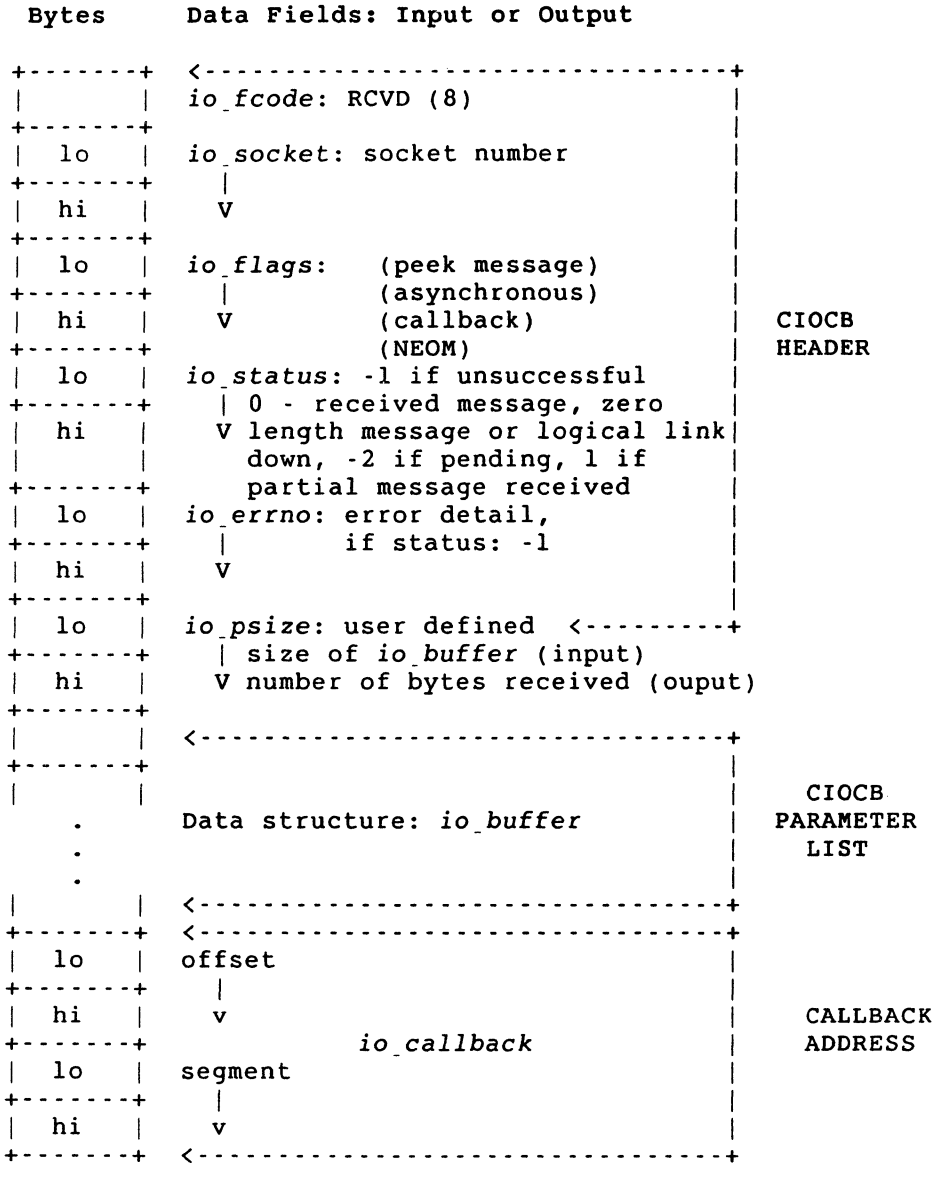
NAME

RCVD – receive data on a specified socket.

IOCB Data Members



CIOCB Data Members – For Asynchronous Mode



DESCRIPTION

The *RCVD* call is used to receive data from your peer. If no messages are available at the socket, the *RCVD* call waits for a message to arrive unless the socket is nonblocking. In this case, a status of –1 is returned with the field *io__errno* set to EWOULDBLOCK.

If the socket becomes disconnected, queued data can still be received from the broken logical link. However, if you shut down the socket or detach it, queued data cannot be received. When the logical link is not in a connected state, and all data has been read, the *RCVD* call returns zero bytes.

For sequenced sockets, you can read a single message with multiple calls into multiple buffers. (See Appendix A for a description of socket types.) To do this, set the *io_flags* field to MSG_NEOM (not end of message).

NOTE

MSG_NEOM is not a valid option for stream sockets. Stream mode destroys all record boundaries.

For example, you want to receive a 300 byte message but you only specified a receive buffer of 100 bytes. Normally, the *RCVD* call would flush the rest of the message (after you read the first 100 bytes) and return a status of 0. Setting *io_flags* to MSG_NEOM indicates that the caller does not want the remaining unread data to be flushed. If the user did not receive the entire message, *io_status* returns a value of 1. MSG_NEOM allows you to issue another *RCVD* call and read the remaining 200 bytes of the buffer.

In addition to flagging the *RCVD* call with MSG_NEOM, you can also set the flags option to MSG_PEEK. This option enables you to "peek" or read the next pending message without removing it from the receive queue. When the *RCVD* call is flagged with MSG_PEEK and MSG_NEOM, multiple *RCVD* calls can be made to peek at the entire message. You cannot peek at more than one message.

The *SELECT* call may be used to determine when more data has arrived. (See Section 6.7.14.)

DESCRIPTION – For Asynchronous Mode

The *RCVD* call is used to receive data from your peer. If the socket becomes disconnected, queued data can still be received from the broken logical link. However, if you shut down the socket or detach it, queued data cannot be received. When the logical link is not in a connected state, and all data has been read, the *RCVD* call returns zero bytes.

When the asynchronous form of the *RCVD* call is used, it is possible that the call may not complete. If so, there are two ways for you to check the status of the call:

- If the asynchronous form of the *RCVD* call is used, you can also specify that a callback routine be used. The message option, MSG_CALLBACK, allows the network to issue a callback routine when the *RCVD* call completes.
- If there is no callback routine, you should poll for status. To do this, examine the *io_status* field. A value of –2, 0, –1 or 1 may be returned. A value of –2 indicates pending (no change) status. If a value of 0 is returned, you need to see how the call was completed: If the message was received, the number of bytes sent by your peer is returned in *io_psize*. Otherwise, a zero length message was received or the logical link has been disconnected.

A value of 1 indicates that a partial message was received.

If the call is unsuccessful, *io_status* returns a value of –1. The error reason will be contained in *io_errno*. For a description of error messages, see the DIAGNOSTICS section.

For sequenced sockets, you can read a single message with multiple calls into multiple buffers. To do this, set the *io_flags* field to MSG_NEOM (not end of message).

NOTE

MSG_NEOM is not a valid option for stream sockets. Stream mode destroys all record boundaries.

For example, you want to receive a 300 byte message, but you only specified a receive buffer of 100 bytes. Normally, the *RCVD* call would flush the rest of the message (after you read the first 100 bytes) and return a status of 0. Setting *io_flags* to MSG_NEOM indicates that the caller does not want the remaining unread data to be flushed. If the user did not receive the entire message, *io_status* returns a value of 1. MSG_NEOM allows you to issue another *RCVD* call and read the remaining 200 bytes of the buffer.

In addition to flagging the *RCVD* call with MSG_NEOM, you can also set the flags option to MSG_PEEK. This option enables you to "peek" or read the next pending message without removing it from the receive queue. When the *RCVD* call is flagged with MSG_PEEK and MSG_NEOM, multiple *RCVD* calls can be made to peek at the entire message. You cannot peek at more than one message.

The *SELECT* call may be used to determine when more data has arrived. (See Section 6.7.14.)

Input Data

io_fcode specifies *RCVD* as the function code. It has a decimal value of 8.

io_socket specifies the number for a socket created by the *ACCEPT* or *ATTACH* call.

io_flags defines specific flag options. You can set this field to 0 for reading normal messages. To read the next pending message without removing it from the receive queue, set the *io_flags* field to MSG_PEEK. To receive a single message having multiple parts, set the *io_flags* field to MSG_NEOM. The hexadecimal value for each flag option is listed in Appendix A.

For Asynchronous Mode: You can set this data member to MSG_ASYNC and MSG_CALLBACK for implementing asynchronous callback routines. MSG_ASYNC processes the asynchronous I/O form of the *RCVD* function call. MSG_CALLBACK allows the network to issue a callback routine when the *RCVD* call completes. In addition to asynchronous callbacks, you can set the *io_flags* to MSG_PEEK and/or MSG_NEOM. MSG_PEEK allows you to read

the next pending message without removing it from the receive queue. MSG_NEOM allows you to receive a single message in multiple parts. The hexadecimal value for each flag option is listed in Appendix A.

io_psize specifies the size of the user defined buffer.

io_buffer specifies the address for the buffer which contains the incoming message. (Refer to Appendix B on how *io_buffer* is formatted.)

io_callback **For Asynchronous Mode:** If the *MSG_CALLBACK* bit is set in *io_flags*, then *io_callback* specifies the 4 byte address of the function to be called by the network process when the function completes. (See Appendix B on how the data type *exptr* is formatted.)

Output Data

io_buffer is the address for the buffer which will contain the incoming message. (Refer to Appendix B on how *io_buffer* is formatted.)

io_psize specifies the number of transferred bytes upon successful completion.

io_status If status returns a 0, the message was received. (Check *io_psize* for the number of transferred bytes.) Otherwise, you have received a zero length message, or the logical link has been disconnected. To determine the state of the logical link, use the *GETSOCKOPT* function call with the *DSO_LINKINFO* option (See Section 6.7.17). If the link has been disconnected, then all subsequent receives will return zero bytes.

If an error occurs, *io_status* returns a −1. The *io_errno* field will also contain additional error detail.

For Asynchronous Mode: If the call's status is still pending, a value of −2 is returned. If status returns a 0, the message was received. (Check *io_psize* for the number of transferred bytes.) Otherwise, you have received a zero length message, or the logical link has been disconnected. To determine the state of the logical link, use the *GETSOCKOPT* function call with the *DSO_LINKINFO* option (See Section 6.7.17). If the link has been disconnected, then all subsequent receives will return zero bytes.

If an error occurs, *io_status* returns a −1. The *io_errno* field will also contain additional error detail.

io_errno returns additional error detail if *io_status* returns a −1. (See the DIAGNOSTICS section for a list of error conditions.)

If you receive a zero length message, *io_errno* will return a 1.

Data Structure Type Summary

io_buffer 4 bytes

io_callback 4 bytes

DIAGNOSTICS

When receiving normal data, the following set of error messages can occur:

Blocking I/O

Message	Description
[EBADF]	The argument *io_socket* does not contain a valid socket number.

Nonblocking I/O

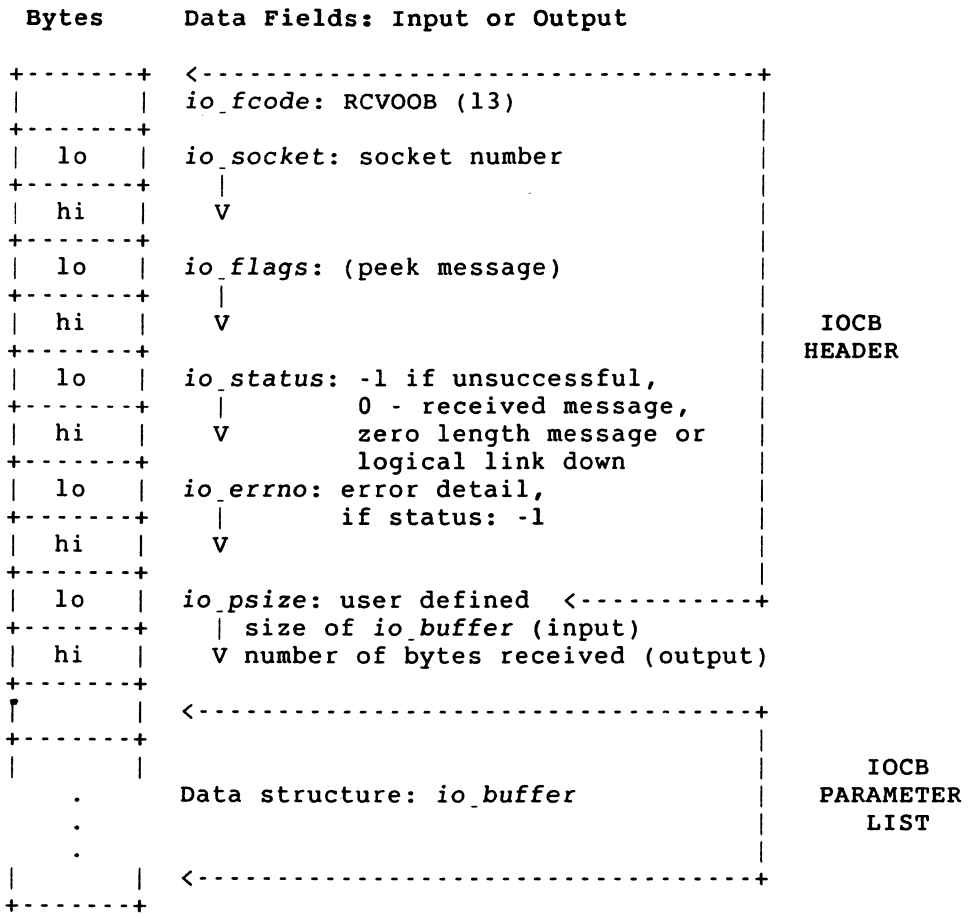
[EBADF]	The argument *io_socket* does not contain a valid socket number.
[EWOULDBLOCK]	The receive operation would block because there is currently no data to receive. EWOULDBLOCK is not a valid error message for the asynchronous form of the *RCVD* call.

6.7.13 RCVOOB

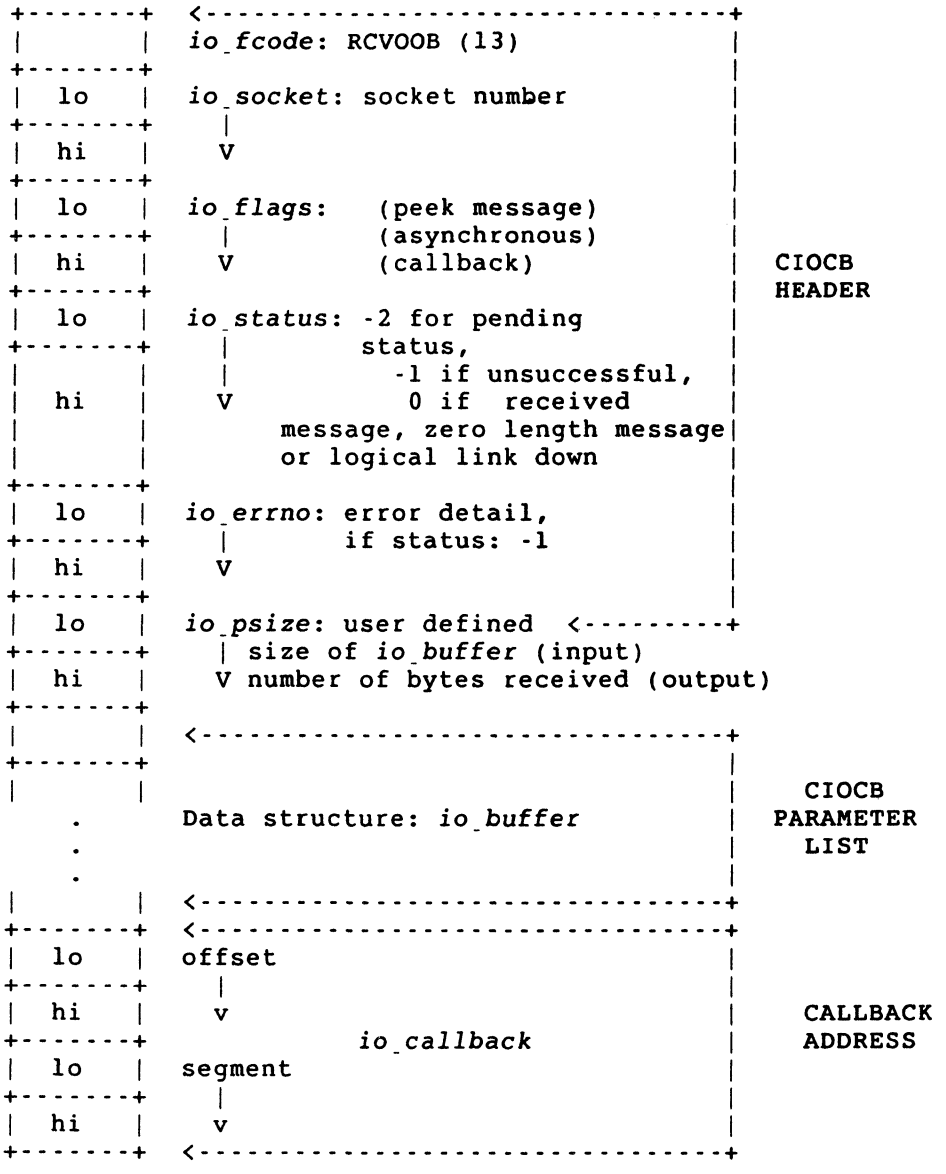
NAME

RCVOOB – receive out-of-band messages on a specified socket.

IOCB Data Members



CIOCB Data Members – For Asynchronous Mode



DESCRIPTION

The *RCVOOB* call is used to receive out-of-band data from another socket. Out-of-band messages are delivered to a receiving task ahead of normal messages. If the socket is set to nonblocking I/O, and there is no data to receive, a status of –1 is returned with the field *io_errno* set to EWOULDBLOCK.

NOTE

This occurs whether or not the socket is in blocking or nonblocking mode.

If the socket becomes disconnected, queued data can still be received from the broken logical link. However, if you shut down the socket or detach it, queued data cannot be received. When the logical link is not in a connected state, and all data has been read, the *RCVOOB* call will not return.

The *SELECT* call may be used to determine when more data has arrived. (See Section 6.7.14.)

DESCRIPTION – For Asynchronous Mode

The *RCVOOB* call is used to receive out-of-band data from another socket. Out-of-band messages are delivered to a receiving task ahead of normal messages.

When the asynchronous form of the *RCVOOB* call is used, it is possible that the call may not complete. If so, there are two ways for you to check the status of the call:

- If the asynchronous form of the *RCVOOB* call is used, you can also specify that a callback routine be used. The message option, MSG_CALLBACK, allows the network to issue a callback routine when the *RCVOOB* call completes.
- If there is no callback routine, you can poll for status. To do this, examine the *io_status* field. A value of –2, 0, or –1 may be returned. A value of –2 indicates pending (no change) status. If status returns a 0, the message was received. (Check *io_psize* for the number of transferred bytes.) Otherwise, you have received a zero length message, or the logical link has been disconnected.

  If the call is unsuccessful, *io_status* returns a value of –1. Additional error detail will also be contained in *io_errno*. For a description of these error messages, see the DIAGNOSTICS section.

If the socket becomes disconnected, queued data can still be received from the broken logical link. However, if you shut down the socket or detach it, queued data cannot be received. When the logical link is not in a connected state, and all data has been read, the *RCVOOB* call will return but the function will not complete. The *SELECT* call may be used to determine when more data has arrived. (See Section 6.7.14.)

Input Data

io_fcode specifies *RCVOOB* as the function code. It has a decimal value of 13.

io_socket specifies the number for a socket which has been created by the *ACCEPT* or *CONNECT* call.

io_flags defines specific flag options. You can set this field to MSG_PEEK to read the next pending message without removing it from the receive queue. The hexadecimal value for each flag option is listed in Appendix A.

For Asynchronous Mode: You can set this data member to MSG_ASYNC and MSG_CALLBACK for implementing asynchronous callback routines. MSG_ASYNC processes the asynchronous I/O form of the *RCVOOB* function call. MSG_CALLBACK allows the network to issue a callback routine when the *RCVOOB* call completes. In addition to asynchronous callbacks, you can set the *io_flags* to MSG_PEEK which allows you to read the next pending message without removing it from the receive queue. The hexadecimal value for each flag option is listed in Appendix A.

io_psize specifies the size of the user defined buffer.

io_buffer specifies the address for the buffer which contains the incoming out-of-band message. (Refer to Appendix B on how *io_buffer* is formatted.)

io_callback **For Asynchronous Mode:** If the *MSG_CALLBACK* bit is set in *io_flags*, then *io_callback* specifies the 4 byte address of the function to be called by the network process when the function completes. (See Appendix B on how the data type *exptr* is formatted.)

Output Data

io_buffer is the address for the buffer which will contain the incoming out-of-band message. (Refer to Appendix B on how *io_buffer* is formatted.)

io_psize specifies the number of transferred bytes upon successful completion.

io_status If status returns a 0, the message was received. (Check *io_psize* for the number of transferred bytes.) Otherwise, you have received a zero length message, or the logical link has been disconnected. To determine the state of the logical link, use the *GETSOCKOPT* function call with the *DSO_LINKINFO* option (See Section 6.7.17). If the link has been disconnected, then all subsequent receives will return zero bytes.

If an error occurs, *io__status* returns a −1. The *io__errno* field will also contain additional error detail.

For Asynchronous Mode: If the call's status is still pending, a value of −2 is returned. If status returns a 0, the message was received. (Check *io__psize* for the number of transferred bytes.) Otherwise, you have received a zero length message, or the logical link has been disconnected. To determine the state of the logical link, use the *GETSOCKOPT* function call with the *DSO__LINKINFO* option (See Section 6.7.17). If the link has been disconnected, then all subsequent receives will return zero bytes.

If an error occurs, *io__status* returns a −1. The *io__errno* field will also contain additional error detail.

io__errno returns additional error detail if *io__status* returns a −1. (See the DIAGNOSTICS section for a list of error conditions.)

If you receive a zero length message, *io__errno* will return a 1.

Data Structure Type Summary

io__buffer 4 bytes

io__callback 4 bytes

DIAGNOSTICS

When receiving out-of-band data, the following set of error messages can occur:

Blocking I/O

Message	Description
[EBADF]	The argument *io__socket* does not contain a valid socket number.
[EWOULDBLOCK]	The receive operation would block because there is currently no data to receive. EWOULDBLOCK is not a valid error message for the asynchronous form of the *RCVOOB* call.

Nonblocking I/O

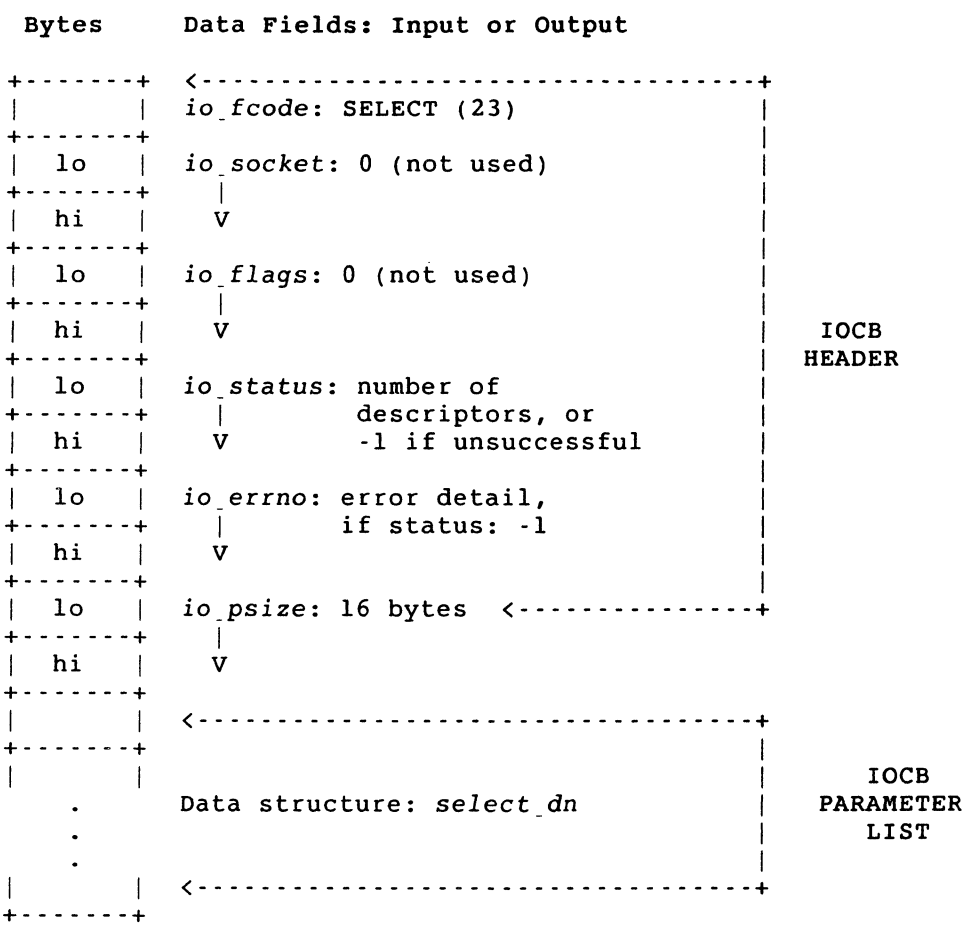
Message	Description
[EBADF]	The argument *io__socket* does not contain a valid socket number.
[EWOULDBLOCK]	The receive operation would block because there is currently no data to receive. EWOULDBLOCK is not a valid error message for the asynchronous form of the *RCVOOB* call.

6.7.14 SELECT

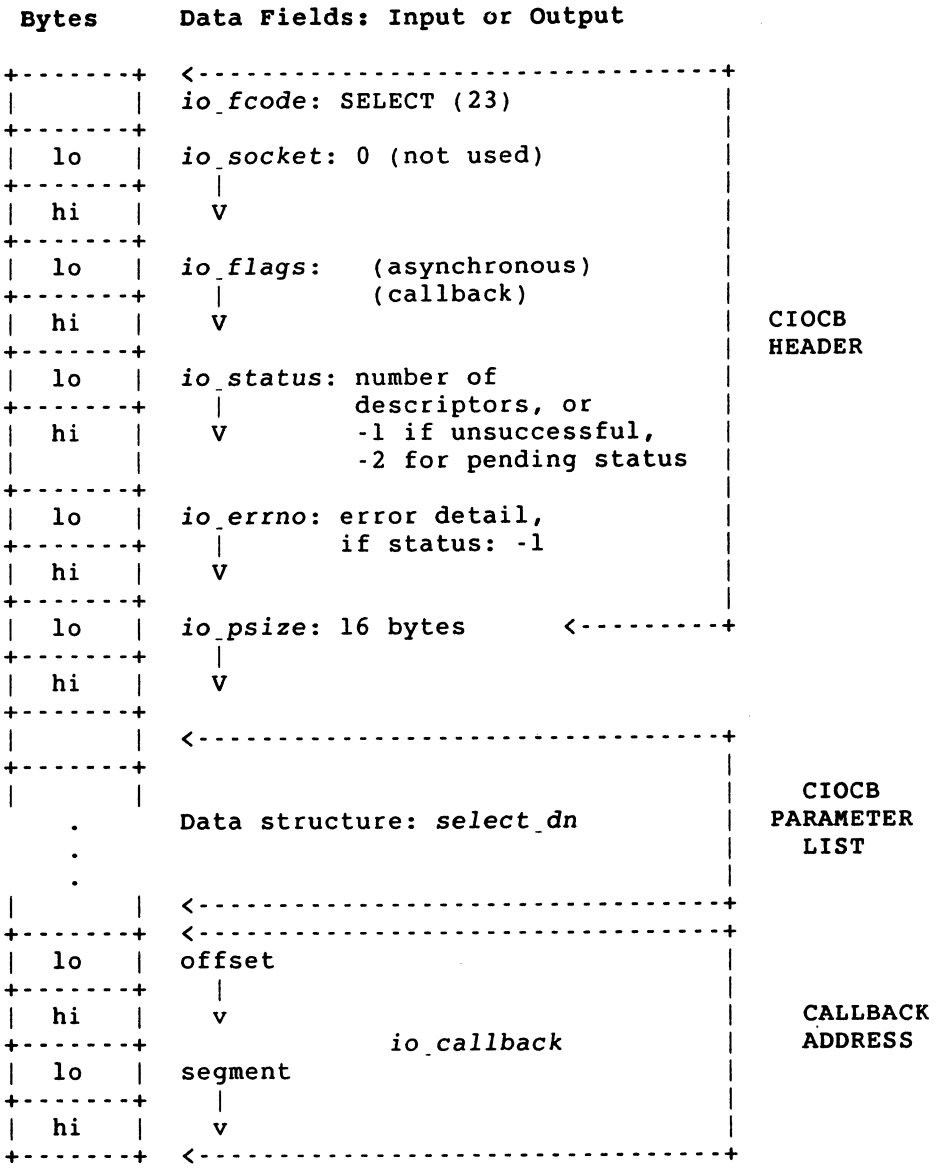
NAME

SELECT – check the I/O status of the network sockets.

IOCB Data Members



CIOCB Data Members – For Asynchronous Mode



DESCRIPTION

The *SELECT* call checks the network sockets specified by the bit masks in the data structure *select__dn* to see if they are ready for reading or writing, or if they have any outstanding out-of-band messages.

The *SELECT* call does not tell you if the logical link has been broken. You should use the *SELECT* call to help manage your *ACCEPT*, *SEND*, *SENDOOB*, *RCVD* and *RCVOOB* calls.

The I/O descriptors are long words which contain bit masks. Each bit in a mask represents one socket number. For example, socket "3" is the fourth bit or has a hex value of 8.

NOTE

The *SELECT* call can only check socket numbers in the range 0 to 31.

To specify the bit for any socket number, use the value created by the *ATTACH* or the *ACCEPT* call, as "*1 < <s*".

DESCRIPTION – For Asynchronous Mode

The *SELECT* call checks the network sockets specified by the bit masks in the data structure *select_dn* to see if they are ready for reading or writing, or if they have any outstanding out-of-band messages.

The *SELECT* call does not tell you if the logical link has been broken. You should use the *SELECT* call to help manage your *ACCEPT*, *SEND*, *SENDOOB*, *RCVD* and *RCVOOB* calls.

The *SELECT* call examines the network sockets until the call time out (see *sel_seconds*) or status is returned. If you specify multiple sockets to be examined, and one socket becomes detached, the *SELECT* call will return with the error message, EBADF. You must reissue the *SELECT* call in order to examine the remaining sockets.

The I/O descriptors are long words which contain bit masks. Each bit in a mask represents one socket number. For example, socket "3" is the fourth bit or has a hex value of 8.

NOTE

The *SELECT* call can only check socket numbers in the range 0 to 31.

To specify the bit for any socket number, use the value created by the *ATTACH* or the *ACCEPT* call, as "*1 < <s*".

Input Data

io_fcode specifies *SELECT* as the function code. It has a decimal value of 23.

io_socket specifies the socket number. This field is set to 0.

io_flags defines specific flag options. You must set this data member to 0. It is not used with the synchronous form of the *SELECT* call.

For Asynchronous Mode: You can set this data member to MSG_ASYNC and MSG_CALLBACK. MSG_ASYNC processes the asynchronous I/O form of the *SELECT* function call. MSG_CALLBACK allows the network to issue a callback routine when the *SELECT* call completes.

io_psize specifies the size of the data structure *select_dn* as 16 bytes.

select_dn specifies the select data structure which is used for examining bit masks. The user fills in data for each field. (See Appendix B on how *select_dn* is formatted.)

The structure contains the following data fields:

sel_nfds specifies the highest socket number to be checked. The bits from (1 < < 0) to (1 < < (nfds–1)) are examined.

sel_read specifies the socket numbers (as bit masks) to be examined for read ready. For listening sockets, a read ready condition indicates that an incoming connection request can be read and either accepted or rejected. For sequenced sockets, there is a complete message to be read. For stream sockets, there is some data to be read. If a socket disconnects or aborts, a read ready condition will always occur.

This descriptor can be given as a zero value if of no interest.

sel_write specifies the socket numbers (as bit masks) to be examined for write ready. A write ready condition exists when the logical link is available. This descriptor can be given as a zero value if of no interest.

sel_except specifies the socket numbers (as bit masks) to be examined for out-of-band data ready. There is a pending out-of-band data message to receive. This descriptor can be given as a zero value if of no interest.

sel_seconds defines the maximum interval to wait for a descriptor selection to be completed. If the time value is set to –1, the *SELECT* call will wait until an event occurs. If the time value equals 0, then the *SELECT* call will return after an immediate poll. If the time value is greater than zero, the *SELECT* call will return either after *n* seconds have expired, or when an event occurs, whichever one comes first.

io_callback **For Asynchronous Mode:** If the *MSG_CALLBACK* bit is set in *io_flags*, then *io_callback* specifies the 4 byte address of the function to be called by the network process when the function completes. (See Appendix B on how the data type *exptr* is formatted.)

Output Data

sel__read — If a socket is read ready, the bit is returned "on", and *sel__read* returns the socket numbers (as bit masks) to be examined. If the socket is not read ready, the bit is cleared.

sel__write — If a socket is write ready, the bit is returned "on", and *sel__write* returns the socket numbers (as bit masks) to be examined. If the socket is not write ready, the bit is cleared.

sel__except — If the socket is out-of-band data ready, the bit is returned "on", and *sel__except* returns the socket numbers (as bit masks) to be examined. If the socket is not out-of-band data ready, the bit is cleared.

For Asynchronous Mode: The number of sockets to be examined for read ready, write ready, or out-of-band data ready, are not returned until *io__status* has changed from a pending condition (–2) and the call has successfully completed.

io__status — returns the number of descriptors to be examined upon successful completion. If an error occurs, *io__status* returns a –1. The *io__errno* field will also contain additional error detail.

For Asynchronous Mode: If the call's status is still pending, a value of –2 is returned. The *io__status* field will return the number of descriptors upon successful completion. If an error occurs, *io__status* will return a –1. The *io__errno* field will also contain additional error detail.

io__errno — returns additional error detail if *io__status* returns a –1. (See the DIAGNOSTICS section for a possible error condition.)

Data Structure Type Summary

sel__ndfs	2 bytes
sel__read	4 bytes
sel__write	4 bytes
sel__except	4 bytes
sel__seconds	2 bytes

DIAGNOSTICS

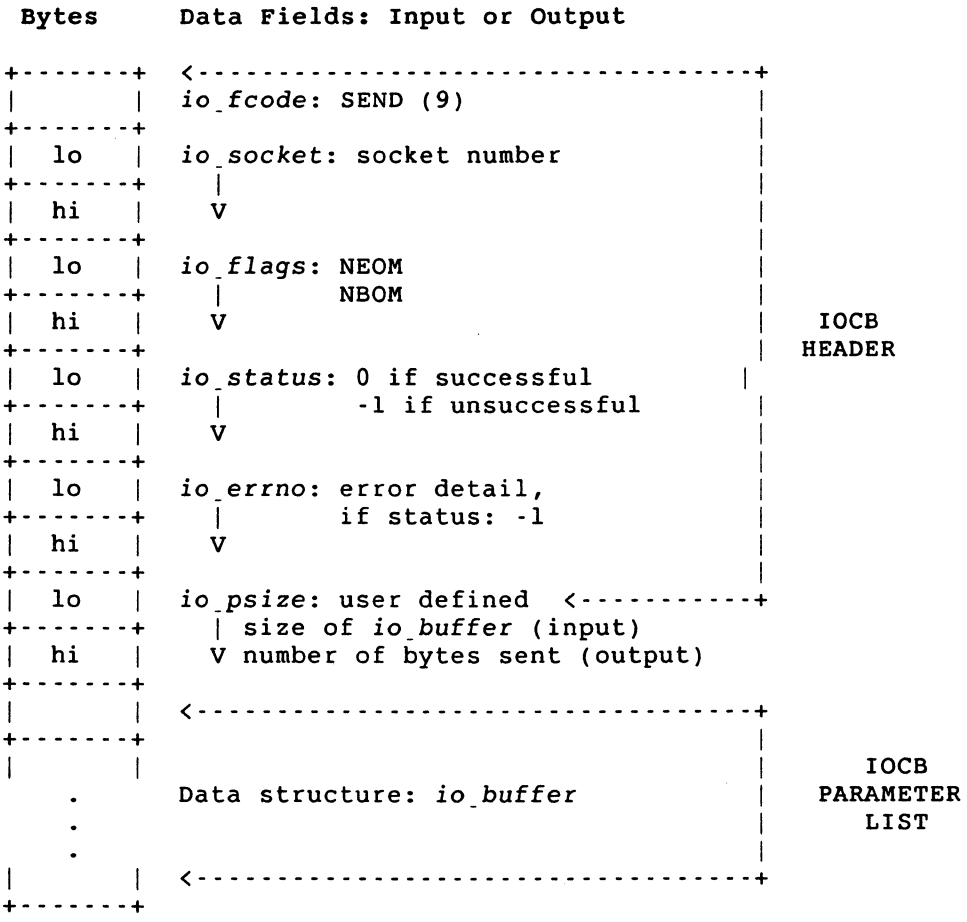
[EBADF] — One of the specified bit masks is an invalid descriptor.

6.7.15 SEND

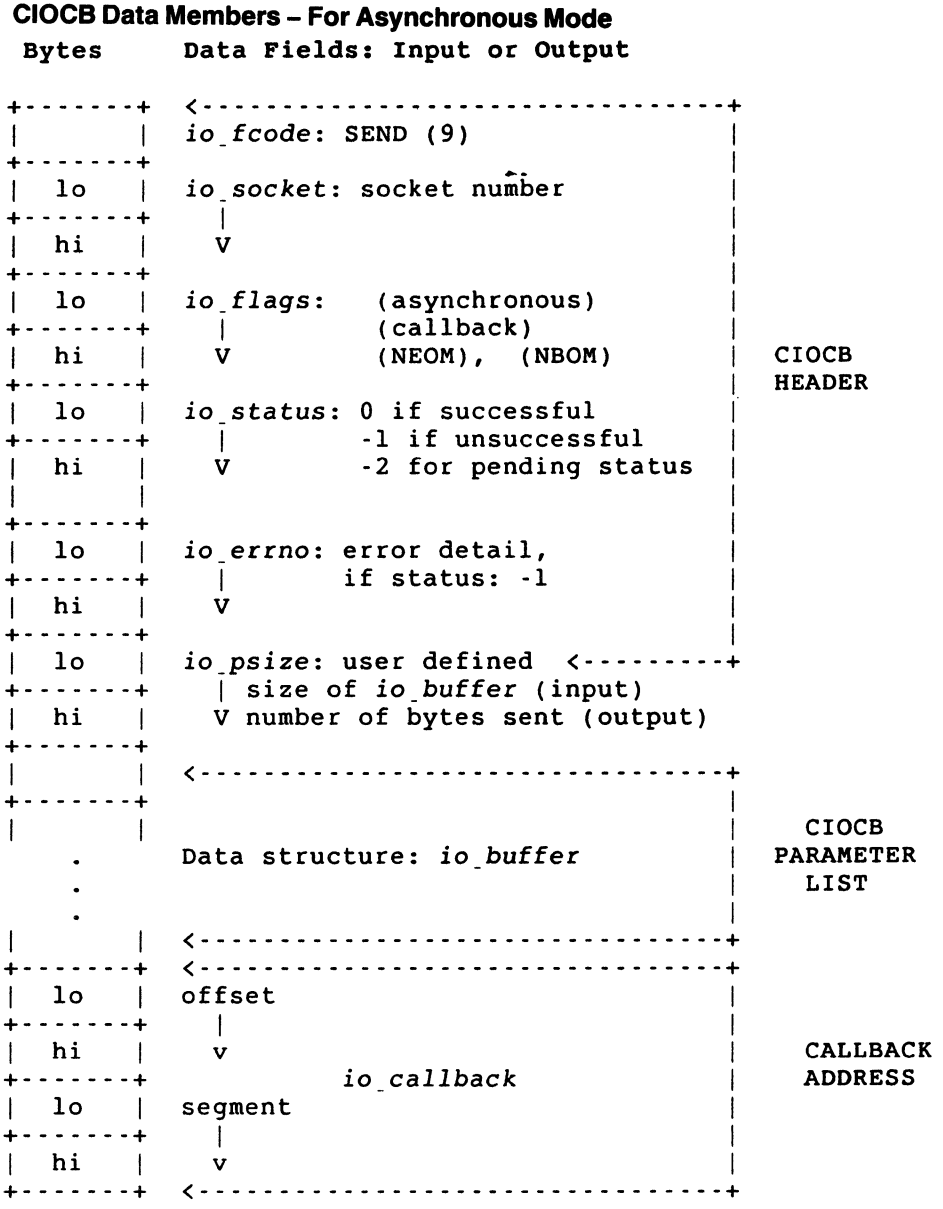
NAME

SEND – send data on a specified socket.

IOCB Data Members



CIOCB Data Members – For Asynchronous Mode



DESCRIPTION

The *SEND* call is used to transmit data to your peer. The client task uses the socket number returned by the *ATTACH* call. The server task uses the socket number returned by the *ACCEPT* call.

If you cannot get enough buffer space on a blocking socket, the call is blocked. You must wait until current transmissions are finished. If the socket is set to nonblocking, the call returns with –1 in *io_status*, and the error value EWOULDBLOCK in *io_errno*. If a socket disconnects, any outstanding data to be sent is discarded.

For sequenced sockets, you can send a multi-part message as if it is a single message. To do this, you should flag the *SEND* call with the required message options, NEOM (not end of message) and NBOM (not beginning of message).

NOTE

NEOM and NBOM are not valid options for stream sockets. Stream mode destroys all record boundaries.

The following example describes how to send a 3-part message and have the *SEND* call treat it as one message. The *io_flags* are set as follows:

- First buffer (*io_flags* = NEOM)
- Second buffer (*io_flags* = NEOM and NBOM)
- Third buffer (*io_flags* = NBOM)

At the receive end, the three-part message would be reconstructed and treated as a single message.

DESCRIPTION – For Asynchronous Mode

The *SEND* call is used to transmit data to your peer. The client task uses the socket number returned by the *ATTACH* call. The server task uses the socket number returned by the *ACCEPT* call.

When the asynchronous form of the *SEND* call is used, it is possible that the call may not complete. If so, there are two ways for you to check the status of the call:

- If the asynchronous form of the *SEND* call is used, you can also specify that a callback routine be used. The message option, MSG_CALLBACK, allows the network to issue a callback routine when the *SEND* call completes.
- If there is no callback routine, you can poll for status. To do this, examine the *io_status* field. A value of –2, 0, or –1 may be returned. A value of –2 indicates pending (no change) status. If a value of 0 is returned, and the *SEND* call has completed, the number of bytes transferred to your peer is returned in *io_psize*.

  If the call is unsuccessful, *io_status* returns a value of –1. Error detail will be contained in *io_errno*. For a description of these error messages, see the DIAGNOSTICS section.

For sequenced sockets, you can send a multi-part message as if it were a single message. To do this, you should flag the *SEND* call with the required message options, NEOM (not end of message) and NBOM (not beginning of message).

NOTE

NEOM and NBOM are not valid options for stream sockets. Stream mode destroys all record boundaries.

The following example describes how to send a 3-part message and have the *SEND* call treat it as one message. The *io_flags* are set as follows:

- First buffer (*io_flags* = NEOM)
- Second buffer (*io_flags* = NEOM and NBOM)
- Third buffer (*io_flags* = NBOM)

At the receive end, the three-part message would be reconstructed and treated as a single message.

Input Data

io_fcode specifies *SEND* as the function code. It has a decimal value of 9.

io_socket specifies the number for a socket created by the *ACCEPT* or the *CONNECT* call.

io_flags defines specific bit options. For sequenced sockets, you can send a multi-part message as if it were a single message. To do this, the *SEND* call is flagged with the message options, MSG_NEOM and MSG_NBOM. When the message is received, it would be reconstructed and treated as a single message.

The hexadecimal value for each option is listed in Appendix A.

For Asynchronous Mode: You can set *io_flags* to one or more of the following bits: MSG_ASYNC processes the asynchronous I/O form of the *SEND* function call. MSG_CALLBACK allows the network to issue a callback routine when the *SEND* call completes. For sequenced sockets, you can send a multi-part message as if it were a single message. To do this, the *SEND* call is flagged with the message options, MSG_NEOM and MSG_NBOM. When the message is received, it would be reconstructed and treated as a single message.

The hexadecimal value for each option is listed in Appendix A.

io_psize specifies the size of the user defined buffer.

io_buffer specifies the address of the buffer which contains the outgoing message. (Refer to Appendix B on how *io_buffer* is formatted.)

io_callback **For Asynchronous Mode:** If the *MSG_CALLBACK* bit is set in *io_flags*, then *io_callback* specifies the 4 byte address of the function to be called by the network process when the function completes. (See Appendix B on how the data type *exptr* is formatted.)

Output Data

io_psize specifies the number of transferred bytes upon successful completion.

io_status Upon successful completion, a value of 0 is returned. The number of bytes transferred to your peer is returned in *io_psize*. If an error occurs, *io_status* returns a –1. The *io_errno* field will also contain additional error detail.

For Asynchronous Mode: If the call's status is still pending, a value of –2 is returned. Upon successful completion, a value of 0 is returned. The number of bytes transferred to your peer is returned in *io_psize*. If an error occurs, *io_status* returns a –1. The *io_errno* field will also contain additional error detail.

io_errno returns error detail if *io_status* returns a –1. (See the DIAGNOSTICS section for a list of error conditions.)

Data Structure Type Summary

io_buffer 4 bytes

DIAGNOSTICS

When sending normal data, the following set of error messages can occur:

Blocking I/O

Message	Description
[EBADF]	The argument *io_socket* does not contain a valid socket number.
[EMSGSIZE]	The size of the outgoing message is more than 2048 bytes.
[ENOTCONN]	The *SEND* call did not complete and the link was disconnected.
[EPIPE]	The link has been disconnected, aborted, or shut down. No further messages can be sent.

Nonblocking I/O

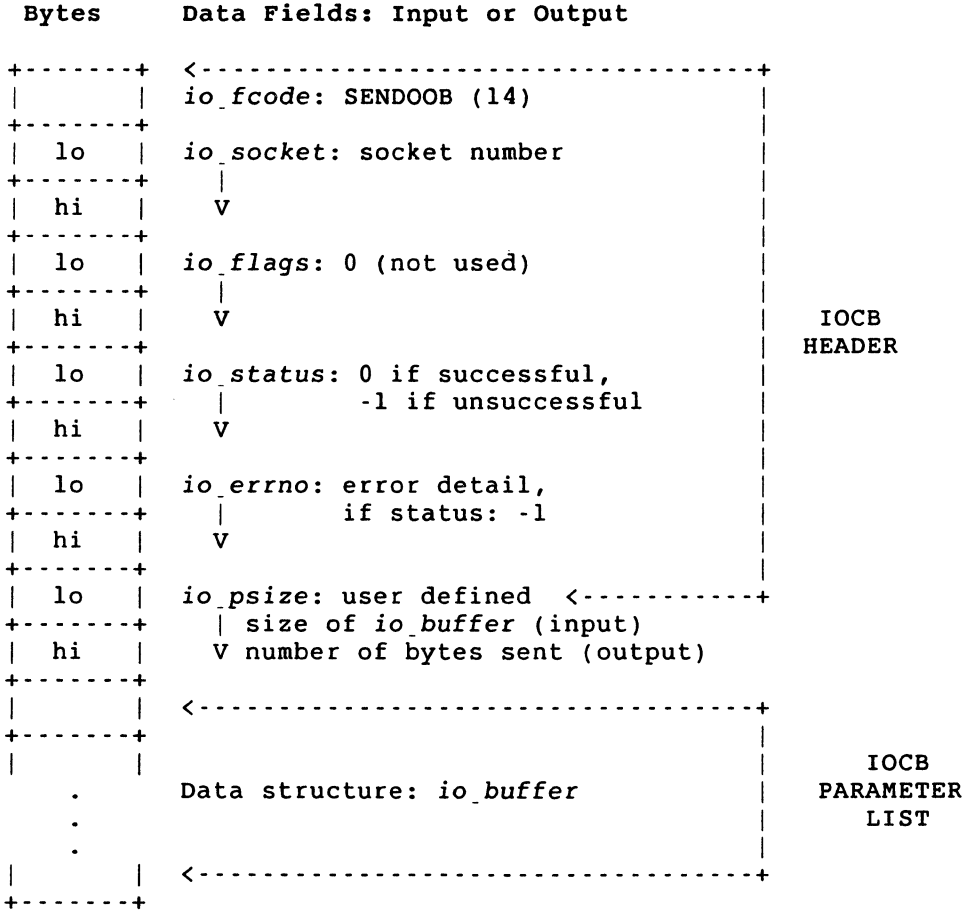
Message	Description
[EBADF]	The argument *io_socket* does not contain a valid socket number.
[EMSGSIZE]	The size of the outgoing message is more than 2048 bytes.
[ENOTCONN]	The *SEND* call did not complete and the link was disconnected.
[EPIPE]	The link has been disconnected, aborted, or shut down. No further messages can be sent.
[EWOULDBLOCK]	The outbound quota was full, and the message could not be sent. EWOULDBLOCK is not a valid error message for the asynchronous form of the *SEND* call.

6.7.16 SENDOOB

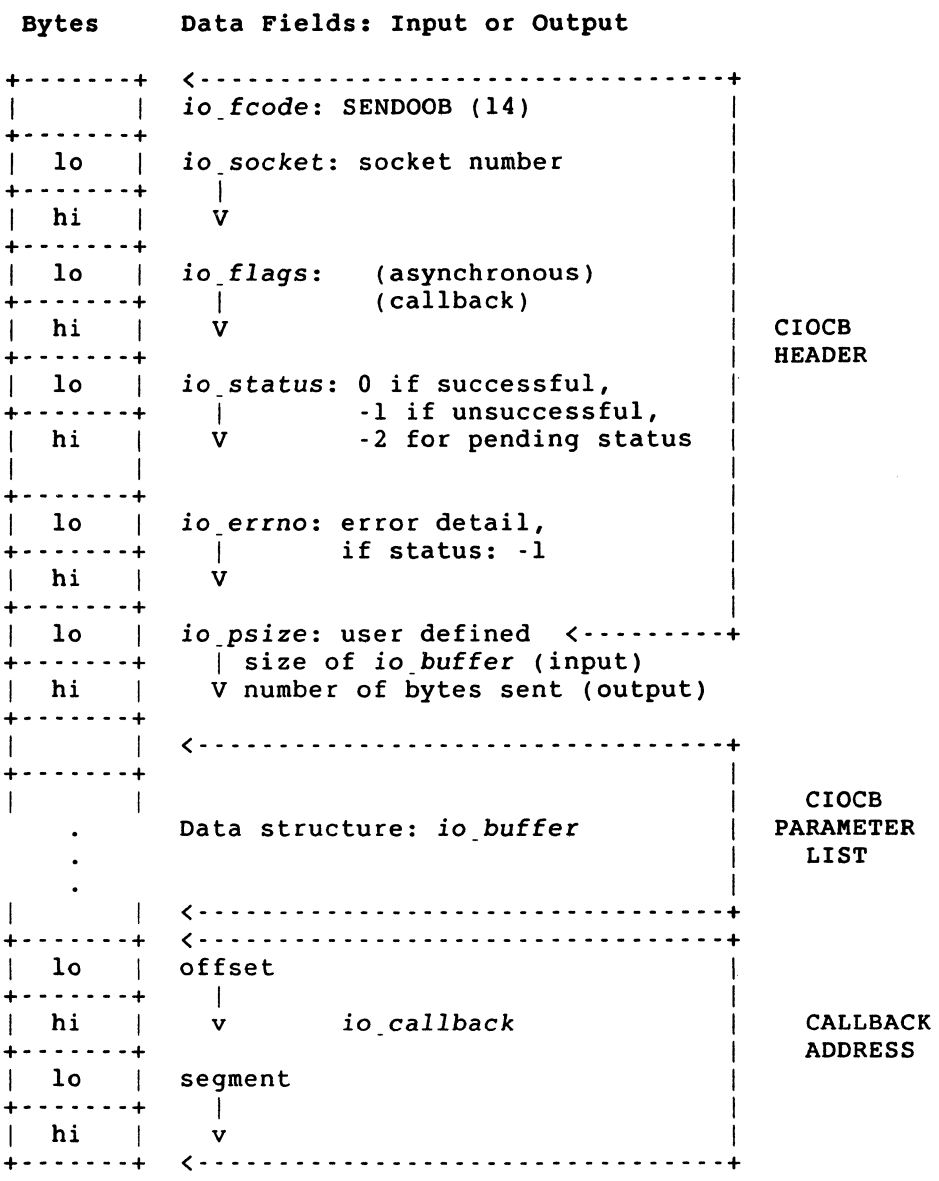
NAME

SENDOOB – send out-of-band messages on a specified socket.

IOCB Data Members



CIOCB Data Members – For Asynchronous Mode



DESCRIPTION

The *SENDOOB* call is used to send out-of-band data to your peer. An out-of-band message is a high priority message that you can send to your peer. Out-of-band messages are sent to a receiving task ahead of normal messages. If a socket disconnects, any outstanding data to be sent is lost.

DESCRIPTION – For Asynchronous Mode

The *SENDOOB* call is used to send out-of-band data to your peer. An out-of-band message is a high priority message that you can send to your peer. Out-of-band messages are sent to a receiving task ahead of normal messages. If a socket disconnects, any outstanding data to be sent is lost.

For the asynchronous form of the *SENDOOB* call, you can only have 1 active out-of-band message and 1 pending out-of-band message.

When the asynchronous form of the *SENDOOB* call is used, it is possible that the call may not complete. If so, there are two ways for you to check the status of the call:

- If the asynchronous form of the *SENDOOB* call is used, you can also specify that a callback routine be used. The message option, MSG_CALLBACK, allows the network to issue a callback routine when the *SENDOOB* call completes.
- If there is no callback routine, you can poll for status. To do this, examine the *io_status* field. A value of –2, 0, or –1 may be returned. A value of –2 indicates pending (no change) status. If a value of 0 is returned, and the *SEND* call has completed, the number of bytes transferred to your peer is returned in *io_psize*.

  If the call is unsuccessful, *io_status* returns a value of –1. Additional error detail will also be contained in *io_errno*. For a description of these error messages, see the DIAGNOSTICS section.

Input Data

io_fcode specifies *SENDOOB* as the function code. It has a decimal value of 14.

io_socket specifies the number for a socket created by the *ACCEPT* or *CONNECT* call.

io_flags defines specific bit options. You must set this data member to 0. It is not used with the synchronous form of the *SENDOOB* call.

For Asynchronous Mode: You can set *io_flags* to MSG_ASYNC and MSG_CALLBACK. MSG_ASYNC processes the asynchronous I/O form of the *SENDOOB* function call. MSG_CALLBACK allows the network to issue a callback routine when the *SENDOOB* call completes.

The hexadecimal value of each flag option is listed in Appendix A.

io_psize specifies the size of the user defined buffer.

io_buffer specifies the address of the buffer which contains the outgoing out-of-band message. (Refer to Appendix B on how *io_buffer* is formatted.)

io_callback **For Asynchronous Mode:** If the *MSG_CALLBACK* bit is set in *io_flags*, then *io_callback* specifies the 4-byte address of the function to be called by the network process when the function completes. (See Appendix B on how the data type *exptr* is formatted.)

Output Data

io_psize specifies the number of transferred bytes upon successful completion.

io_status Upon successful completion, a value of 0 is returned. The number of bytes transferred to your peer is returned in *io_psize*. If an error occurs, *io_status* returns a –1. The *io_errno* field will also contain additional error detail.

For Asynchronous Mode: If the call's status is still pending, a value of –2 is returned. Upon successful completion, a value of 0 is returned. The number of bytes transferred to your peer is returned in *io_psize*. If an error occurs, *io_status* returns a –1. The *io_errno* field will also contain additional error detail.

io_errno returns error detail if *io_status* returns a –1. (See the DIAGNOSTICS section for a list of error conditions.)

Data Structure Type Summary

io_buffer 4 bytes

io_callback 4 bytes

DIAGNOSTICS

When sending out-of-band data, the following set of error messages can occur:

Blocking I/O

Message	Description
[EALREADY]	The out-of-band message could not be sent. A similar transmission request is still in progress.
[EBADF]	The argument *io_socket* does not contain a valid socket number.
[EMSGSIZE]	The size of the outgoing message is more than 16 bytes.
[ENOTCONN]	The *SEND* call did not complete and the link was disconnected.
[EPIPE]	The link has been disconnected, aborted, or shut down. No further messages can be sent.

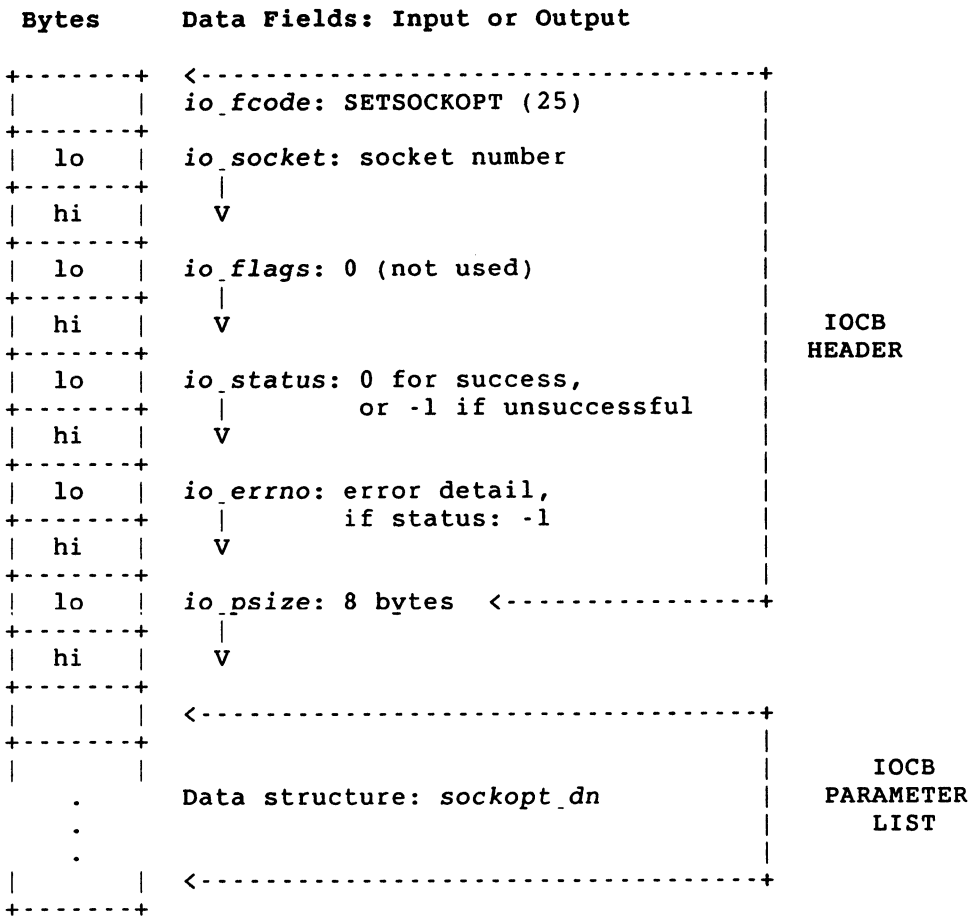
Message	Description
[EALREADY]	The out-of-band message could not be sent. A similar transmission request is still in progress.
[EBADF]	The argument *io_socket* does not contain a valid socket number.
[EMSGSIZE]	The size of the outgoing message is more than 16 bytes.
[ENOTCONN]	The *SEND* call did not complete and the link was disconnected.
[EPIPE]	The link has been disconnected, aborted, or shut down. No further messages can be sent.

6.7.17 SETSOCKOPT and GETSOCKOPT

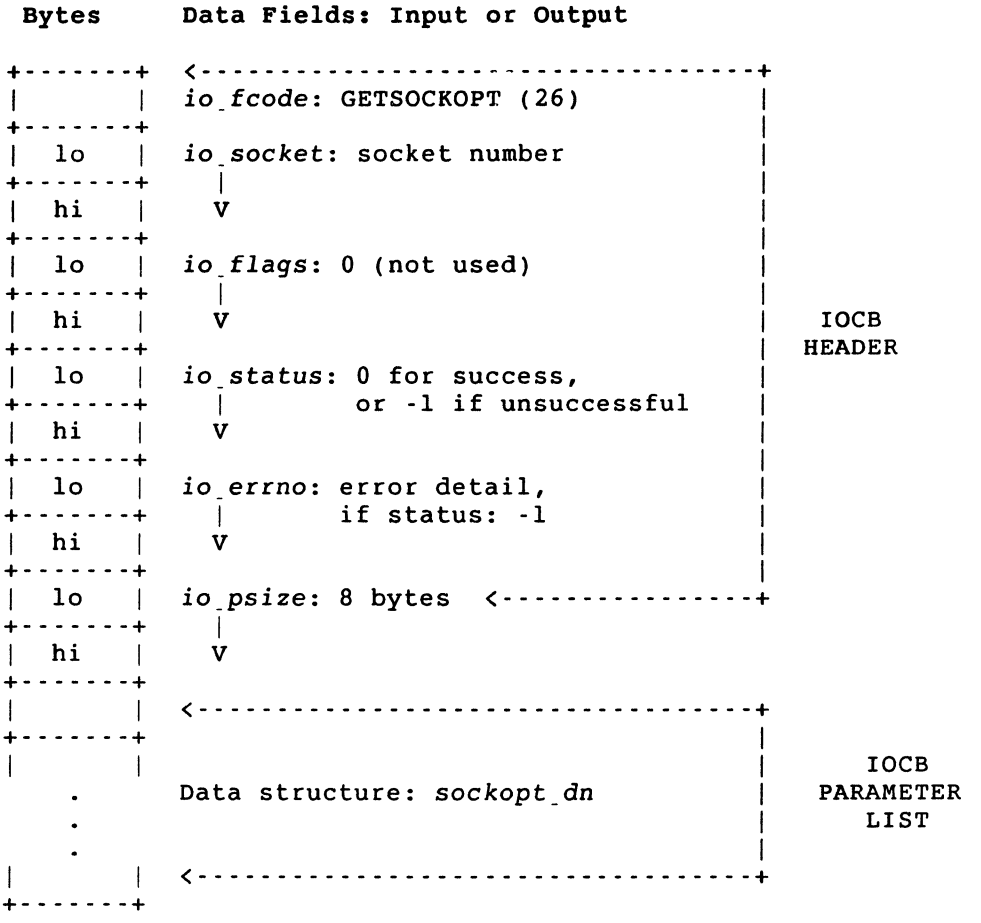
NAME

SETSOCKOPT and GETSOCKOPT – set and get the options associated with sockets.

IOCB Data Members



IOCB Data Members



DESCRIPTION

The *SETSOCKOPT* and *GETSOCKOPT* calls manipulate various options associated with a socket. Options exist at multiple levels; therefore you must specify the level number for the desired operation.

NOTE

In the following discussion, references are made to symbolic values. See Appendix A for details.

At the socket level (SOL_SOCKET), the options include:

- **SO_KEEPALIVE.** If this option is set on a socket, any links and sockets associated with this socket will remain active, despite any attempts to disconnect them.

  NOTE

  Before you can terminate a connection over a socket with the option SO_KEEPALIVE set, you must first issue a *SETSOCKOPT* call with SO_KEEPALIVE turned off.

  You then issue either the *ABORT*, *DETACH*, or *DISCONNECT* call. The logical links (if any) are disconnected, and the socket and associated sockets (if any) are aborted or only disconnected. However, if you issue either call without turning off SO_KEEPALIVE, the socket remains attached, and the links (if any) stay active.

- **SO_LINGER.** SO_LINGER controls the actions taken when unsent messages are queued on a socket and a *DISCONNECT* call is issued. If SO_LINGER is set, the connection is maintained until the outstanding messages have been sent. This is the default condition.

- **SO_DONTLINGER.** SO_DONTLINGER also controls the actions of unsent messages. If SO_DONTLINGER is set, and the *DISCONNECT* call is issued, any outstanding messages queued to be sent will be lost. The connection is then terminated.

- **SO_REUSEADDR.** SO_REUSEADDR allows the reuse of a name already bound to a socket. For most situations, a name is bound to a socket only once. However, this option enables you to reuse the same name. This particular option **must** only be used for outgoing connection requests. It cannot be used for incoming connections.

  VAX/VMS proxy access by user name is made possible if the client task uses the *BIND* call specifying the user name as the object name. If you wish to make more than one proxy connection with the same user name, you must use the SO_REUSEADDR option.

At the DECnet level (DNPROTO_NSP), socket options may specify the way in which a connection request is accepted or rejected, may be used to set up optional user data and/or access control information, and may be used to obtain current link state information. The following socket options can be specified:

- **DSO_ACCEPTMODE.** The accept option mode is used at the DECnet level for processing *ACCEPT* calls. A socket must be bound (see *BIND*, Section 6.7.4) before specifying this option. There are two values which can be supplied for this option. They are immediate mode, ACC_IMMED, and deferred mode, ACC_DEFER.

  - **ACC_IMMED.** ACC_IMMED mode is the default condition for this option. When immediate mode is in effect, control is immediately returned to the

server task following an *ACCEPT* call with the connection request accepted. The access control information and/or optional user data is ignored by the server task.

- **ACC__DEFER.** ACC__DEFER mode indicates that the server task completes the *ACCEPT* call without fully completing the connection to the client task. In this case, the server task can examine the optional access control or user data before it decides to accept or reject the connection request. The server task can then issue the *SETSOCKOPT* call with the appropriate reject or accept option.

- **DSO__CONACCEPT.** DSO__CONACCEPT allows the server task to accept the pending connection on the socket returned by the *ACCEPT* call. The original listening socket was set to deferred accept mode. Any optional user data previously set by DSO__CONDATA will also be sent.

- **DSO__CONREJECT.** DSO__CONREJECT allows the server task to reject the pending connection on the socket returned by the *ACCEPT* call. The original listening socket was set to deferred accept mode. Any optional user data previously set by DSO__DISDATA will also be sent. The reject reason is the value passed with this option.

- **DSO__CONDATA.** DSO__CONDATA allows up to 16 bytes of optional user data to be set by the *SETSOCKOPT* call. It can be sent as a result of the *CONNECT* or the *ACCEPT* (with the deferred option) calls. The optional data is passed in a structure of type *optdata__dn*. (See Appendix B on how *optdata__dn* is formatted.) The data is read by the task issuing the *GETSOCKOPT* call with this option.

- **DSO__DISDATA.** DSO__DISDATA allows up to 16 bytes of optional data to be set by the *SETSOCKOPT* call. It can be sent as a result of the *DISCONNECT* call. The optional data is passed in a structure of type *optdata__dn*. (See Appendix B on how *optdata__dn* is formatted.) The data is read by the task issuing the *GETSOCKOPT* call with this option.

- **DSO__CONACCESS**. DSO__CONACCESS allows access control information to be passed by the user task. This information is set with the *SETSOCKOPT* call. The access data is sent to the server task. It is passed with the *CONNECT* call in a structure of type *accessdata__dn*. (See Appendix B on how *accessdata__dn* is formatted.) The access data is read by the task issuing the *GETSOCKOPT* call with this option.

- **DSO__LINKINFO**. DSO__LINKINFO determines the state of the logical link connection.

  When *GETSOCKOPT* call is issued with this option, the state of the logical link is returned in a logical link information data structure, *linkinfo__dn*. (See Appendix B on how *linkinfo__dn* is formatted.)

Input Data

io_fcode specifies *SETSOCKOPT* as the function code. It has a decimal value of 25. This data member also specifies *GETSOCKOPT* as the function code. It has a decimal value of 26.

io_socket specifies the number for a socket created by the *ACCEPT* and/or deferred-mode *ACCEPT* or *ATTACH* call.

io_flags defines specific flag options. You must set this data member to 0. It is not used with the *SETSOCKOPT* or the *GETSOCKOPT* call.

io_psize specifies the size of the data structure *sockopt_dn* as 8 bytes.

sockopt_dn specifies the socket option data structure. (See Appendix B on how *sockopt_dn* is formatted.)

The structure contains the following data fields:

sop_level specifies the level at which options are manipulated.

If the level is set to SOL_SOCKET, then *sop_optval* and *sop_optlen* are ignored. If the level is set to DNPROTO_NSP, then the rest of the data structure can contain either access control or optional user data, or access mode information.

sop_optname specifies options to be interpreted.

When the socket level is set to DNPROTO_NSP, *sop_optname* can be set to one of 6 specific options (for example, DSO_CONDATA). See Appendix A for a list of the specific options.

sop_optval, sop_optlen specify access option values used with the *SETSOCKOPT* and the *GETSOCKOPT* calls. The interpretation of each argument is function dependent as shown here:

SETSOCKOPT call

sop_optval specifies the pointer to a buffer which contains information for setting access option values.

sop_optlen specifies the size of the option value buffer.

GETSOCKOPT call

sop_optval specifies the pointer to a buffer which will contain the returned value for the requested option(s).

sop_optlen is a value result parameter. It should initially contain the size of the buffer pointed to by *sop_optval*. On return, it will contain the actual size of the returned value.

Output Data

io__status returns a 0 upon successful completion. If an error occurs, *io__status* returns a −1. The *io__errno* field will also contain additional error detail.

io__errno returns error detail if *io__status* returns a −1. (See the DIAGNOSTICS section for a list of error conditions.)

GETSOCKOPT call

sop__optval specifies the pointer to a buffer which contains the returned value for the requested socket option(s).

sop__optlen is a value result parameter. On return, it contains the actual size of the returned value for the buffer pointed to by *sop__optval*.

Data Structure Type Summary

sop__level 2 bytes

sop__optname 2 bytes

sop__optval 4 bytes

op__optlen 4 bytes

snd__how 2 bytes

DIAGNOSTICS

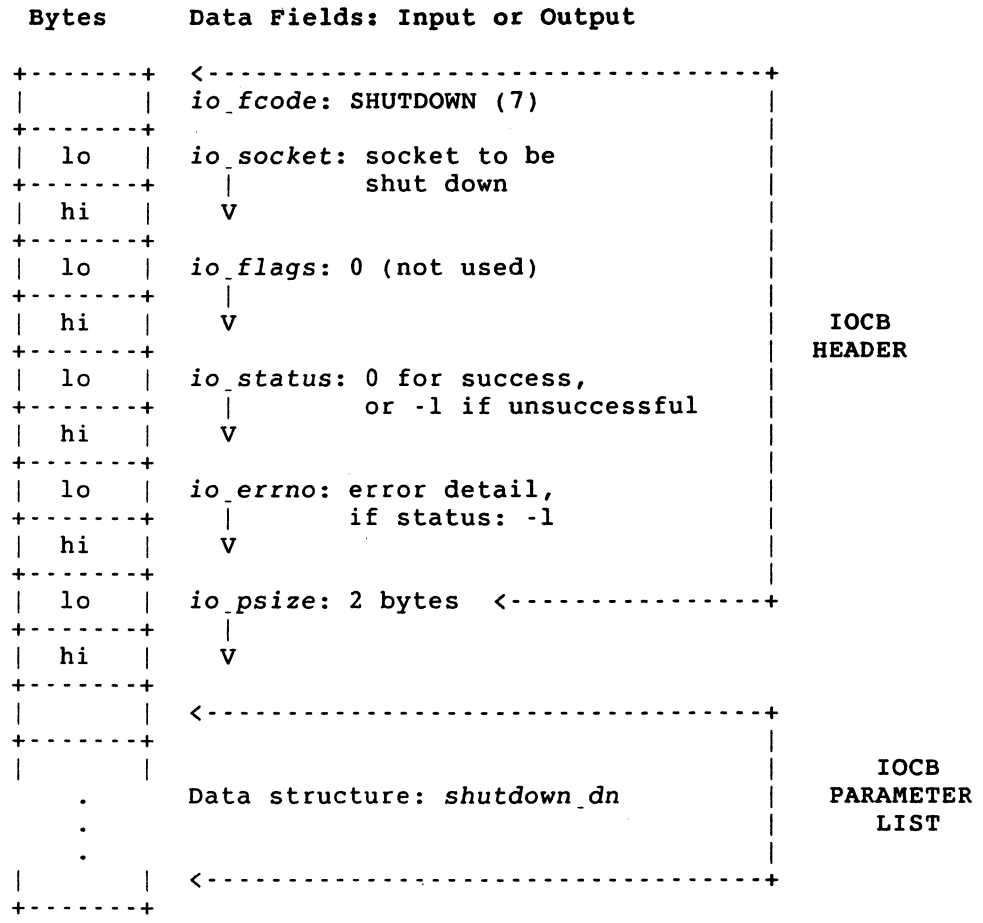
[EACCES]	Unable to disconnect the socket.
[EBADF]	The argument *io__socket* does not contain a valid socket number.
[ECONNABORTED]	The accept connect did not complete. The peer task disconnected and the connection was aborted.
[EDOM]	The acceptance mode is not valid.
[ENOBUFS]	There are no available buffers for optional access control and/or user data.
[ENOPROTOOPT]	There was no access control information supplied with the connection request.
[EOPNOTSUPP]	The option is unknown.

6.7.18 SHUTDOWN

NAME

SHUTDOWN – shut down all or part of a full duplex logical link.

IOCB Data Members



DESCRIPTION

The *SHUTDOWN* call causes all or part of a full duplex connection on the original socket to be shut down.

Input Data

io_fcode specifies *SHUTDOWN* as the function code. It has a decimal value of 7.

io_socket specifies the socket number.

io_flags defines specific flag options. You must set this data member to 0. It is not used with the *SHUTDOWN* call.

io_psize specifies the size of the data structure *shutdown_dn* as 2 bytes.

shutdown_dn specifies the type of shutdown. (See Appendix B on how *shutdown_dn* is formatted.)

The structure contains the following data field:

snd_how specifies the type of shutdown. This argument can be set to:

0 which disallows further receives.

1 which disallows further sends.

2 which disallows further sends and receives.

Output Data

io_status returns a 0 upon successful completion. If an error occurs, *io_status* returns a −1. The *io_errno* field will also contain additional error detail.

io_errno returns error detail if *io_status* returns a −1. (See the DIAGNOSTICS section for a list of error conditions.)

DIAGNOSTICS

[EBADF] The argument *io_socket* does not contain a valid descriptor.

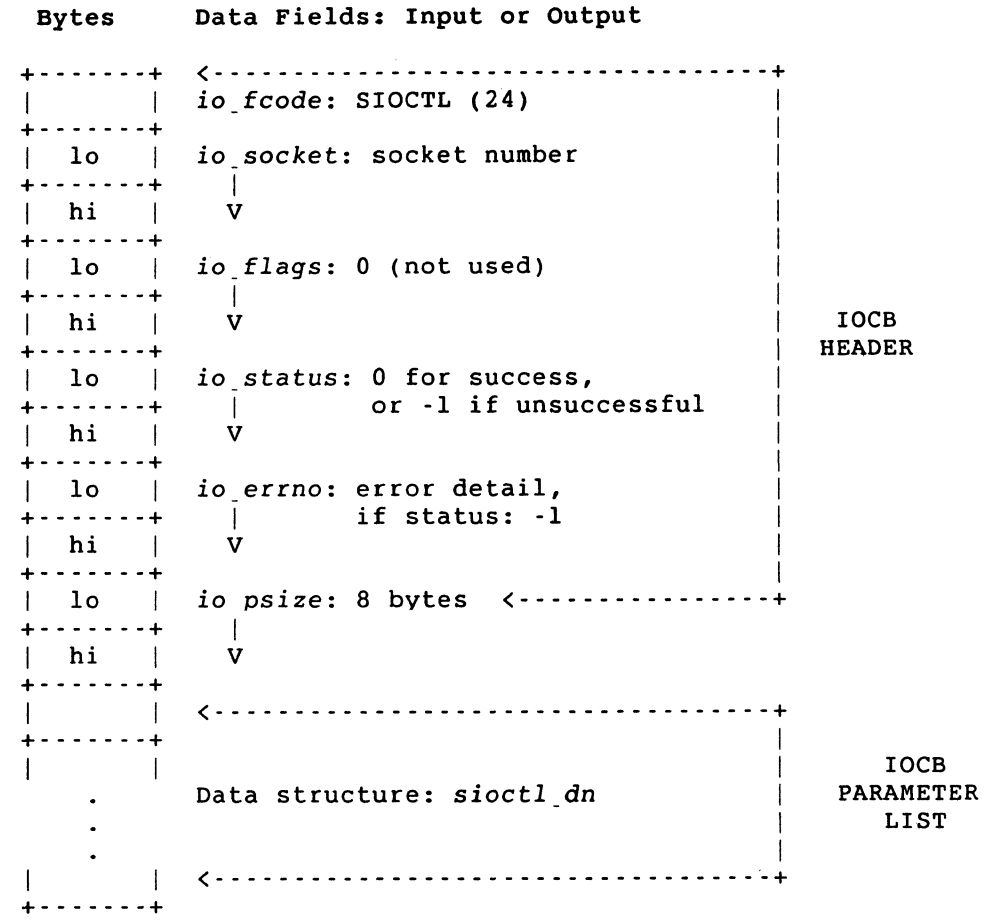
[ENOTCONN] The specified socket is not connected.

6.7.19 SIOCTL

NAME

SIOCTL – control the operations of open sockets.

IOCB Data Members



DESCRIPTION

The *SIOCTL* call controls the operations of open sockets. The call indicates whether an argument is an input or output argument and the size of the specific argument in bytes.

Input Data

io_fcode specifies *SIOCTL* as the function code. It has a decimal value of 24.

io_socket specifies the socket number.

io_flags defines specific flag options. You must set this data member to 0. It is not used with the *SIOCTL* call.

io_psize specifies the size of the data structure *sioctl_dn* as 8 bytes.

sioctl_dn specifies the socket I/O control function data structure. (See Appendix B on how *sioctl_dn* is formatted.)

The structure contains the following data fields:

sio_s This data field is ignored.

sio_request specifies the I/O control function to be used. The control levels include:

FIONREAD returns the total byte count of all messages waiting to be read. The *argp* field points to a word.

FIONBIO sets/clears blocking or nonblocking I/O operation. *argp* points to a byte that contains a value of 0 or 1. For blocking I/O, *argp* should point to a value of 0. For nonblocking I/O, *argp* should point to a value of 1.

FIORENUM renumbers an assigned socket number to another number. In this way, the original socket number is made available again. The valid range for socket numbers is 0 to 31. *argp* points to a word.

The *SELECT* function call cannot accept socket numbers that exceed this range. (See Section 6.7.14 for details.)

If you specify a socket number that is already in use, an error message, EEXIST, is returned.

argp specifies the address of the argument list.

Output Data

io_status returns a 0 upon successful completion. When the call is successful, *argp* (only if it is FIONREAD) returns the total byte count of all messages waiting to be read.

If an error occurs, *io_status* returns a −1. The *io_errno* field will also contain error detail.

io_errno returns error detail if *io_status* returns a −1. (See the DIAGNOSTICS section for a list of error conditions.)

Data Structure Type Summary

sio_s	2 bytes
sio_request	2 bytes
argp	4 bytes

DIAGNOSTICS

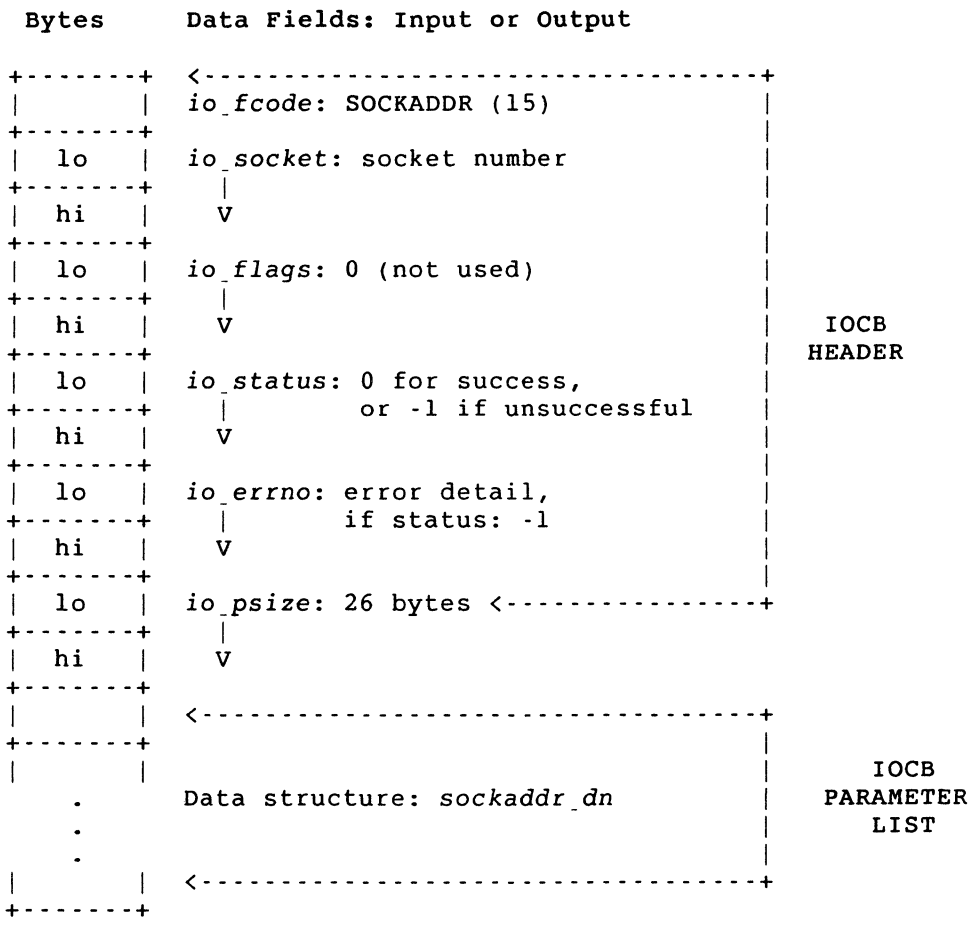
[EBADF]	The argument *io_socket* does not contain a valid socket number.
[EOPNOTSUPP]	The socket type does not support the socket I/O operation.

6.7.20 SOCKADDR

NAME

SOCKADDR – retrieve socket information set by the *BIND* call.

IOCB Data Members



DESCRIPTION

The *SOCKADDR* call returns socket information set by the *BIND* call. If no *BIND* call was ever executed, undefined results are returned in output data fields.

Input Data

io_fcode specifies *SOCKADDR* as the function code. It has a decimal value of 15.

io_socket specifies the number for a socket which was bound to a name by the *BIND* call.

io_flags defines specific flag options. You must set this data member to 0. It is not used with the *SOCKADDR* call.

io_psize specifies the size of the data structure *sockaddr_dn* as 26 bytes.

Output Data

io_status returns a 0 upon successful completion. If an error occurs, *io_status* returns a –1. The *io_errno* field will also contain error detail.

io_errno returns error detail if *io_status* returns a –1. (See the DIAGNOSTICS section for a list of error conditions.)

sockaddr_dn specifies the socket address data structure. A user retrieves data from the fields filled in by this function call. (See Appendix B on how *sockaddr_dn* is formatted.)

The following data fields can be filled in by this function call:

sdn_family is the address family AF_DECnet.

sdn_objnum is the object number for the local task. It can be a number from 0 to 255.

sdn_objnamel is the size of the object name.

sdn_objname is the object name of the local task. It can be up to a 16-byte array. It is only used when *sdn_objnum* equals 0.

sdn_add specifies the address structure for the local node. (See Appendix B on how *sockaddr_dn* is formatted.)

Data Structure Type Summary

sdn_family 2 bytes

sdn_flags 1 byte

sdn_objnum 1 byte

sdn_objnamel 2 bytes

sdn_objname 16-byte array

sdn_add 4 bytes

DIAGNOSTICS

[EBADF] The argument *io_socket* does not contain a valid socket number.

A
Socket Definitions

The following definitions are related to socket types, option flags, and other related socket definitions. The symbols that appear in this appendix are defined in the DECnet header files.

A.1 Communications Domain

DECnet–VAXmate supports the following communications domain:

Decimal Value	Domain	Description
1	AF__DECnet	Enables multiple computer systems to participate in communications and resource sharing within a DECnet network. The symbol is defined in the <socket.h> header file.

A.2 DECnet Layers

The following DECnet layers are supported by DECnet–VAXmate:

Hexadecimal/ Decimal Value	Layer	Description
0xffff	SOL__SOCKET	Specifies the socket session interface layer.
1	DNPROTO__NSP	Specifies the DECnet layer. How a connection request is accepted/rejected, optional access control and/or user data, or link state can be specified. (See Appendix B for a list of defined data structures.) These symbols are defined in the <socket.h> header file.

A.3 DECnet Objects

Certain DECnet object numbers are used as arguments to the *dnet_conn* call. The following are ASCII strings:

Object	ASCII String
DNOBJ_FAL	#17 (File Access Listener)
DNOBJ_NICE	#19 (Network Information and Control Exchange)
DNOBJ_TERM	#23 (Network command terminal handler – host side)
DNOBJ_MIRROR	#25 (Loopback mirror – MIR)
DNOBJ_EVR	#26 (Event receiver – EVR)
DNOBJ_MAIL11	#27 (Personal message utility)
DNOBJ_PHONE	#29 (Phone utility)
DNOBJ_CTERM	#42 (Command terminal operations)
DNOBJ_DTR	#63 (DECnet test receiver tool – DTR)

The following are decimal numbers:

Decimal Value	Object	Process Type
17	DNOBJECT_FAL	File Access Listener
19	DNOBJECT_NICE	Network Information and Control Exchange
23	DNOBJECT_DTERM	Network command terminal handler – host side
25	DNOBJECT_MIRROR	Loopback mirror (MIR)
26	DNOBJECT_EVR	Event receiver (EVR)
27	DNOBJECT_MAIL11	Personal message utility
29	DNOBJECT_PHONE	Phone utility
42	DNOBJECT_CTERM	Command terminal operations
63	DNOBJECT_DTR	DECnet test receiver tool (DTR)

These symbols are defined in the <dn.h> header file.

A.4 DECnet Options

At the DECnet layer (DNPROTO_NSP), socket options can define how a connection request is accepted/rejected, specify optional user data and/or access control information, or obtain current link state information. The following options can be used to specify or retrieve data with the *setsockopt* and *getsockopt* function calls:

Decimal Value	Option	Description
1	DSO_CONDATA	Allows up to 16 bytes of optional user data to be set by the *setsockopt* call. The optional data is passed in the *optdata_dn* data structure. The user task reads the data by issuing the *getsockopt* call with the connect option. The call returns a connect status.
2	DSO_DISDATA	Allows up to 16 bytes of optional user data to be set by the *setsockopt* call. The optional data is passed in the *optdata_dn* data structure. The user task reads the data by issuing the *getsockopt* call with the disconnect option. The call returns a disconnect status.
3	DSO_CONACCESS	Allows access control information to be set by the *setsockopt* call. The access control information is passed in the *accessdata_dn* data structure. The user task reads the data by issuing the *getsockopt* call. The information is processed once the task issues the *accept* call.
4	DSO_ACCEPTMODE	Defines the way in which a user task accepts a pending *accept* call. A socket must issue a *bind* call before this option is valid. The acceptance mode can be specified as follows:
0	ACC_IMMED	Specifies the default condition. The *accept* call is immediately completed.
1	ACC_DEFER	Allows the server task to complete the *accept* call without fully completing the connection to the client task. The server task can examine the source address, access control and/or optional user data before accepting or rejecting the pending connection.

Decimal Value	Option	Description
5	DSO__CONACCEPT	Allows the server task to accept the pending connection on the socket previously set to the deferred accept mode (ACC__DEFER). Any optional user data previously set by DSO__CONDATA will also be sent.
6	DSO__CONREJECT	Allows the server task to reject the pending connection on the socket previously set to the deferred accept mode (ACC__DEFER). Any optional user data previously set by DSO__DISDATA will also be sent.
7	DSO__LINKINFO	Allows the user task to retrieve the state of the logical link connection. There are four supported link states. (See Section A.6.) The link state is returned in the *linkinfo__dn* data structure. It is retrieved with the *getsockopt* call.
7	DSO__MAX	Specifies the allowable number of defined socket options. These symbols are defined in the <dn.h> header file.

A.5 Flag Options

The following bits can be set in the *io__flags* field which is included in the IOCB and/or CIOCB:

Hexadecimal Value	Message	Description
0x1	MSG__OOB	Process out-of-band messages with the *send* and *recv* calls. The symbol is defined in the <socket.h> header file.
0x2	MSG__PEEK	Read the next pending message without removing the message from the receive queue. The symbol is defined in the <socket.h> header file.
0x8	MSG__ASYNC	Process the asynchronous I/O form of DECnet function calls. The symbol is defined in the <socket.h> header file.

Hexadecimal Value	Message	Description
0x10	MSG__CALLBACK	Allows the network to issue a callback routine when a specific function call completes. The symbol is defined in the <socket.h> header file.
0x20	MSG__NEOM	For sequenced sockets, use MSG__NEOM with the option MSG__NBOM to send a multi-part message as if it were a single message. To receive a single message in multiple parts, flag the receive call with MSG__NEOM. The symbol is defined in the <socket.h> header file.
0x40	MSG__NBOM	For sequenced sockets, flag the send call with MSG__NBOM and MSG__NEOM to send a multi-part message as if it were a single message. MSG__NBOM cannot be used for receive operations. The symbol is defined in the <socket.h> header file.

A.6 Logical Link States

The following logical link states are supported by DECnet–VAXmate.

Decimal Value	State	Description
0	LL__INACTIVE	The logical link is inactive.
1	LL__CONNECTING	The logical link is connecting.
2	LL__RUNNING	The logical link is running.
3	LL__DISCONNECTING	The logical link is disconnecting.

The symbols are defined in the <dn.h> header file.

A.7 Maximum Number of Incoming Connection Requests

The maximum number of incoming connection requests is specified as follows:

Hexadecimal Value	Message	Description
0x5	SOMAXCONN	Defines the maximum number of incoming connection requests which are allowed on the specified socket. The symbol is defined in the <socket.h> header file.

A.8 Socket Interface Options

At the socket level (SOL_SOCKET), the following options exist:

Hexadecimal Value	Flag	Description
0x04	SO_REUSEADDR	Allows the reuse of a bound socket name. This option must only be used for outgoing connection requests.
0x08	SO_KEEPALIVE	If this option is set on a socket, any links and sockets associated with this socket remain active, despite any attempts to abort, detach and/or disconnect them. The effects of ABORT, DETACH, and DISCONNECT functions are only realized after SO_KEEPALIVE is turned off.
0x80	SO_LINGER	Controls the actions taken when unsent messages are queued on a socket and the *sclose* (or the *DISCONNECT*) call is issued. If SO_LINGER is set, the connection is maintained until the outstanding messages have been sent.
~SO_LINGER	SO_DONTLINGER	Controls the actions of unsent messages. If SO_DONTLINGER is set, and the *sclose* (or the *DISCONNECT*) call is issued, any outstanding messages queued to be sent will be lost. The connection is then terminated.

The symbols are defined in the <socket.h> header file.

A.9 Socket Types

DECnet–VAXmate supports the following socket types:

Decimal Value	Type	Description
1	SOCK_STREAM	Stream sockets cause bytes to accumulate until internal DECnet buffers are full. The receiving task does not know how many bytes were sent in each write operation.
5	SOCK_SEQPACKET	Sequenced sockets cause bytes to be sent immediately. The receiving task receives those bytes in one "record".

The symbols are defined in the <socket.h> header file.

A.10 Defined Software Modules

The following software modules are supported by DECnet–VAXmate. They have defined three letter acronym (tla) strings.

Module Name	TLA String	Description
DNMOD_SES	SES	MS–NET to DECnet Session Interface
DNMOD_LAT	LAT	LAT driver
DNMOD_PDV	PDV	Port driver
DNMOD_SCH	SCH	Real-Time Scheduler
DNMOD_DLL	DLL	Data Link Layer
DNMOD_DNP	DNP	DECnet Network Process

The following interrupt vectors have been defined for these DECnet–VAXmate software modules:

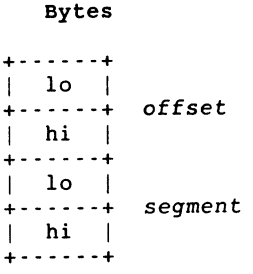
Vector Number (Hex)	Symbol	Description
0x2a	DNMODULE_SES	MS–NET to DECnet Session interface
0x6a	DNMODULE_LAT	LAT driver
0x6b	DNMODULE_PDV	Port Driver
0x6c	DNMODULE_SCH	Real-Time Scheduler
0x6d	DNMODULE_DLL	Data Link Layer
0x6e	DNMODULE_DNP	DECnet Network Process

The symbols are defined in the <dn.h> header file.

B
Defined Data Structures and Data Members

The following data structures can be used with specific socket interface and assembly language network driver interface calls. Guidelines for specifying a data structure are detailed with the appropriate function call. The symbols that appear in this appendix are defined in the DECnet header files.

If data type *exptr* is used as an address (or a long pointer), it takes the following format. It is defined in the <types.h> header file.



B.1 Access Control Information Data Structure

The *accessdata_dn* data structure contains the following data members:

Type	Size	Data Member	Description
unsigned short	2 bytes	*acc_accl*	Defines the length of the account string.
unsigned char	40-byte array	*acc_acc*	Specifies the account string.
unsigned short	2 bytes	*acc_passl*	Defines the length of the password string.
unsigned char	40-byte array	*acc_pass*	Specifies the password string.
unsigned short	2 bytes	*acc_userl*	Defines the length of the user ID string.
unsigned char	40-byte array	*acc_user*	Specifies the user ID string.

The symbols are defined in the <dn.h> header file.

B.2 Attach Data Structure

The *attach_dn* data structure contains the following data members:

Type	Size	Data Member	Description
int	2 bytes	*att_socket*	Specifies the number of the socket. If nonzero, the other data structure members are ignored.
unsigned short	2 bytes	*att_domain*	Specifies the communications domain for the socket as AF_DECnet.
unsigned short	2 bytes	*att_type*	Specifies the socket type for the socket. For example, SOCK_STREAM. (See Appendix A for a list of defined socket types.)
unsigned short	2 bytes	*att_protocol*	Specifies the protocol for the socket. For example, DNPROTO_NSP. (See Appendix A for a list of defined protocol interfaces.)
unsigned short	2 bytes	*att_srp*	Specifies the socket recovery period.
unsigned short	2 bytes	*att_supreq*	Specifies the support requirements.

The symbols are defined in the <dnmsdos.h> header file.

B.3 DECnet Node Address Data Structure

The *dn__naddr* data structure contains the following data members:

Type	Size	Data Member	Description
unsigned short	2 bytes	*a__len*	Specifies the length of the DECnet node address.
unsigned char	2-byte array	*a__addr[DN__MAXADDL]*	Specifies the DECnet Phase IV node address for the user task. When *a__addr[DN__MAXADDL]* is used as a 16-bit unsigned integer, bits 0–9 are the node number, and bits 10–15 are the area number.

The symbols are defined in the <dn.h> header file.

B.4 Listen Data Structure

The *listen__dn* data structure contains the following data member:

Type	Size	Data Member	Description
int	2 bytes	*lsn__backlog*	Defines the maximum number of unaccepted incoming connects which are allowed on this particular socket.

The symbol is defined in the <dnmsdos.h> header file.

B.5 Local Node Information Data Structure

The *localinfo_dn* data structure contains the following data members:

Type	Size	Data Member	Description
unsigned char	3-byte array	*lcl_version*	Specifies the software version number for the network process.
unsigned char	7-byte array	*lcl_nodename*	Specifies the node name for the local node. It is terminated by a null character.
unsigned short	2 bytes	*lcl_nodeaddr*	Specifies the DECnet Phase IV node address for the local node. The node address is formatted as a 16-bit unsigned integer, where bits 0–9 are the node number and bits 10–15 are the area number.
unsigned short	2 bytes	*lcl_segsize*	Specifies the minimum buffer segment size used on the logical link. This number should match the value defined with the NCP command, DEFINE EXECUTOR SEGMENT BUFFER SIZE. (Refer to the *DECnet–VAXmate User's Guide* for more details.)
unsigned char	1 byte	*lcl_sockets*	Specifies the number of sockets available for data exchange.
unsigned char	1 byte	*lcl_decnet_device*	Specifies the DECnet database device name.
exptr	4 bytes	*lcl_decnet_path*	Specifies the address of a buffer that contains the DECnet database path specification string which includes the device name.

The symbols are defined in the <dnmsdos.h> header file.

B.6 Logical Link Information Data Structure

The *linkinfo_dn* data structure contains the following data members:

Type	Size	Data Member	Description
unsigned short	2 bytes	*idn_segsize*	Specifies the buffer segment size in use on the logical link.
unsigned char	1 byte	*idn_linkstate*	Specifies the state of the logical link. (See Appendix A for a list of logical link states.)

The symbols are defined in the <dn.h> header file.

B.7 Optional User Data Structure

The *optdata_dn* data structure contains the following data members:

Type	Size	Data Member	Description
unsigned short	2 bytes	*opt_status*	Specifies an extended status value returned by function call. A list of the extended error codes appears in Appendix D.
unsigned short	2 bytes	*opt_optl*	Is the size of the optional user data.
unsigned char	16-byte array	*opt_data*	Specifies the optional user data.

The symbols are defined in the <dn.h> header file.

B.8 Select Data Structure

The *select_dn* data structure contains the following data members:

Type	Size	Data Member	Description
unsigned short	2 bytes	*sel_nfds*	Specifies the highest socket number to be checked.
field32	4 bytes	*sel_read*	Specifies the socket numbers to be examined for read ready or incoming connections.
field32	4 bytes	*sel_write*	Specifies the socket numbers to be examined for write ready.
field32	4 bytes	*sel_except*	Specifies the socket numbers to be examined for exception or out-of-band data ready.
unsigned short	2 bytes	*sel_seconds*	Specifies the time to wait for the socket selection to complete.

NOTE

The field32 data member is the same as unsigned long for type.

The symbols are defined in the <dnmsdos.h> header file.

B.9 Shutdown Data Structure

The *shutdown_dn* data structure contains the following data member:

Type	Size	Data Member	Description
int	2 bytes	*snd_how*	Specifies the type of shutdown. The argument can be set to: 0 which disallows further receives. 1 which disallows further sends. 2 which disallows further sends and receives.

The symbol is defined in the <dnmsdos.h> header file.

B.10 Socket Address Data Structure

The *sockaddr_dn* data structure contains the following data members:

Type	Size	Data Member	Description
unsigned short	2 bytes	*sdn_family*	Specifies the communications domain as AF_DECnet.
unsigned char	1 byte	*sdn_flags*	Specifies the object flag option. It **must** be set to zero, if not used.
unsigned char	1 byte	*sdn_objnum*	Defines the object number for the socket.
unsigned short	2 bytes	*sdn_objnamel*	Is the size of the node's object name.
char	16-byte array	*sdn_objname*	Defines the name of the network task.
struct	4 bytes	*sdn_add*	Specifies the node address data structure. (See the description of the *dn_naddr* data structure in this appendix.)

The symbols are defined in the <dn.h> header file.

B.11 Socket I/O Status Data Structure

The *sioctl_dn* data structure contains the following data members:

Type	Size	Data Member	Description
int	2 bytes	*sio_s*	This data member is not used by the *ATTACH* call.
int	2 bytes	*sio_request*	Specifies the I/O control level to be used. (See Section 4.4.14 for details.)
exptr	4 bytes	*argp*	Specifies the address of the argument list.

The symbols are defined in the <dnmsdos.h> header file.

B.12 Socket Option Data Structure

The *sockopt_dn* data structure contains the following data members:

Type	Size	Data Member	Description
int	2 bytes	*sop_level*	Specifies the layer at which options are manipulated. If the level is set to SOL_SOCKET, then the rest of the data structure is ignored. If the level is set to DNPROTO_NSP, then the rest of the data structure can contain either access control and/or optional data; or acceptance mode information.
int	2 bytes	*sop_optname*	Specifies options to be passed for interpretation. When the socket level is set to DNPROTO_NSP, *sop_optname* can be set to one of 7 specific options. For example, DSO_CONDATA. (See Appendix A for a list of the specific options.)
exptr	4 bytes	*sop_optval*	Specifies an address for the buffer which contains either access control or optional user data. (See Section 4.4.12 for the relationship between *sop_optname* and *sop_optval* arguments.) Specifies an address for the buffer which contains acceptance mode information.
exptr	4 bytes	*sop_optlen*	Specifies the size of the option value buffer used as a parameter for the *setsockopt* call. It is also a value result parameter for the *getsockopt* call.

The symbols are defined in the <dnmsdos.h> header file.

B.13 User Access Control Information Data Structure

The *dnet_accent* data structure contains the following data members:

Type	Size	Data Member	Description
char	1 byte	*acc_status*	Is used internally by this function call.
char	1 byte	*acc_type*	Specifies the type of privilege associated with the user name or password. The four access types are: 0 for no access rights, 1 for read only access, 2 for write only access, and 3 for read and write access.
char	40-byte array	*acc_user*	Specifies the user name. It consists of a 1- to 39-alphabetic character string terminated by a null character.
char	40-byte array	*acc_pass*	Specifies the password associated with a user name. It consists of a 1- to 39-alphabetic character string terminated by a null character.

These symbols are defined in the <dnetdb.h> header file.

B.14 User Defined Callback Routine Data Structure

The *io_callback* member of the CIOCB has the following format:

Type	Size	Data Member	Description
exptr	4 bytes	*io_callback*	Specifies the address for the callback routine which will be returned when a function call completes.

You should refer to Chapter 6 of this manual for more details on the CIOCB data structure.

B.15 User Defined Data Buffer Structure

The *io_buffer* member of the IOCB(CIOCB) data structure has the following format:

Type	Size	Data Member	Description
exptr	4 bytes	*io_buffer*	Specifies the address for the buffer which contains user defined data.

You should refer to Chapter 6 of this manual for more details on the IOCB and CIOCB data structures.

C
Summary of Error Completion Codes

This appendix lists the error completion codes returned by DECnet–VAXmate in *errno*. They provide extended error information to transparent file access, transparent task-to-task operations, and nontransparent task-to-task communication.

These error codes are a subset of the error codes contained in the external variable *errno*. The following descriptions are standard ULTRIX definitions. You should refer to specific DECnet–VAXmate calls for a network definition of the error codes.

Mnemonic	Decimal Value	Description
ESRCH	3	No such process
E2BIG	7	Argument list too long
EBADF	9	Bad file number
EACCES	13	Permission was denied
EFAULT	14	Bad address
EBUSY	16	Mount device busy
EEXIST	17	File exists
EINVAL	22	Invalid argument
EMFILE	24	Too many open files
ENOSPC	28	No space left on device
EPIPE	32	Broken pipe

(continued on next page)

Mnemonic	Decimal Value	Description
Math software		
EDOM	33	Argument too large
ERANGE	34	Result too large
Nonblocking and interrupt I/O		
EWOULDBLOCK	35	Operation would block
EINPROGRESS	36	Operation now in progress
EALREADY	37	Operation already in progress
Argument errors		
ENOTSOCK	38	Socket operation on nonsocket
EDESTADDRREQ	39	Destination address required
EMSGSIZE	40	Message too long
ENOPROTOOPT	42	Protocol not available
EPROTONOSUPPORT	43	Protocol not supported
ESOCKTNOSUPPORT	44	Socket type not supported
EOPNOTSUPP	45	Operation not supported on socket
EAFNOSUPPORT	47	Address family not supported by protocol family
EADDRINUSE	48	Address already in use
EADDNOTAVAIL	49	Cannot assign requested address
Operational errors		
ENETDOWN	50	Network is down
ENETUNREACH	51	Network is unreachable
ECONNABORTED	53	Software caused connection abort
ECONNRESET	54	Connection reset by peer
ENOBUFS	55	No buffer space available
EISCONN	56	Socket is already connected
ENOTCONN	57	Socket is not connected
ETOOMANYREFS	59	Too many references: cannot splice
ETIMEDOUT	60	Connection timed out

(continued on next page)

Mnemonic	Decimal Value	Description
Operational errors (cont.)		
ECONNREFUSED	61	Connection refused
ENAMETOOLONG	63	File name too long
EHOSTDOWN	64	Host is down
EHOSTUNREACH	65	No route to host

D
Summary of Extended Error Codes

DECnet–VAXmate supports extended error support to certain socket operations. When you write a program which uses the *getsockopt* function call, extended error codes can be returned in *opt_status*, a data member of *optdata_dn*. This can occur following an attempted connection request or after disconnecting a logical link.

Table D–1 lists extended error codes which can be returned following an attempted connection. It lists the error messages found in *derrno.h*, the decimal value for each message, their equivalent error message that *dnet_conn* returns in *errno*, and the error reason.

Table D–1: Extended Error Messages – Unable to Make a Connection

Decimal Error Code	derrno.h Mnemonic	dnet_conn In errno	Reason
0	EREJBYOBJ	ECONNREFUSED	Connect failed. Connection rejected by object.
1	EINSSNETRES	ENOSPC	Connect failed. Insufficient network resources.
2	EUNRNODNAM	EADDRNOTAVAIL	Connect failed. Unrecognized node name.

(continued on next page)

Table D–1 (cont.): Extended Error Messages – Unable to Make a Connection

Decimal Error Code	derrno.h Mnemonic	dnet_conn In errno	Reason
3	EREMNODESHUT	ENETDOWN	Connect failed. Remote node shutting down.
4	EUNROBJ	ESRCH	Connect failed. Unrecognized object.
5	EINVOBJNAM	EINVAL	Connect failed. Invalid object name format.
6	EOBJBUSY	ETOOMANYREFS	Connect failed. Object too busy.
10	EINVNODNAM	ENAMETOOLONG	Connect failed. Invalid node name format.
11	ELOCNODESHUT	EHOSTDOWN	Connect failed. Local node shutting down.
32	ENODERESOURCES	ENOSPC	Connect failed. No node resources for new logical link.
33	EUSERESOURCES	ENOSPC	Connect failed. No user resources for new logical link.
34	EACCONREJ	ECONNABORTED	Connect failed. Access control rejected.
36	EBADACCOUNT	ECONNABORTED	Connect failed. Bad account information.
38	ENORESPOBJ	ETIMEDOUT	Connect failed. No response from object.
39	ENODUNREACH	ENETUNREACH	Connect failed. Node unreachable.
43	ECONNTOOBIG	ECONNABORTED	Connect failed. Connect image data field too long.

Table D–2 lists extended error codes which can be returned following a disconnection. It lists the error messages found in *derrno.h*, the decimal value for each message and the error reason.

Table D–2: Extended Error Messages – Disconnecting a Logical Link

Decimal Error Code	derrno.h Mnemonic	Reason
0	EREJBYOBJ	The end user disconnected a running logical link.
8	EABTBYNMGT	The logical link was disconnected by a third party.
9	EUSERABORT	The remote end user has aborted the link.
38	ENORESPOBJ	The end user or node at the other end of the link has crashed or failed.
39	ENODUNREACH	The connection has been lost due to a local timeout.
41	ENOLINK	The connection has been lost due to a protocol failure, no such link found at remote.
42	ECOMPLETE	No error, a local end user initiated disconnection has completed.

E
Data Access Protocol (DAP) Error Messages

The Network Task Error log utility provides extended error support to transparent file access operations. This appendix lists DAP error messages that can be returned by this utility. The Network File Transfer utility may also return some of these error messages.

E.1 Overview

The DAP messages return status from the remote file system or from the operation of the cooperating process using DAP. The 2-byte status field (16 bits) is divided into two fields:

- Maccode (bits 12–15): Contains the error type code (see Table E–1 in Section E.1.1).
- Miccode (bits 0–11): Contains the specified error reason code (see Tables E–2, E–3, and E–4, depending on error type, as described in Section E.1.2).

E.1.1 Maccode Field

The value returned in the maccode field describes the functional type of the error that has occurred. The specific reason for the error is given in the miccode field. Miccode values correlating to each maccode value listed in Table E–1 are found in the table referenced in the last column of Table E–1.

Table E–1: DAP Maccode Field Values

Field Value (Octal)	Error Type	Meaning	Miccode Table
0	Pending	The operation is in progress.	E–3
1	Successful	Returns information that indicates success.	E–3
2	Unsupported	This implementation of DAP does not support the specified request.	E–2
3	Reserved		
4	File open	Errors that occur before a file is successfully opened.	E–3
5	Transfer error	Errors that occur after a file is opened and before it is closed.	E–3
6	Transfer warning	For operations on open files, indicates that the operation completed, but not with complete success.	E–3
7	Access termination	Errors associated with terminating access to a file.	E–3
10	Format	Error in parsing a message. Format is not correct.	E–2
11	Invalid	Field of message is invalid (that is, bits that are meant to be mutually exclusive are set, an undefined bit is set, a field value is out of range, or an illegal string is in a field.)	E–2
12	Sync	DAP message received out of synchronization.	E–4
13–15	Reserved		
16–17	User-defined status maccodes		

E.1.2 Miccode Field

The value returned in this field identifies the specific reason for the error type defined in the maccode field (see Section E.1.1). Miccode field values are defined in three different tables, each table associated with certain maccode values, as outlined below:

- Table E–2: For use with maccode values 2, 10, 11
- Table E–3: For use with maccode values 0, 1, 4, 5, 6, 7
- Table E–4: For use with maccode value 12

Table E–2 follows. The DAP message type number (column 1) is specified in bits 6–11, and the DAP message field number (column 2) is specified in bits 0–5. The field where the error is located is described in the third column.

Table E–2: DAP Miccode Values for Use with Maccode Values of 2, 10, 11

Type Number (bits 6–11)	Field Number (bits 0–5)	Field Description
Miscellaneous message errors		
00	00	Unspecified DAP message error
	10	DAP message type field (TYPE) error
Configuration message errors		
01	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	BITCNT field (BITCNT)
	20	Buffer size field (BUFSIZ)
	21	Operating system type field (OSTYPE)
	22	File system type field (FILESYS)
	23	DAP version number (VERNUM)
	24	ECO version number field (ECONUM)
	25	USER protocol version number field (USRNUM)
	26	DEC software release number field (DECVER)
	27	User software release number field (USRVER)
	30	System capabilities field (SYSCAP)

(continued on next page)

Table E–2 (cont.): DAP Miccode Values for Use with Maccode Values of 2, 10, 11

Type Number (bits 6–11)	Field Number (bits 0–5)	Field Description
Attributes message errors		
02	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN 256)
	14	Bit count field (BITCNT)
	20	Attributes menu field (ATTMENU)
	21	Data type field (DATATYPE)
	22	Field organization field (ORG)
	23	Record format field (RFM)
	24	Record attributes field (RAT)
	25	Block size field (BLS)
	26	Maximum record size field (MRS)
	27	Allocation quantity field (ALQ)
	30	Bucket size field (BKS)
	31	Fixed control area size field (FSZ)
	32	Maximum record number field (MRN)
	33	Run-time system field (RUNSYS)
	34	Default extension quantity field (DEQ)
	35	File options field (FOP)
	36	Byte size field (BSZ)
	37	Device characteristics field (DEV)
	40	Spooling device characteristics field (SDC); reserved
	41	Longest record length field (LRL)
	42	Highest virtual block allocated field (HBK)
	43	End-of-file block field (EBK)
	44	First free byte field (FFB)
	45	Starting LBN for contiguous file field (SBN)

(continued on next page)

Table E–2 (cont.): DAP Miccode Values for Use with Maccode Values of 2, 10, 11

Type Number (bits 6–11)	Field Number (bits 0–5)	Field Description
Access message errors		
03	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Access function field (ACCFUNC)
	21	Access options field (ACCOPT)
	22	File specification field (FILESPEC)
	23	File access field (FAC)
	24	File-sharing field (SHR)
	25	Display attributes request field (DISPLAY)
	26	File password field (PASSWORD)
Control message errors		
04	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Control function field (CTLFUNC)
	21	Control menu field (CTLMENU)
	22	Record access field (RAC)
	23	Key field (KEY)
	24	Key of reference field (KRF)
	25	Record options field (ROP)
	26	Hash code field (HSH); reserved for future use
	27	Display attributes request field (DISPLAY)
	30	Block count (BLKCNT)

(continued on next page)

Table E–2 (cont.): DAP Miccode Values for Use with Maccode Values of 2, 10, 11

Type Number (bits 6–11)	Field Number (bits 0–5)	Field Description
Continue message errors		
05	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Continue transfer function field (CONFUNC)
Acknowledge message errors		
06	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	15	System-specific field (SYSPEC)
Access complete message errors		
07	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Access complete function field (CMPFUNC)
	21	File options field (FOP)
	22	Checksum field (CHECK)

(continued on next page)

Table E–2 (cont.): DAP Miccode Values for Use with Maccode Values of 2, 10, 11

Type Number (bits 6–11)	Field Number (bits 0–5)	Field Description
Key definition message errors		
12	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Key definition menu field (KEYMENU)
	21	Key option flags field (FLG)
	22	Data bucket fill quantity field (DFL)
	23	Index bucket fill quantity field (IFL)
	24	Key segment repeat count field (SEGCNT)
	25	Key segment position field (POS)
	26	Key segment size field (SIZ)
	27	Key of reference field (REF)
	30	Key name field (KNM)
	31	Null key character field (NUL)
	32	Index area number field (IAN)
	33	Lowest level area number field (LAN)
	34	Data level area number field (DAN)
	35	Key data type field (DTP)
	36	Root VBN for this key field (RVB)
	37	Hash algorithm value field (HAL)
	40	First data bucket VBN field (DVB)
	41	Data bucket size field (DBS)
	42	Index bucket size field (IBS)
	43	Level of root bucket field (LVL)
	44	Total key size field (TKS)
	45	Minimum record size field (MRL)

(continued on next page)

Table E–2 (cont.): DAP Miccode Values for Use with Maccode Values of 2, 10, 11

Type Number (bits 6–11)	Field Number (bits 0–5)	Field Description
Allocation message errors		
13	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Allocation menu field (ALLMENU)
	21	Relative volume number field (VOL)
	22	Alignment options field (ALN)
	23	Allocation options field (AOP)
	24	Starting location field (LOC)
	25	Related file identification field (RFI)
	26	Allocation quantity field (ALQ)
	27	Area identification field (AID)
	30	Bucket size field (BKZ)
	31	Default extension quantity field (DEQ)
Summary message errors		
14	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Summary menu field (SUMENU)
	21	Number of keys field (NOK)
	22	Number of areas field (NOA)
	23	Number of record descriptors field (NOR)
	24	Prologue version number (PVN)

(continued on next page)

Table E–2 (cont.): DAP Miccode Values for Use with Maccode Values of 2, 10, 11

Type Number (bits 6–11)	Field Number (bits 0–5)	Field Description
Date and time message errors		
15	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Date and time menu field (DATMENU)
	21	Creation date and time field (CDT)
	22	Last update date and time field (RDT)
	23	Deletion date and time field (EDT)
	24	Revision number field (RVN)
	25	Backup date and time field (BDT)
	26	Physical creation date and time field (PDT)
	27	Accessed date and time field (ADT)
Protection message errors		
16	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Protection menu field (PROTMENU)
	21	File owner field (OWNER)
	22	System protection field (PROTSYS)
	23	Owner protection field (PROTOWN)
	24	Group protection field (PROTGRP)
	25	World protection field (PROWLD)
Name message errors		
17	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Name type field (NAMETYPE)
	21	Name field (NAMESPEC)

(continued on next page)

Table E–2 (cont.): DAP Miccode Values for Use with Maccode Values of 2, 10, 11

Type Number (bits 6–11)	Field Number (bits 0–5)	Field Description
Access control list message errors (reserved for future use)		
20	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	15	System-specific field (SYSPEC)
	20	Access control list repeat count field (ACLCNT)
	21	Access control list entry field (ACL)

Table E–3 follows. The error code number (column 1) is contained in bits 0–11. For corresponding RMS or FCS status codes, refer to the appropriate DECnet or RMS documentation for each remote system.

Table E–3: DAP Miccode Values for Use with Maccode Values 0, 1, 4, 5; 6, 7

Error Code (bits 0–11)	Error Description
0	Unspecified error
1	Operation aborted
2	F11–ACP could not access file
3	File activity precludes operation
4	Bad area ID
5	Alignment options error
6	Allocation quantity too large or 0 value
7	Not ANSI D format
10	Allocation options error
11	Invalid (that is, synchronous) operation at AST level
12	Attribute read error
13	Attribute write error
14	Bucket size too large

(continued on next page)

Table E–3 (cont.): DAP Miccode Values for Use with Maccode Values 0, 1, 4, 5, 6, 7

Error Code (bits 0–11)	Error Description
15	Bucket size too large
16	BLN length error
17	Beginning of file detected
20	Private pool address
21	Private pool size
22	Internal RMS error condition detected
23	Cannot connect RAB
24	$UPDATE changed a key without having attribute of XB$CHG set
25	Bucket format check-byte failure
26	RSTS/E close function failed
27	Invalid or unsupported COD field
30	F11–ACP could not create file (STV – system error code)
31	No current record (operation not preceded by get/find)
32	F11–ACP deaccess error during close
33	Data area number invalid
34	RFA-accessed record was deleted
35	Bad device, or inappropriate device type
36	Error in directory name
37	Dynamic memory exhausted
40	Directory not found
41	Device not ready
42	Device has positioning error
43	DTP field invalid
44	Duplicate key detected; XB$DUP not set
45	F11–ACP enter function failed
46	Operation not selected in ORG$ macro

(continued on next page)

Table E–3 (cont.): DAP Miccode Values for Use with Maccode Values 0, 1, 4, 5, 6, 7

Error Code (bits 0–11)	Error Description
47	End of file
50	Expanded string area too short
51	File expiration date not yet reached
52	File extend failure
53	Not a valid FAB (BID does not = FB$BID)
54	Illegal FAC for record operation, or FB$PUT not set for create
55	File already exists
56	Invalid file ID
57	Invalid flag-bits combination
60	File is locked by other user
61	F11–ACP find function failed
62	File not found
63	Error in file name
64	Invalid file options
65	Device/file full
66	Index area number invalid
67	Invalid IFI value or unopened file
70	Maximum NUM (254) areas/key XABS exceeded
71	$INIT macro never issued
72	Operation illegal or invalid for file organization
73	Illegal record encountered (with sequential files only)
74	Invalid ISI value on unconnected RAB
75	Bad key buffer address (KBF = 0)
76	Invalid key field (KEY = 0 or negative)
77	Invalid key of reference ($GET/$FIND)
100	Key size too large

(continued on next page)

Table E–3 (cont.): DAP Miccode Values for Use with Maccode Values 0, 1, 4, 5, 6, 7

Error Code (bits 0–11)	Error Description
101	Lowest level index area number invalid
102	Not ANSI-labeled tape
103	Logical channel busy
104	Logical channel number too large
105	Logical extend error; prior extend still valid
106	LOC field invalid
107	Buffer-mapping error
110	F11–ACP could not mark file for deletion
111	MRN value = negative or relative key > MRN
112	MRS value = 0 for fixed length records and/or relative files
113	NAM block address invalid (NAM = 0 or is not accessible)
114	Not positioned to EOF (with sequential files only)
115	Cannot allocate internal index descriptor
116	Indexed file; primary key defined
117	RSTS/E open function failed
120	XABs not in correct order
121	Invalid file organization value
122	Error in file's prologue (reconstruct file)
123	POS field invalid (POS > MRS; STV = XAB indicator)
124	Bad file date field retrieved
125	Privilege violation (OS denies access)
126	Not a valid RAB (BID does not = RB$BID)
127	Illegal RAC value
130	Illegal record attributes
131	Invalid record buffer address (either odd or not word aligned if BLK–IO)

(continued on next page)

Table E–3 (cont.): DAP Miccode Values for Use with Maccode Values 0, 1, 4, 5, 6, 7

Error Code (bits 0–11)	Error Description
132	File read error
133	Record already exists
134	Bad RFA value (RFA = 0)
135	Invalid record format
136	Target bucket locked by another stream
137	F11–ACP remove function failed
140	Record not found
141	Record not locked
142	Invalid record options
143	Error while reading prologue
144	Invalid RRV record encountered
145	RAB stream currently active
146	Bad record size (RSZ > MRS or NOT = MRS if fixed length records)
147	Record too big for user's buffer
150	Primary key out of sequence (RAC = RB$SEQ for $PUT)
151	SHR field invalid for file (cannot share sequential files)
152	SIZ field invalid
153	Stack too big for save area
154	System directive error
155	Index tree error
156	Error in file type extension on FNS is too big
157	Invalid user buffer address (0, odd, or not word aligned if BLK–IO)
160	Invalid user buffer size (USZ = 0)
161	Error in version number
162	Invalid volume number

(continued on next page)

Table E–3 (cont.): DAP Miccode Values for Use with Maccode Values 0, 1, 4, 5, 6, 7

Error Code (bits 0–11)	Error Description
163	File write error (STV = system error code)
164	Device is write locked
165	Error while writing prologue
166	Not a valid XAB (@XAB = odd; STV = XAB indicator)
167	Default directory invalid
170	Cannot access argument list
171	Cannot close file
172	Cannot deliver AST
173	Channel assignment failure (STV = system error code)
174	Terminal output ignored due to CTRL/O
175	Terminal input aborted due to CTRL/Y
176	Default file name string address error
177	Invalid device ID field
200	Expanded string address error
201	File name string address error
202	FSZ field invalid
203	Invalid argument list
204	Known file found
205	Logical name error
206	Node name error
207	Operation successful
210	Inserted record had duplicate key
211	Index update error occurred; record inserted
212	Record locked, but read anyway
213	Record inserted in primary key is okay; may not be accessible by secondary keys or RFA

(continued on next page)

Table E–3 (cont): DAP Miccode Values for Use with Maccode Values 0, 1, 4, 5, 6, 7

Error Code (bits 0–11)	Error Description
214	File was created, but not opened
215	Bad prompt buffer address
216	Asynchronous operation pending completion
217	Quoted string error
220	Record header buffer invalid
221	Invalid related file
222	Invalid resultant string size
223	Invalid resultant string address
224	Operation not sequential
225	Operation successful
226	Created file superseded existing version
227	File name syntax error
230	Timeout period expired
231	FB$BLK record attribute not supported
232	Bad byte size
233	Cannot disconnect RAB
234	Cannot get JFN for file
235	Cannot open file
236	Bad JFN value
237	Cannot position to end of file
240	Cannot truncate file
241	File currently in an undefined state; access is denied
242	File must be opened for exclusive access
243	Directory full
244	Handler not in system

(continued on next page)

Table E–3 (cont.): DAP Miccode Values for Use with Maccode Values 0, 1, 4, 5, 6, 7

Error Code (bits 0–11)	Error Description
245	Fatal hardware error
246	Attempt to write beyond EOF
247	Hardware option not present
250	Device not attached
251	Device already attached
252	Device not attachable
253	Shareable resource in use
254	Illegal overlay request
255	Block check or CRC error
256	Caller's nodes exhausted
257	Index file full
260	File header full
261	Accessed for write
262	File header checksum failure
263	Attribute control list error
264	File already accessed on LUN
265	Bad tape format
266	Illegal operation on file descriptor block
267	Rename; two different devices
270	Rename; new file name already in use
271	Cannot rename old file system
272	File already open
273	Parity error on device
274	End of volume detected
275	Data overrun

(continued on next page)

Table E–3 (cont.): DAP Miccode Values for Use with Maccode Values 0, 1, 4, 5, 6, 7

Error Code (bits 0–11)	Error Description
276	Bad block on device
277	End of tape detected
300	No buffer space for file
301	File exceeds allocated space; no blocks left
302	Specified task not installed
303	Unlock error
304	No file accessed on LUN
305	Send/receive failure
306	Spool or submit command file failure
307	No more files
310	DAP file transfer checksum error
311	Quota exceeded
312	Internal network error condition detected
313	Terminal input aborted due to CTRL/C
314	Data bucket fill size > bucket size in XAB
315	Invalid expanded string length
316	Illegal bucket format
317	Bucket size of LAN does not = IAN in XAB
320	Index not initialized
321	Illegal file attributes (corrupt file header)
322	Index bucket fill size > bucket size in XAB
323	Key name buffer not readable or writeable in XAB
324	Index bucket will not hold two keys for key of reference
325	Multibuffer count invalid (negative value)
326	Network operation failed at remote node

(continued on next page)

Table E–3 (cont.): DAP Miccode Values for Use with Maccode Values 0, 1, 4, 5, 6, 7

Error Code (bits 0–11)	Error Description
327	Record is already locked
330	Deleted record successfully accessed
331	Retrieved record exceeds specified key value
332	Key XAB not filled in
333	Nonexistent record successfully accessed
334	Unsupported prologue version
335	Illegal key of reference in XAB
336	Invalid resultant string length
337	Error updating RRVs; some paths to data may be lost
340	Data types other than string limited to one segment in XAB
341	Reserved
342	Operation not supported over network
343	Error on write behind
344	Invalid wildcard operation
345	Working set full (cannot lock buffers in working set)
346	Directory listing: error in reading volume set name, directory name, or file name
347	Directory listing: error in reading file attributes
350	Directory listing: protection violation in trying to read the volume set, directory, or file name
351	Directory listing: protection violation in trying to read file attributes
352	Directory listing: file attributes do not exist
353	Directory listing: unable to recover directory list after continue transfer (skip)
354	Sharing not enabled
355	Sharing page count exceeded

(continued on next page)

Table E–3 (cont.): DAP Miccode Values for Use with Maccode Values 0, 1, 4, 5, 6, 7

Error Code (bits 0–11)	Error Description
356	UPI bit not set when sharing with BRO set
357	Error in access control string
360	Terminator not seen
361	Bad escape sequence
362	Partial escape sequence
363	Invalid wildcard context value
364	Invalid directory rename operation
365	User structure (FAB/RAB) became invalid during operation
366	Network file transfer mode precludes operation
6000 to 7777	User-defined errors

Table E–4 follows. The message type number is contained in bits 0–11.

Table E–4: DAP Miccode Values for Use with Maccode Value 12

Type Number (bits 0–11)	Message Type
0	Unknown message type
1	Configuration message
2	Attributes message
3	Access message
4	Control message
5	Continue transfer message
6	Acknowledge message
7	Access complete message
10	Data message
11	Status message
12	Key definition attributes extension message
13	Allocation attributes extension message
14	Summary attributes extension message
15	Date and time attributes extension message
16	Protection attributes extension message
17	Name message
20	Access control list extended attributes message

F
Transparent File Access Error Messages

This appendix summarizes extended error messages for transparent file access operations. The following messages are displayed by the TNT utility.

Extended Error Message

Cannot get a handle to the network driver.

Too many logical links already in use. The maximum number is 4.

Error in node specification.

The node name specification was not found in DECPARM.DAT.

Unable to transmit user buffer.

Invalid DAP message type received.

Unsupported DAP flag field received.

Invalid DAP message format received.

Unexpected DAP message received.

Unsupported DAP data type.

Unsupported file organization. DECnet–VAXmate supports sequential file organization.

Remote system DAP buffer size is less than 128 bytes.

The file to be accessed is not open.

Error – unknown error.

The maximum record size has exceeded 128 bytes.

The buffer size for the records contained in the remote input file is too small.

Error in closing file.

G
Transporting DECnet–VAXmate Programs

If you develop code to be transported to a DECnet–ULTRIX or any other system that supports the socket interface library, it is recommended that you use these suggestions:

- The *select* function has a feature specific only to DECnet–VAXmate. The bit mask *exceptfds* is presently not supported by DECnet–ULTRIX. As a result, you cannot transport code that includes the exception bit mask.
- Include a special prefix and compatibility mode header file in your DECnet–VAXmate program.
- Define certain function call names depending upon which system compilation is to take place. For DECnet–VAXmate programs, the socket function calls – *ioctl*, *read*, *write* and *close* must be prefixed with an "s".

An example compatibility header file is shown below:

```
#ifdef MSDOS

#define ioctl(s, f, a)      sioctl(s, f, a)       /* control    */
                                                  /* socket i/o*/

#define read(s, buf, len)   sread(s, buf, len)    /* read from */
                                                  /* a socket  */
#define write(s, buf, len)  swrite(s, buf, len)   /* write to  */
                                                  /* a socket  */
#define close(s)            sclose(s)             /* close a   */
                                                  /* socket    */
#endif                                            /* MSDOS     */
```

H
DECnet–VAXmate Programming Examples

H.1 Example Client Task Program

The following networking program uses the DECnet–VAXmate socket interface. In this example, the client task tries to connect to a task on a remote node;tries to send data and waits 30 seconds for any incoming data before timing out or until the peer task decides to close down the link.

```
/*
 * Include standard headers.
 */
#include <stdio.h>

/*
 * User defined symbols for conditional compilation.
 */
#include "dnprefix.h"

/*
 * Include some network interface headers.
 */
#include "types.h"  /* Type definitions, abstract data types.
*/
#include "time.h"   /* Time data structures.
*/
#include "dn.h"     /* Network data structures and
*/
                    /* definitions.
*/
*/
#include "socket.h" /* Socket interface layer definitions.
*/
#include "sioctl.h" /* Socket I/O control functions.
*/
```

(continued on next page)

```
#include "errno.h"  /* Global user error definitions returned
*/
                    /* in 'errno'.
*/

/*
 * Conditionalize for DECnet-ULTRIX compatibility.
 */
#ifndef MSDOS
#define sclose(s) close(s)
#define sioctl(s,f,a) ioctl(s,f,a)
#endif

#define SEQUENCED_PACKET 0
#define STREAM           1

/*
 * Version string.
 */
static char      version[] = "V1.01";

/*
 * Main line code.
 */
main(argc, argv)

int     argc;
char    *argv[];

{
   /*
    * Local data.
    */
    struct  timeval tmv;
    char    *node, *object;
    u_char  optional_send[16];
    u_char  optional_receive[16];
    u_char  data_buffer[10];
    field32 readfds, writefds;
    int     rec_len;
    int     sock_type;
    int     sock;
    int     loop;
    int     count;
    int     len;
    int     ind0;
    char    bio[1];

    /*
     * Make sure there are a valid number of input arguments.
     */
    if (argc < 3)
    {
        printf("Usage: test <node name or address>\
 <#objnum or objnam>\n");
        exit(1);

    }
```

(continued on next page)

```
    /*
     * Display our current version.
     */
     printf("\t\tSample - %s\n", &version [0]);

    /*
     * Set up optional data to send with connect.
     */
    strcpy(&optional_send[0], "hello");

    /*
     * Attempt to connect to the object on the remote node.
     */
    rec_len = sizeof(optional_receive);
    node = *++argv;
    object = *++argv;
    sock_type = SEQUENCED_PACKET;
    printf("connecting to node \"%s\", object \"%s\"\n",
          node, object);
    if ((sock = dnet_conn(node, object, sock_type,
         &optional_send[0],
               strlen("hello"),
               &optional_receive[0], &rec_len)) < 0)
    {
      nerror("dnet_conn");
      exit(1);
    }
    printf("connect complete with node \"%s\",\
object \"%s\"\n", node, object);

    /*
     * Check for returned optional data.
     */
    if (rec_len)
    {
        puts("optional data received:");
        for (ind0 = 0; ind0 < rec_len; ind0++)
        {
            printf(" %d", optional_receive[ind0]);
        }
        puts("");
     }

    /*
     * Fill a data buffer with dummy data.
     */
    for (loop = 0; loop < sizeof(data_buffer); loop++)
    {
        data_buffer[loop] = loop;
    }

    /*
     * Try to send a dummy data buffer 10 times
     * to target object as long as link is still active.
     */
    loop = 10;
    while (loop--)
```

(continued on next page)

```
{
    if (dnet_eof(sock) == 1)
    {
        printf("link is down.\n");
        sclose(sock);
        exit(1);
    }

    if ((count = send(sock, &data_buffer[0],
                      sizeof(data_buffer), 0)) < 0)
    {
        perror("write");
        sclose(sock);
        exit(1);
    }
}
printf("data successfully sent to %s\n", node);

/*
 * Now set the socket to nonblocking mode.
 */

bio[0] = 1;
sioctl(sock, FIONBIO, &bio[0]);

/*
 * Clean out the data buffer.
 */
bzero(&data_buffer[0], sizeof(data_buffer));

/*
 * Continue to receive data from target object until
 * disconnected.
 */
while (1)

{
    /*
     * Check if link is still active.
     */
    if (dnet_eof(sock) == 1)
    {
        printf("link is down.\n");
        sclose(sock);
        exit(1);
    }

    /*
     * Now check to see if the socket has data available
     * to read and timeout after 30 seconds.
     */
    readfds = 1 << sock;
    tmv.tv_sec = 30;
    if ((ind0 = select(sock + 1, &readfds, 0, 0, &tmv)) < 0)
    {
            perror("select");
    }
    else
    {
```

(continued on next page)

```
            if (ind0 == 0)
            {
                printf("receive wait timed out.\n");
                sclose(sock);
                exit(1);
            }

        }

        if ((count = recv(sock, &data_buffer[0],
                          sizeof(data_buffer), 0)) < 0)
        {
            if (errno != EWOULDBLOCK)
            {
                perror("read");
                break;
            }
            else
                continue;
        }
        printf("data received (%d bytes):\n", count);
        for (ind0 = 0; ind0 < count; ind0++)
        {
             printf(" %d", data_buffer[ind0] );
        }
        puts("");
    }

    /*
     * Finish up. Make the socket linger on close to allow
     * things to get cleaned.
     */
    if (setsockopt(sock, SOL_SOCKET, SO_LINGER, 0, 0) < 0)
    {
        perror("setsockopt");
    }

    /*
     * Close the socket and exit program.
     */
    sclose(sock);
    exit(0);
}
```

H.2 Example Client Transparent Task-to-Task Program

The following program illustrates transparent task-to-task communication. It describes the functions that the client task uses to communicate over the network.

```
/*
 *
 * Sample client program written in C that shows Transparent
 * Task-to-Task using DECnet-VAXmate.
 *
 * When running this program, the command line argument should
 * look like a network task specification. See the following
 * examples as well as examples cited in the documentation:
 * For example:
 *
 *      \\t\pcdos\\#100(to connect by object number)
 *      \\t\pcdos\smith\xxxxx\\TIMESRV(to connect by object name)
 *
 * After getting a handle (for example, by connecting to a
 * remote object), an attempt is made to write/send some data to
 * the object and then close the handle.
 *
 * o    All C include files and external functions are
 *      distributed with the DECnet-VAXmate kit in the file
 *      DNETLIB.SRC.
 *
 * o    When attempts to run this program fail, run the utility
 *      TNT.EXE shipped with the DECnet-VAXmate kit to examine
 *      DECnet errors.
 *
 */
#include        <stdio.h>
#include        "types.h"
#include        "scbdef.h"
#include        "errno.h"

static char buf[100];

/*
 * Function(s) included in DNETLIB.SRC
 */
extern int hopen();
extern int hwrite(), hclose();


main(argc, argv)

int argc;

char *argv[];

{
    int i;
    int j;
    int len;
    int h = 0;
```

(continued on next page)

```
if (argc < 2)
{
    printf("Usage: ttttst <TTT_network_task_string>\n");
    printf("\n example:\n");
    printf("\t ttttst \\\\t\\pygmy\\\\<#object_number>\n");
    printf("\t or\n");
    printf("\t ttttst \\\\t\\pygmy\\\\<object_name>\n");
    exit(1);
}

/*
 * Fill a dummy data buffer.
 */
for (i = 0; i < sizeof(buf); i++)
     buf[i] = i;

/*
 * Open file (access remote network object).
 */
h = hopen(argv[1], SCBC_HOPEN);
if (h == ERROR)
{
    perror("\nopen");
    printf("\n (run TNT.EXE to examine network error)");
    exit(1);
}
printf("\nopen succeeded  handle: %u (connected to object)",
        h);

/*
 * Write to file (send data to remote object).
 */
if (hwrite(h, &buf[0], sizeof(buf)) != sizeof(buf))
{
    perror("\nwrite");
    printf("\n (run TNT.EXE to examine network error)");
}
else
    printf("\nwrite succeeded (sent data to object)");

/*
 * Read from file handle (receive data from object, if any).
 */
len = hread(h, &buf[0], sizeof(buf));
if (len < 0)
{
    perror("\nread");
    printf("\n (run TNT.EXE to examine network error)");
}
else
{
    printf("\nread %u byte(s) (received from object)\n",
            len);
    for (i = j = 0; i < len; i++, j++)
    {
        if (j > 9)
        {
            printf("\n");
            j = 0;
```

(continued on next page)

```
            }
            printf(" %4u", buf[i]);
        }
    }
    /*
     * Close file handle (disconnect link).
     */
    hclose(h);

    printf("\nfinished.");
    exit(0);
}
```

H.3 Example Server Task Program

The following networking program uses the DECnet–VAXmate socket interface. The example describes the activities of a DECnet–VAXmate server task.

```
/*
 * Program - MIRROR
 *
 * Copyright (C) 1985, All Rights Reserved, by
 * Digital Equipment Corporation, Maynard, Mass.
 *
 * This software is furnished under a license and may be used
 * and copied only in accordance with the terms of such license
 * and with the inclusion of the above copyright notice. This
 * software or any other copies thereof may not be provided or
 * otherwise made available to any other person. No title to
 * and ownership of the software is hereby transferred.
 *
 * The information in this software is subject to change without
 * notice and should not be construed as a commitment by
 * Digital Equipment Corporation.
 *
 * Digital assumes no responsibility for the use or reliability
 * of its software on equipment which is not supplied
 * by Digital.
 *
 *
 * MODULE DESCRIPTION:
 *
 * Program MIRROR
 *
 *  DECnet-VAXmate, mirror server, DECnet object 25
 *
 * Networks & Communications Software Engineering
 *
 * IDENT HISTORY:
 *
 * V1.00        20-Nov-85
 *              DECnet-VAXmate, Version 1.1
 */

#include <stdio.h>
#include "types.h"
#include "dnmsdos.h"
#include "dn.h"
#include "socket.h"
#include "time.h"
#include "errno.h"
#include "scbdef.h"

#define MAX_BUF_SIZE 2048     /* maximum loop data buffer */

struct sockaddr_dn sockaddr; /* accept connect data structure */
struct optdata_dn opt;       /* optional data buffer */
char buff[MAX_BUF_SIZE];     /* data buffer */
int lsock = -1;              /* incoming connections on */
                             /* listening socket */
int sock = -1;               /* data communications socket */
char mode[1];                /* accept mode */
char msg_version[] = "MIRROR listening (V1.1)";
```

(continued on next page)

```
/*
 * Sample DECnet-VAXmate server task. This task will bind itself
 * as DECnet object number 25, the standard DECnet object
 * reserved for a mirror task. When started, the mirror is the
 * only running task. To terminate, the user may
 * press any key.

 */
main(argc, argv)
int argc;
char **argv;
{
    extern char *malloc();
    extern char *dnet_ntoa();
    int len;
    int nfds;
    unsigned long read;
    struct timeval tmv;

    /*
     * Set up listening socket for incoming connect requests.
     */
    if ((lsock = socket(AF_DECnet, SOCK_SEQPACKET, 0)) < 0)
       mir_exit("socket failed", errno);

    /*
     * Bind task to DECnet object 25.
     */
    bzero(&sockaddr, sizeof(sockaddr));
    sockaddr.sdn_family = AF_DECnet;
    sockaddr.sdn_objnum = 25;
    if (bind(lsock, &sockaddr, sizeof(sockaddr)) < 0)
        mir_exit("bind failed", errno);

    /*
     * Set up listening socket to listen for incoming connect
     * requests.  Allow for up to 5 pending incoming
     * connect requests.
     */
    if (listen(lsock, 5) < 0)
       mir_exit("listen failed", errno);

    /*
     * Listen for incoming connect requests until
     * there is keyboard input.
     */
    while(1)
    {
       /*
        * Display mirror version message.
        */
         printf("\n%s", msg_version);

      /*
       * Poll listening socket for incoming connect request.
       */
      while(1)
      {
          if (mir_keyboard_input())
```

(continued on next page)

```
            mir_exit(NULL, 0);
        bzero(&tmv, sizeof(tmv));
        read = 1 << lsock;
        nfds = lsock + 1;
        if (select(nfds, &read, 0, 0, &tmv) > 0)
        {
            if (read & (1 << lsock))
                break;
        }
    }

    /*
     * Issue a deferred accept on the connect request - send
     * some optional data along with it.
     */
    mode[0] = ACC_DEFER;
    if (setsockopt(lsock, DNPROTO_NSP, DSO_ACCEPTMODE,
                   &mode[0], sizeof(mode)) < 0)
    {
        mir_exit("set accept mode", 1);
    }

    len = sizeof(sockaddr);
    if ((sock = accept(lsock, &sockaddr, &len)) < 0)
        mir_exit("accept failed", errno);

    /*
     * Set up outgoing optional data - maximum mirror
     * data buffer size.
     */
    bzero(&opt, sizeof(opt));
    opt.opt_optl = sizeof(unsigned short);
    *(unsigned short *)&opt.opt_data[0] = MAX_BUF_SIZE;
    if (setsockopt(sock, DNPROTO_NSP, DSO_CONDATA, &opt,
        sizeof(opt)) < 0)
    {
        mir_exit("set socket option - optional data",
                 errno);
    }

    if (setsockopt(sock, DNPROTO_NSP, DSO_CONACCEPT,
        0, 0) < 0)
    {
        mir_exit("set connect accept", 1);
    }

    /*
     * Display peer information.
     */
    printf("\n");
    printf("\nLoop connect request from node: %s",
           dnet_ntoa(&sockaddr.sdn_add));

    if (sockaddr.sdn_objnum == 0)
        printf("\nRequesting object name: %s",
               &sockaddr.sdn_objname[0]);
    else
        printf("\nRequesting object number: %d",
               sockaddr.sdn_objnum);
    printf("\n");
```

(continued on next page)

```
        /*
         * Loop data while link is still active and other end is
         * still sending data.
         */
        while(!dnet_eof(sock))
        {
            len = MAX_BUF_SIZE;
            len = sread(sock, buff, &len);
            if (len == 0)
            {
                if (dnet_eof(sock))
                    mir_exit(NULL, 0);
            }
            else
            {
               if (len < 0)
                   mir_exit("sread", 1);
            }
            if (buff[0] != 0)
            {
                buff[0] = -1;
                len = 1;
            }
            else
            {
                buff[0] = 1;
            }

            if (swrite(sock, buff, len) < 0)
                mir_exit("swrite", 1);
        }

        /*
         * Finished with current data socket, close it up.
         */
        if (sock != -1)
            sclose(sock);
    }
}

int mir_keyboard_input()
{
    SCB scb;

    scb.AH = SCBC_CKSTAT;
    msdos(&scb);
    if (scb.AL)
        return(1);
    return(0);
}

mir_exit(sp, err)
char *sp;
int err;
{
    if (sp != NULL)
    {
        strcpy(buff, "\nmirror - ");
        strcat(buff, sp);
```

(continued on next page)

```
            perror(buff);
        }

        if (lsock != -1)
            sclose(lsock);

        if (sock != -1)
            sclose(sock);

        exit(err);
}
```

Index

A

B

C

D

E

M

N

O

P

R

S

T

U

W

HOW TO ORDER ADDITIONAL DOCUMENTATION

DIRECT TELEPHONE ORDERS

In Continental USA
and Puerto Rico
call 800–258–1710

In Canada
call 800–267–6146

In New Hampshire
Alaska or Hawaii
call 603–884–6660

ELECTRONIC ORDERS (U.S. ONLY)

Dial 800–DEC–DEMO with any VT100 or VT200
compatible terminal and a 1200 baud modem.
If you need assistance, call 800–DEC–INFO.

DIRECT MAIL ORDERS (U.S. and Puerto Rico*)

DIGITAL EQUIPMENT CORPORATION
P.O. Box CS2008
Nashua, New Hampshire 03061

DIRECT MAIL ORDERS (Canada)

DIGITAL EQUIPMENT OF CANADA LTD.
940 Belfast Road
Ottawa, Ontario, Canada K1G 4C2
Attn: A&SG Business Manager

INTERNATIONAL

DIGITAL
EQUIPMENT CORPORATION
A&SG Business Manager
c/o Digital's local subsidiary
or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC),
Digital Equipment Corporation, Northboro, Massachusetts 01532

*Any prepaid order from Puerto Rico must be placed
with the Local Digital Subsidiary:
809–754–7575

DECnet–VAXmate
Programmer's Reference Manual
AA–GV34A–TH

READER'S COMMENTS

What do you think of this manual? Your comments and suggestions will help us to improve the quality and usefulness of our publications.

Please rate this manual:

	Poor				Excellent
Accuracy	1	2	3	4	5
Readability	1	2	3	4	5
Examples	1	2	3	4	5
Organization	1	2	3	4	5
Completeness	1	2	3	4	5

Did you find errors in this manual? If so, please specify the error(s) and page number(s).

_____

_____

_____

_____

General comments:

_____

_____

_____

_____

Suggestions for improvement:

_____

_____

_____

_____

Name _____ Date _____

Title _____ Department _____

Company _____ Street _____

City _____ State/Country _____ Zip Code _____

DO NOT CUT FOLD HERE AND TAPE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY LABEL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

digital

Networks and
Communications Publications
550 King Street
Littleton, MA 01460–1289

DO NOT CUT FOLD HERE

CUT ALONG DOTTED LINE