VSI OpenVMS

VSI DECnet-Plus Programming

Operating System and Version: VSI OpenVMS IA-64 Version 8.4-1H1 or higher
VSI OpenVMS Alpha Version 8.4-2L1 or higher

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

VSI DECnet-Plus Programming

Legal Notice

Table of Contents

Preface

This manual contains information about how to design and write an application that uses the OpenVMS Interprocess Communication ($IPC) and Queue Input/Output ($QIO) system services, OpenVMS system service and OSI transport service, and the Common Management Information Service (CMISE) API.

1. About VSI

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

2. Intended Audience

This book is intended for programmers writing programs to perform task-to-task communication with remote systems. It is assumed that you have experience with the following:

- OpenVMS operating system.
- OpenVMS system services.
- A programming language supported by the OpenVMS operating system.
- Packet switching. If you are using a packet switching data network (PSDN), you should understand X.25 and packet switching terminology.

You are also assumed to have a knowledge of general communications theory.

3. Document Structure

The manual consists of the following chapters and appendices:

Chapter	Description
Chapter 1, "Introduction to $IPC", *Chapter 2, "Using the OpenVMS $IPC System Service "*, *Chapter 3, "$IPC Reference Calls"*, *Chapter 4, "Queue I/O Request ($QIO) System Service"*	Discuss the Interprocess Communication ($IPC) system service and provide a section for $IPC reference calls. This part also addresses the $QIO system service.
Chapter 5, "Introduction to OSI Transport Programming", *Chapter 6, "Programming Guidelines"*, *Chapter 7, "Calling the System Services"*, *Chapter 8, "System Service Calls Using Network Control Blocks"*, *Chapter 9, "System Service Calls Using Item Lists"*, *Chapter 10, "Negotiating*	Discuss the OpenVMS system services required to communicate with OSI transport services.

Chapter	Description
Protocol Classes and Options", *Chapter 11, "How OpenVMS OSI Transport Service Differs from DECnet-Plus for OpenVMS"*	
Chapter 12, "CMISE Introduction", *Chapter 13, "Common Management Information Services"*, *Chapter 14, "Checking CMISE Status Codes"*	discuss the CMISE API. Before using the CMISE API be sure that you have: ● Installed the DECnet-Plus for OpenVMS software ● Installed the VAX OSAK software as part of the DECnet-Plus for OpenVMS installation ● Configured OSI support in DECnet-Plus for OpenVMS using the NET$CONFIGURE.COM command procedure
Appendix A, "$QIO(W) Status Codes and OSI Reason Codes", *Appendix B, "Mailbox Message Types"*, *Appendix C, "Structure of an IOSB"*, *Appendix D, "LIB $PARSE_NCB"*, *Appendix E, "Programming Examples"*	Discuss the OSI transport services.

4. Related Documents

- *VSI DECnet-Plus for OpenVMS Network Management Guide*
- *VSI OpenVMS I/O User's Reference Manual*
- *VSI OpenVMS System Services Reference Manual*

If you are going to use a packet switching network, you should also be familiar with the *X.25 Introduction* book.

See *VSI DECnet-Plus for OpenVMS Introduction and User's Guide* for a bibliography of the documents that describe the various standards relevant to the OSI model and IEEE 802.3 local area networks.

For CMISE, you should have the following associated documents which define the standards implemented by the CMISE API:

- ISO 7498-4 Information Processing Systems — Open Systems Interconnection — Basic Reference Model — Part 4: Management Framework
- ISO 8649 Information Processing Systems — Open Systems Interconnection — Service Definition for the Association Control Service Element
- ISO 9595 Information Technology — Open Systems Interconnection — Common Management Information Service Definition
- ISO 9596 Information Technology — Open Systems Interconnection — Common Management Information Protocol Specification

5. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmssoftware.com>. Users who have VSI OpenVMS support contracts through VSI can contact <support@vmssoftware.com> for help with this product.

6. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

7. Acronyms and Abbreviations

The following acronyms and abbreviations apply throughout this book:

CR	connection request
CC	connection confirm
NA	Network Architecture
DR	disconnect request
DT	data TPDU
IPDU	internet protocol data unit
ISO	International Organization for Standardization
LES$ACP	LES Ancillary Control Process
NC	network connection
NCB	network connect block
NPDU	network protocol data unit
NS	network service
NSAP	network service access point
NSDU	network service data unit
OSI	Open Systems Interconnection
$QIO (W)	Both $QIO and $QIOW calls
TC	transport connection
TPDU	transport protocol data unit
TSAP	transport service access point
tsap-id	TSAP identifier
TSDU	transport service data unit
UAF	user authorization file

8. Typographical Conventions

VMScluster systems are now referred to as OpenVMS Cluster systems. Unless otherwise specified, references to OpenVMS Cluster systems or clusters in this document are synonymous with VMScluster systems.

The contents of the display examples for some utility commands described in this manual may differ slightly from the actual output provided by these commands on your system. However, when the behavior of a command differs significantly between OpenVMS Alpha and Integrity servers, that behavior is described in text and rendered, as appropriate, in separate examples.

In this manual, every use of DECwindows and DECwindows Motif refers to DECwindows Motif for OpenVMS software.

The following conventions are also used in this manual:

Convention	**Meaning**
Ctrl/*x*	A sequence such as **Ctrl/***x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
…	A horizontal ellipsis in examples indicates one of the following possibilities: ● Additional optional arguments in a statement have been omitted. ● The preceding item or items can be repeated one or more times. ● Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose the options in parentheses if you choose more than one.
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.
[|]	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are options; within braces, at least one choice is required. Do not type the vertical bars on the command line.
{ }	In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line.
bold text	This typeface represents the introduction of a new term. It also represents the name of an argument, an attribute, or a reason.
italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER= *name*), and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
`Monospace type`	Monospace type indicates code examples and interactive screen displays.

Convention	Meaning
	In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

Other conventions are:

- All numbers are decimal unless otherwise noted.
- All Ethernet addresses are hexadecimal.

Chapter 1. Introduction to $IPC

Interprocess Communication ($IPC) is a message-passing system service that enables you to perform interprocess communication on a single OpenVMS system as well as use the services of a DECnet Phase V network. $IPC also provides compatibility with Phase IV systems.

$IPC enables tasks to communicate without modification in the following environments:

- Within a single OpenVMS system.
- Between two nodes in an OpenVMS Cluster.
- Between a DECnet-Plus node and any other node in a DECnet network running DECnet-Plus or Phase IV software.
- Between a DECnet-Plus node and any other node in a multivendor system running software that is compatible with DECnet-Plus.

1.1. Connection and Data Transfer Functions

The $IPC system service provides several methods for establishing a connection to a network application. These include:

- DECdns object name
- Protocol tower
- Phase IV or DECnet-Plus node name and application identification

A network application defined as a DECdns object can reside anywhere in the network. The client task does not need to know where in the network the application currently resides. When establishing a connection to a DECdns object, DECnet-Plus retrieves the addressing and protocol information for the object from the DECdns namespace. The advantage of this method of establishing a connection is that the actual location of the application is hidden from the user.

The $IPC system service also allows a client application to specify the NA protocol tower for an application. The protocol tower, which is an element of the `DNA$Towers` attribute for a DECdns object, contains the address and protocol information necessary to communicate with a given network application. Specifying the DNS protocol tower for both client and server applications is the most efficient method of connection establishment because name server resolution is not required.

For compatibility with Phase IV, the $IPC system service allows you to specify the Phase IV 6-character node name to identify the remote system along with application identification information. The application identification information includes the Phase IV DECnet object number or task name. You may also specify the node name of the remote system in DECnet-Plus full name format with application identification information.

$IPC also allows you to make use of network-specific features such as the ability to send and receive optional data on connects and disconnects, the ability to segment transmitted messages, and the ability to send and receive expedited data. $IPC provides the orderly exchange of data between tasks and the controlled termination of the communication process and the association. $IPC can receive and process multiple inbound connection requests for a single application.

1.2. General $IPC Services

$IPC provides several functions that enable you to manage information about your node and connections. $IPC enables you to maintain protocol tower information in the name service. It enables you to obtain protocol tower, nodename, and connection information. It also enables you to verify node name information on an incoming connection.

For information about:

- Managing node name and connection information using $IPC, see *Section 2.13, "Managing Information"*.
- Obtaining protocol and address information from DECdns, see *Section 2.13.1, "Obtaining Local Protocol and Address Information"*.
- Functions you must complete to perform interprocess communication using $IPC, see *Chapter 2, "Using the OpenVMS $IPC System Service "*.

Chapter 2. Using the OpenVMS $IPC System Service

This chapter explains how to perform interprocess communication over a DECnet-Plus for OpenVMS network using the OpenVMS $IPC system service. It also includes a sample program that illustrates the use of the $IPC system service.

2.1. Introduction

The $IPC system service enables you to establish a dialogue within your distributed application to perform the following functions:

- Open an association
- Initiate a connection
- Complete a connection
  - Declare a network application
  - Accept a connection
  - Reject a connection
- Exchange messages
  - Send and receive data messages
  - Send and receive expedited data
  - Receive asynchronous event notification
- Terminate a connection
  - Synchronously disconnect a connection
  - Abort a connection
- Close an association
- Manage information
  - Obtain protocol tower information
  - Obtain naming information
  - Obtain connection information
  - Verify node name information
- Receive status and error reports

For a quick reference to the $IPC function codes, see *Table 2.1, "$IPCSystem Service Function Codes for Communication"* and *Table 2.2, "The $IPC System Service Function Codes to Manage Information"*.

2.2. The Application Database

The application database is a collection of information about the NA applications that reside on a local system. The application database includes information about both DECnet-Plus for OpenVMS applications such as Mail and the file access listener (FAL) as well as user-written applications.

The application database includes an entry for each application, known as a session control application entity. Using the Network Control Language (NCL), the network manager registers the application and enters the information, known as its characteristics. Examples of these characteristics are task name, image file name, and username. For more information about registering applications, refer to *VSI DECnet-Plus for OpenVMS Network Management Guide*.

You should be aware of the characteristics that the network manager has set for applications. The settings for these characteristics affect the operation of your applications.

You can temporarily modify a subset of these characteristics using $IPC. For more information on which characteristics you can modify, refer to *Section 2.4, "$IPC Function Codes for Communication"* for descriptions on using the function codes.

The application database lists application objects running on the local system. It also records characteristics of each application. The network manager creates and maintains the application database through NCL.

The Session Control layer uses the list to associate incoming connection requests with the relevant process to handle them.

2.3. Passing and Receiving Information from the $IPC System Service

The following data structures enable you to pass information to and receive information from $IPC:

- Interprocess Communication Block (IPCB)
- Network item list

This section describes these data structures and explains how to use them in your application.

2.3.1. Using the Interprocess Communication Block (IPCB)

All $IPC functions require an IPCB. If you are issuing asynchronous $IPC calls, a different IPCB must be supplied for each outstanding call. When a $IPC function completes, the IPCB can be used again for another $IPC function.

The file SYS$LIBRARY:NET_EXTERNALS defines the IPCB.

The IPCB contains required and optional information needed to perform the function. You use the IPCB to do the following:

- Pass identification information about an association or connection.
- Identify a target task.
- Modify the action of the $IPC function codes.

- Pass the address of a buffer you supply for receipt or transmission of data.
- Request information from $IPC.

The $IPC system service uses the Interprocess Communication Block (IPCB) to do the following:

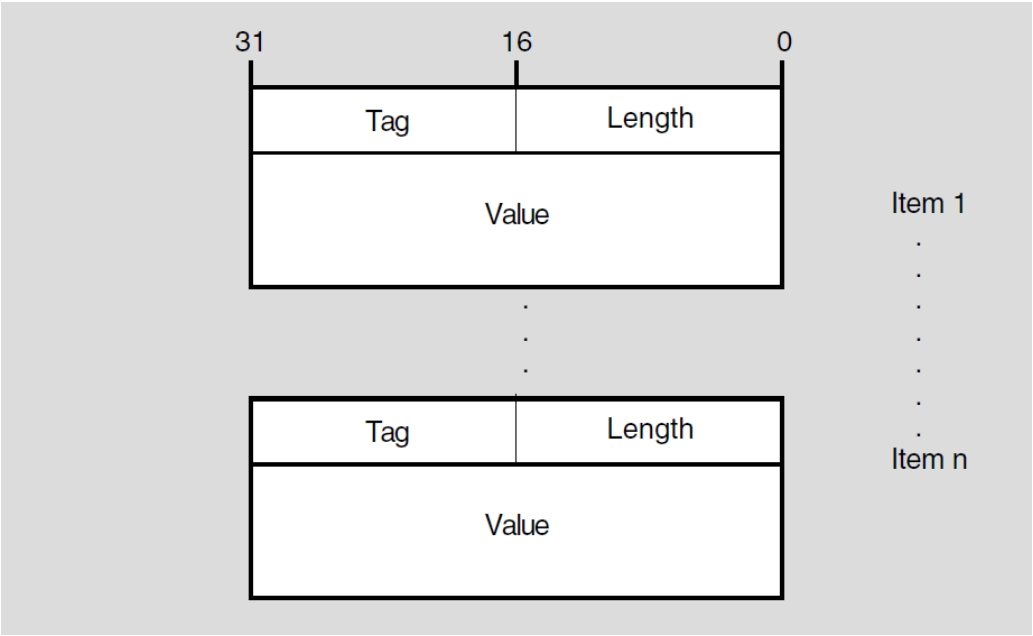
- Return information about an association or connection.
- Provide the length of the buffer that contains return information.
- Provide the completion status of a request.

2.3.2. Using Network Item Lists

A network item list is a contiguous block of memory containing one or more items, each of which contains the item's actual value. You use network item lists to pass required or optional information to the $IPC system service and to request information from $IPC. The $IPC system service builds an output item list to pass requested information to you.

Figure 2.1, "Network Item List" shows how the items are formatted.

Figure 2.1. Network Item List



Item	Description
Length	The 16-bit length of the body of the item, including both length and tag fields. For example, an item whose value is a longword has a length field of 8 bytes. Items may be fixed or variable length. All items include the length field.
Tag	A 16-bit item code indicating the nature of the information to be passed. Each item code has a symbolic name; these symbolic names have the format NET $K_TAG_code.
Value	The value of the data. A null value is allowed, in which case the default length is 4 bytes.

Specify types of data in the correct format, depending on the item code:

- ASCII character string (includes DECdns full name string)

- DECdns opaque full name
- Hexadecimal data
- Longword of flags
- Longword value
- Protocol tower
- Protocol tower set

You pass network item lists in the IPCB. The following types of network item lists are passed in the IPCB.

- Input Item List — You use the input item list to pass required and optional information to the $IPC system service. Specify the input item list using the IPCB$Q_INPUTLST_DESC field.
- Template Item List — You use the template item list to specify which items the $IPC system service should return (in the output item list) upon completion of the request. Specify the template list using the IPCB$Q_TEMPLATETLST_DESC field. Each item consists of a length and tag field. The length is always 4 bytes since no actual data values are included in a template.
- Output Item List — The output item list contains the items specified in the template item list. Specify the output item list using the IPCB$Q_OUTPUTLST_DESC field.

2.4. $IPC Function Codes for Communication

Table 2.1, "$IPCSystem Service Function Codes for Communication" summarizes the function codes discussed in *Section 2.6, "Opening an Association"* through *Section 2.12, "Terminating an Association"*. See *Section 3.4, "Function Codes"* for a more complete description of each function code.

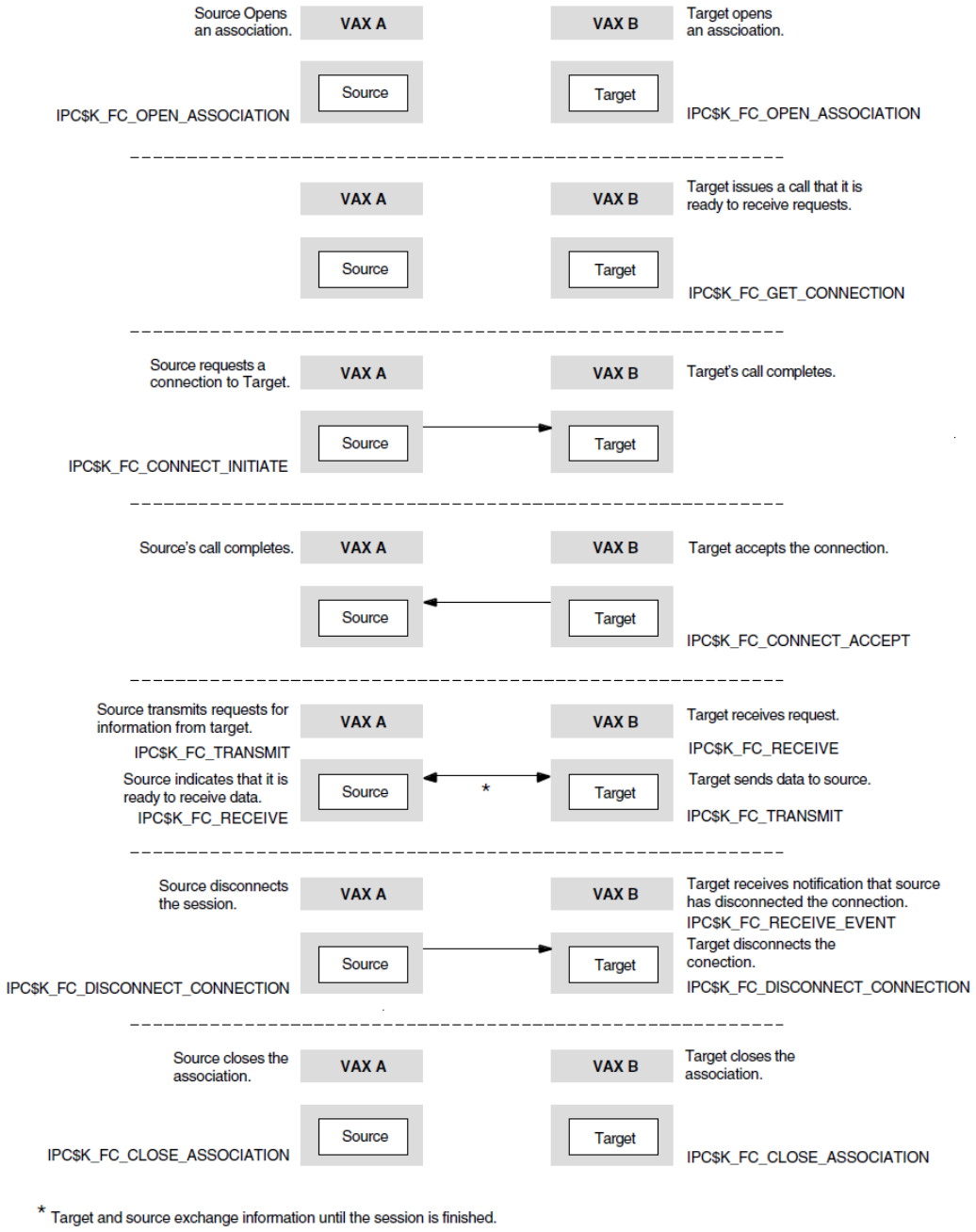
Table 2.1. $IPCSystem Service Function Codes for Communication

Function Code	Function
IPC$K_FC_ABORT_CONNECTION	Terminate the connection
IPC$K_FC_CLOSE_ASSOCIATION	Terminate the association with the Session Control layer
IPC$K_FC_CONNECT_ACCEPT	Accept a received connect request
IPC$K_FC_CONNECT_INITIATE	Request a logical connection to a target task
IPC$K_FC_CONNECT_REJECT	Reject a received connect request
IPC$K_FC_DISCONNECT_CONNECTION	Synchronously disconnect the connection
IPC$K_FC_GET_CONNECTION	Associate an incoming connection request with the calling process
IPC$K_FC_OPEN_ASSOCIATION	Create an association between an application and the Session Control layer
IPC$K_FC_RECEIVE	Receive a message from the target application
IPC$K_FC_RECEIVE_EVENT	Receive event notification from the network
IPC$K_FC_SHUT_ASSOCIATION	Terminate the connection between session and the calling application
IPC$K_FC_TRANSMIT	Send a message to the target application

2.5. Source and Target $IPC Operations

Figure 2.2, "Source and Target Communicating Using the $IPC System Service" illustrates the communication operations between two tasks using $IPC. The tasks, called Source and Target, are running on different systems in a DECnet-Plus network.

Figure 2.2. Source and Target Communicating Using the $IPC System Service



* Target and source exchange information until the session is finished.

2.6. Opening an Association

To prepare for interprocess communication using $IPC, you must open an association with the Session Control layer. Opening an association notifies the NA Session Control layer that a task will be using DECnet-Plus for OpenVMS services. A client is a program that initiates communications to a task; a server is a task with which the client wants to communicate.

When you open an association for a task acting as a server, you inform the Session Control layer that the task is available to process inbound connection requests. If a task is acting as a client, you inform the Session Control layer that the task will be requesting a connection to a (remote) task.

To open an association in your application, use the $IPC system service call with a function code of IPCK_FC_OPEN_ASSOCIATION. When you open an association for a client task, you usually specify a null input item list. When you open an association for a server task, do the following:

- Set privileges.
- Declare the name of your application to the Session Control layer.
- Indicate if node name verification is required on the incoming connection request.

Declaring the Name of Your Server Application

You declare the name of your server application, or end user identification, in an input item list. You can pass this information in the following forms:

- DECdns full name
- Phase IV task name or Phase IV object number

A DECdns opaque full name is the internal representation of the complete path specification to the object. DECdns provides a routine that converts the full name string, or external string name as seen and entered by users, to its opaque full name. For information on this routine, refer to DECnet/OSI DECdns Programming. You can also specify the end user identification using the Phase IV task name and the Phase IV object number.

The following table shows the item code, the data specified, and the format of the data specified when declaring a name:

Item Code	**Data**	**Format**
NET$K_TAG_ENDUSERID_NAME	Opaque full name	DECdns internal representation
NET$K_TAG_ENDUSERID_TASK	Task name	1 to 12 ASCII characters
NET$K_TAG_ENDUSERID_NUMBER	Task number	ASCII string representing decimal 0 – 255

For more information about network management, see *VSI DECnet-Plus for OpenVMS Network Management Guide*. For more information about NCL, see *VSI DECnet-Plus for OpenVMS Network Control Language Reference Guide*.

Indicating if Node Name Verification is Required

When opening an association for a server task, you can indicate whether node name verification is required on an incoming request. Use node name verification to ensure that the node with which you are communicating corresponds to its name. The node name verification requirement is the default for this call. Set the IPCB$V_FLAGS_NOVERIFY_NODENAME flag to bypass node name verification.

$IPC will return the status of the operation and an association ID (if the association request is successful) via the IPCB.

Note

If the application is defined in the application database, the database parameters override the parameters specified in this call. Use this call to augment the values in the application database or to open an association for private applications (applications that are not registered in the application database). For more information on registering applications in the application database, see *VSI DECnet-Plus for OpenVMS Network Management Guide*.

2.7. Enabling Event Notification

If you want to receive network events, you must first enable event notification. Do this by passing an input item list containing NET$K_TAG_EVENTMASK to IPC with the OPEN_ASSOCIATION function. This item contains a longword that is interpreted by IPC as a bit mask; each bit set enables a specific type of notification. The bits used by IPC are defined by:

- NET$V_EVENT_INCOMING
- NET$V_EVENT_EXPEDITED
- NET$V_EVENT_DISCONNECTS

2.8. Initiating a Connection

After a client task opens an association, it can request a connection to a target task. A client task must request a connection to a target task before any message exchange can take place. You request a connection to a target task by using the IPC$K_FC_CONNECT_INITIATE function code. The IPC$K_FC_CONNECT_INITIATE function code, appropriate item codes, and flags enable you to do the following:

- Identify the target task.
- Specify client protocol and address information.
- Disable outgoing proxy.
- Automatically disconnect the connection if network connectivity is lost.
- Specify optional access verification information.
- Pass a user-specified longword to the Session Control layer.
- Request client and target address information.
- Send optional user data.

A request for a connection succeeds if the server task accepts the connection. The request fails if the server task rejects the connection attempt, or does not respond within a time period. (For more information on connection control, see the *VSI DECnet-Plus for OpenVMS Network Control Language Reference Guide*.)

2.8.1. Identifying the Target Task

You can identify the target task by specifying one of the following in an input item list:

- DECdns object name and optional target information

- Protocol tower and optional client protocol tower information
- Phase IV or DECnet-Plus node name with end user identification
- Client name of the source task

DECdns Object Name and Optional Target Information

You can specify the full name of a target application that has been registered with DECdns. You do not have to specify protocol or address information when you use the DECdns object name. The Session Control layer retrieves the DNA$Towers attribute for that object. The Session Control layer extracts protocol and address information for an object from the DNA$Towers attribute.

You can pass the DECdns full name in opaque full name format using the NET$K_TAG_DNSOBJECTNAME_INT item code or use NET$K_TAG_DNSOBJECTNAME to pass the external string name. You can use DECdns routines to convert an external string name to an opaque full name.

You also have the option of passing user tower information along with the object name. An application can maintain a form of a protocol tower known as a user tower. A protocol tower is the mechanism that DECnet-Plus for OpenVMS software and DECdns use to coordinate protocol and address information. An application maintains protocol and addressing information about where its specific services are located in the NA Application layer of the protocol tower. This information is stored in the name service.

The user tower extends from the Session Control layer upwards, as represented in the following table:

Layer	**Protocol Identifiers**	**Associated Information**
NA Application	Level 2	
NA Application	Level 1 protocol identifier	NA Application layer parameters and addressing to reach Level 2 of the NA Application layer module
Session	Session layer protocol identifier	Session layer protocol parameters and addressing to reach Level 1 of the NA Application layer module

You should pass the user tower information in protocol tower set format using the NET $K_TAG_USERTOWER item code. For information on obtaining protocol tower sets, see *Section 2.13.3, "Obtaining Protocol Tower Information"*.

Protocol Tower and Optional Client Protocol Tower Source Information

You can also identify the target task by specifying the appropriate protocol tower for the target object. A protocol tower includes explicit address and protocol information from the Network to the Session Control layers.

You can obtain this information through the $IPC system service by using the IPC$K_FC_RESOLVE_NAME function code. This function returns the set of protocol towers mutually supported by the client and target nodes. You can then choose the protocol tower that you want to use to connect with the target node. You must pass this information in an input item list using the NET$K_TAG_DESTINATIONTOWER item code. For information on the

IPC$K_FC_RESOLVE_NAME function code, see *Section 2.13.3, "Obtaining Protocol Tower Information"*.

Use this mode of task specification if you want to use one protocol over another.

You also have the option of specifying the protocol and address information for the client node when you identify a target by protocol tower. You specify this information, known as the source towers, using the NET$K_TAG_SOURCETOWER item code.

You can obtain source protocol and address information by using the IPC$K_FC_ENUMERATE_LOCAL_TOWERS function code. For more information on this call, see *Section 2.13.1, "Obtaining Local Protocol and Address Information"*.

Phase IV or DECnet-Plus Node Name with End User Identification

You can also specify the target application by providing the node name on which the application resides along with the appropriate end user identification. You can specify the node name as a DECnet-Plus DECdns full name string or in Phase IV six-character format using the NET$K_TAG_NODENAME item code. You can specify the opaque full name of the node using the NET$K_TAG_NODENAME_INT item code.

The end user identification can be the Phase IV object number (NET$K_TAG_ENDUSERID_NUMBER), the Phase IV task name (NET$K_TAG_ENDUSERID_TASK), or the DECdns opaque full name of the target application (NET$K_TAG_ENDUSERID_NAME).

Specifying the Client Name of the Source Task

You have the option of specifying the client name of the source task in the input item list using the NET$K_TAG_CLIENTNAME item code. The client name is in ASCII string format and is the name network managers use to identify which application is using a specific connection. If you do not provide the client name, the Session Control layer will construct one for your application.

2.8.2. Disabling Outgoing Proxy

If you use the IPCB$V_FLAGS_NOPROXY function modifier, you can send the connection request with outgoing proxy disabled. For more information on proxy, see *VSI DECnet-Plus for OpenVMS Network Management Guide*.

2.8.3. Automatically Disconnecting the Connection

Use the IPCB$V_FLAGS_AUTODISCONNECT function modifier to automatically disconnect the connection if it appears to the DECnet-Plus for OpenVMS services that network connectivity has been lost. If the DECnet-Plus for OpenVMS services lose connectivity, you will receive notification before the default timeout period elapses.

Note

After you receive notification of a disconnection of this type, you must formally disconnect the connection by issuing the IPC$K_FC_DISCONNECT_CONNECTION or IPC$K_FC_ABORT function codes or the Session Control layer will issue IPC$K_FC_ABORT for you when your process exits. For more information on the IPC$K_FC_DISCONNECT_CONNECTION function code, see *Section 2.11.1, "Synchronously Disconnecting a Connection"*. For more information on the IPC$K_FC_ABORT function code, see *Section 2.11.2, "Aborting a Connection"*.

2.8.4. Specifying Optional Access Verification Information

You can also specify optional access verification information such as the destination user name (using the NET$K_TAG_DESTINATIONUSER item code), the access verification password (using the NET$K_TAG_DESTINATIONPASSWORD item code), and the destination account (using the NET$K_TAG_DESTINATIONACCOUNT item code) when initiating a connection. You specify these in an input item list.

2.8.5. Requesting Source and Target Address Information

You can request that $IPC return the source and destination protocol towers used to make the connection. You can request the source protocol tower by specifying the NET$K_TAG_SOURCETOWER item code in the template item list. You request the destination protocol tower by specifying the NET$K_TAG_DESTINATIONTOWER item code in the template item list.

2.8.6. Passing a User-Specified Longword to the Session Control Layer

A task can pass a longword of information to the Session Control layer by using the IPCB$L_CONNECTION_CONTEXT field of the IPCB.

2.8.7. Sending Optional User Data

The $IPC functions IPC$K_FC_CONNECT_INITIATE and IPC$K_FC_DISCONNECT_CONNECTION allow you to send up to 16 bytes of optional user data. The fields IPCB$L_BUFFER_LENGTH and IPCB$A_BUFFER define the length and address of the data. User data sent with the IPC$K_FC_CONNECT_INITIATE function is received with the IPC$K_FC_GET_CONNECTION function in the buffer described with the fields IPCB$L_BUFFER_LENGTH and IPCB$A_BUFFER. The field IPCB$L_RET_BUFFER_LENGTH contains the number of bytes of user data actually sent.

The $IPC functions IPC$K_FC_CONNECT_ACCEPT and IPC$K_FC_CONNECT_REJECT may return up to 16 bytes of optional user data. The fields IPCB$L_REPLY_LENGTH and IPCB$A_REPLY_BUFFER describe the data to be sent. The IPC$K_FC_CONNECT_INITIATE function receives this data in the buffer described with the fields IPCB$L_REPLY_LENGTH and IPCB$A_REPLY_BUFFER. The actual number of bytes of data received is written to the IPCB$L_RET_REPLY_LENGTH field.

2.9. Completing the Connection

When a target (server) task is ready to receive connections, the target task must issue a call to get a connection. The Session Control layer completes the call with an outstanding connection request for that task or waits until it receives a connection request for that task. There is no timeout period. The target task must then accept or reject the connection.

To obtain a connection, use the IPC$K_FC_GET_CONNECTION function code. A task that is to receive multiple connection requests must reissue the IPC$K_FC_GET_CONNECTION request for each connection.

See *Section 2.8.7, "Sending Optional User Data"* for information on optional user data.

2.9.1. Accepting a Connection

After issuing an IPC$K_FC_GET_CONNECTION operation, the target task must accept the connection request before it can exchange messages. Use the IPC$K_FC_CONNECT_ACCEPT function code to accept a received connection request. You must specify an association ID and a connection ID when you accept a connection request.

When accepting a request, you can use the IPCB$V_FLAGS_AUTODISCONNECT function modifier to request that the DECnet-Plus software automatically disconnect the connection if it appears that network connectivity has been lost. For more information on using the IPCB$V_FLAGS_AUTODISCONNECT function modifier, see *Section 2.8.3, "Automatically Disconnecting the Connection"*.

You can pass a longword of information to the Session Control layer by using the IPCB$L_CONNECTION_CONTEXT field of the IPCB.

See *Section 2.8.7, "Sending Optional User Data"* for information on optional user data.

2.9.2. Rejecting a Connection

Use the IPC$K_FC_CONNECT_REJECT function code to reject a received request from an IPC$K_FC_CONNECT_INITIATE operation. You must specify an association ID and a connection ID when you reject a connection request.

See *Section 2.8.7, "Sending Optional User Data"* for information on optional user data.

2.9.3. Requesting Node Names

You can request the name of the node initiating the connection by specifying NET$K_TAG_NODENAME_INT (for the DECdns opaque full name of the node) or NET$K_TAG_NODESYNONYM (for the Phase IV six-character nodename) item codes in the template item list used with the IPC$K_FC_GET_CONNECTION function code. You can request the end user identification of the application with the NET$K_TAG_SOURCENAME, NET$K_TAG_SOURCENUMBER, NET$K_TAG_SOURCETASK, or NET$K_TAG_SOURCEUIC item codes. You can also request the source username using the NET$K_TAG_SOURCEUSER item code. You can request optional access verification information such as the destination user name (using the NET$K_TAG_DESTINATIONUSER).

2.10. Exchanging Messages

Two tasks are ready to exchange messages after the following has occurred:

- Source and target tasks have created associations
- The source task has requested a connection to a target task
- The target task has requested and accepted a connection to a source task

For each message sent by a task, the receiving task must issue the appropriate call to receive the message. Also, the two tasks must agree (via the application protocol) on which task will disconnect the link and whether they will pass optional user data. In the context of a connection, the task sending a message is the transmitter and the task receiving is the receiver. Because connections are full-duplex, each task may be a transmitter and a receiver simultaneously.

You use the data transfer services of the $IPC system service to transmit and receive data across a connection. For each of these services, you must specify a buffer that will contain the data in the IPCB. When you request data transfer services, you cannot use the buffer until the transmit or receive operation completes (the asynchronous trap (AST) has been triggered or the event flag has been set and the IPCB status field has been filled in).

This section describes various types of network messages. It also explains how to perform the following functions:

- Send data messages and expedited data
- Receive data and expedited data
- Receive network-specific information

For information on sending and receiving data and expedited data, refer to *Section 2.10.1, "Sending Data"* and *Section 2.10.2, "Receiving Data"*.

Network-Specific Information

The DECnet network issues network-specific information known as asynchronous event notification. A task can receive the following types of event messages:

- Disconnects
    - Messages that the DECnet-Plus for OpenVMS network generates when the task initiates certain network operations. For example, when one task synchronously disconnects or aborts a connection, a notification message is placed in the specified buffer of the target task.
    - Network event notification messages that inform a task of some unusual network occurrence (such as a third-party disconnect).
    - Notification of network software- and hardware-related problems
    - Notification of connection request timeouts
- Expedited data
- Incoming connection

2.10.1. Sending Data

Use the IPC$K_FC_TRANSMIT function code to send a message to the target application. You must specify an association, a connection ID, and a transmit buffer address and length in the IPCB when you send data.

The IPCB$M_FLAGS_MULT function modifier enables you to transmit segmented data. You can segment the message into *n* number of segments. You should send the first n - 1 messages with the IPCB$M_FLAGS_MULT flag set. You should send the final segment without the flag, signaling to the DECnet network that this is the end of this segmented message. The receiver can either specify a single receive operation with a buffer large enough to hold all the segmented messages or issue numerous IPC$K_FC_RECEIVE requests with the IPCB$M_FLAGS_MULT flag set.

Sending Expedited Data

The IPC$K_FC_TRANSMIT request, along with the IPCB$M_FLAGS_EXPEDITED function modifier, transmits 16 bytes of expedited data to the target task.

You cannot set the IPCB$M_FLAGS_EXPEDITED function modifier in combination with the IPCB$M_FLAGS_MULT flag.

2.10.2. Receiving Data

The target application receives data messages by issuing the IPC$K_FC_RECEIVE function code. You must specify the association ID, connection ID, and a receive buffer address and length in the IPCB. $IPC returns the length of the data returned and the status of the operation in the IPCB.

The IPCB$M_FLAGS_MULT function modifier enables you to receive segmented data. If you post a receive buffer with the IPCB$M_FLAGS_MULT flag set, the DECnet-Plus for OpenVMS software fills the buffer with as much data as will fit and then sets the IPCB completion status (IPC$_MOREDATA). If you issue another IPC$K_FC_RECEIVE call with the IPCB$M_FLAGS_MULT flag set, the DECnet-Plus for OpenVMS software gives you the next segment of the message. If you do not set the IPCB$M_FLAGS_MULT flag and receive a message that is larger than the receive buffer, you will lose the end of the message (Error—IPC$_DATA_OVERRUN).

Receiving Expedited Data

To receive expedited data, use the IPCB$M_FLAGS_EXPEDITED function modifier.

Receiving Network-Specific Information

To receive network-specific information (asynchronous event notification), issue the IPC$K_FC_RECEIVE_EVENT function code.

Events are returned on a per-association basis.

$IPC indicates the type of event that was received in the IPCB$L_EVENT_TYPE field. The three types of events are:

- IPC$K_FC_INCOMING_CONNECT
- IPC$K_FC_INCOMING_DISCONNECT
- IPC$K_FC_INCOMING_EXPEDITED

2.11. Terminating a Connection

The termination of a connection signals the end of the communication between tasks.

A task using the OpenVMS $IPC system service can terminate communication with a remote task by disconnecting the connection either by a synchronous disconnector a disconnect abort. This section describes how to terminate a connection using these methods.

2.11.1. Synchronously Disconnecting a Connection

When you synchronously disconnect a connection, you specify that all messages sent by the source task must be received and acknowledged by the remote Transport layer before the connection is disconnected. You should use this type of disconnect when the user of the connection's services wants to ensure that the transmission of messages has completed before terminating the connection. Note, however, that the service cannot guarantee the delivery of the received data to the remote task, only that it is available for the task to receive if it chooses.

To synchronously disconnect the connection, use the IPC$K_FC_DISCONNECT_CONNECTION function code. You must specify an association ID and a connection ID in the IPCB. The DECnet-Plus for OpenVMS software transmits all pending messages to the remote system and acknowledges them before the connection is disconnected.

You can send up to 16 bytes of optional user data in the IPCB when you perform a synchronous disconnect operation. To send optional user data, you must specify the IPCB$L_BUFFER and IPCB$L_BUFFER_LENGTH fields in the IPCB.

2.11.2. Aborting a Connection

When you abort a connection, all messages that the source task sends might not be received or acknowledged by the Transport layer at the target system before the connection is disconnected. You should use this type of disconnect when the user of the connection's services intends to reset the connection to a known state.

When a task terminates a connection and the target is looking for events, the network sends a notification message in the form of an asynchronous event notification indicating that the connection is disconnected.

In other cases, any outstanding IPC calls are completed with a NET$_ABORT error.

To abort a connection, use the IPC$K_FC_ABORT_CONNECTION function code. Any pending messages will be lost. You must specify an association ID and a connection ID when you abort a connection.

2.12. Terminating an Association

This section describes terminating an association with the Session Control layer. It explains how to perform the following functions:

- Stopping connections to the task's association
- Closing the association

2.12.1. Stopping Connections to the Task's Association

To stop any further connections from being accepted by an application's association with the Session Control layer, use the $IPC system service with a function code of IPC$K_FC_SHUT_ASSOCIATION. You must specify an association ID in the IPCB.

When you stop connections from being accepted by the association, all existing connections will continue to work until disconnected or aborted by the application. This operation gives server applications a graceful way to begin terminating the association.

2.12.2. Closing an Association

Closing an association terminates all pending I/O and disconnects all connections for the specified association.

A function code of IPC$K_FC_CLOSE_ASSOCIATION closes the association between the Session Control layer and the calling application. Issue this call only after all communication between tasks is complete.

You must specify an association ID in the IPCB.

2.12.3. Programming Examples

Programming examples of a client and server communicating using the $IPC system service are located in SYS$COMMON:[SYSHLP.EXAMPLES.DNVOSI].

2.13. Managing Information

OpenVMS $IPC enables you to perform several informational functions when writing your application. You can:

- Obtain protocol tower, node name, and connection information.
- Maintain protocol tower information in the Distributed Name Service.
- Verify node name information for a source application.

2.13.1. Obtaining Local Protocol and Address Information

Use the IPC$K_FC_ENUMERATE_LOCAL_TOWERS function code to obtain the protocols and associated address information that your local system supports. You need this information to do the following:

- Set the DNA$Towers attribute when creating an object in the name service. This call does not, however, return your end user specification. The end user specification supplies the address of your application to the Session Control layer and is built from the end user identification. For information on the end user identification, see *Section 2.8.1, "Identifying the Target Task"*.
- Connect to a target application by protocol tower. You would see what your local system supported, then compare those with the protocols supported by the remote system.

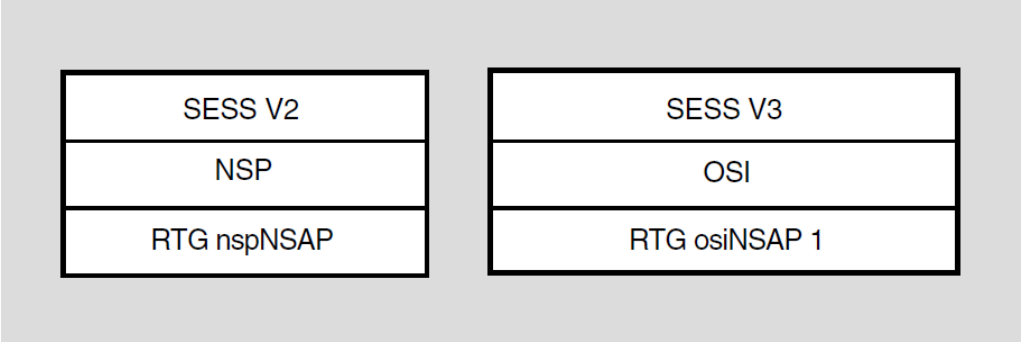
Specify the NET$K_TAG_SOURCETOWERSET item code in a template item list. $IPC returns the local protocol tower information in protocol tower set format in an output item list.

Protocol Tower Sets

When you request protocol and addressing information for an object, the system builds a separate protocol tower for each transport protocol and each address through which each version of the Session Control layer (there are two) might access the Transport module.

Figure 2.3, "Protocol Tower Set Example" shows an example of a protocol tower set.

Figure 2.3. Protocol Tower Set Example



In *Figure 2.3, "Protocol Tower Set Example"*, the Transport layer indicates to the Session Control layer that Session can use the services of NSP and OSI transport. The protocol tower set lists the addresses of the NSP (nspNSAP) and OSI transport protocols (osiNSAP1) in separate towers.

Section 2.13.7, "Protocol Tower Fields" describes the protocol tower set format.

2.13.2. Maintaining the DNA$Towers Attribute

The Session Control layer periodically updates the DNA$Towers attribute in DECdns for all application objects residing on the local system that have requested this service. The DNA$Towers attribute contains protocol and address information that can be used to find the node on which a resource resides.

Use the IPC$K_FC_REGISTER_OBJECT function code to request that the Session Control layer maintain the DNA$Towers attribute of your application object in the DECdns namespace. You can specify the name of your application object in DECdns opaque full name format using the NET$K_TAG_DNSOBJECTNAME_INT item code or in external string name format using the NET$K_TAG_DNSOBJECTNAME in an input item list. You can also request that the Session Control layer add the user towers to the DNA$Towers attribute by specifying the NET$K_TAG_USERTOWERSET item code as well.

Note

For the Session Control layer to provide this service, you must have previously created the application object in the namespace. The access control on the DNA$Towers object must allow modification by the local Session Control module. If you have created the object in the namespace and have not set up the DNA$Towers attribute, the Session Control layer will do it for you.

The Session Control layer stores a list of registered application objects in the tower maintenance database. When you issue the IPC$K_FC_REGISTER_OBJECT function code, the Session Control layer automatically enters the name of the application object in the database. The network manager can delete the entries in the tower maintenance database using NCL. For more information on the tower maintenance database, see *VSI DECnet-Plus for OpenVMS Network Management Guide*.

The Session Control layer maintains information in the namespace by periodically comparing the information in the tower maintenance database with that in the namespace, adding current information, and removing outdated information.

Use the IPC$K_FC_DEREGISTER_OBJECT to stop the Session Control layer from maintaining the DNA$Towers attribute for the specified object.

For more information about creating an object with DECdns, see *VSI DECnet-Plus for OpenVMS DECdns Management Guide*.

2.13.3. Obtaining Protocol Tower Information

$IPC enables you to obtain the set of protocol towers mutually supported by your local system and the remote system on which the application object you specify resides. Use the IPC$K_FC_RESOLVE_NAME function code to perform this function. This information is useful if you want to connect to an object by protocol tower.

You can pass user tower information in this call by specifying the NET$K_TAG_USERTOWERSET item code along with the NET$K_TAG_DNSOBJECTNAME_INT or NET$K_TAG_DNSOBJECTNAME item code in an input item list.

You can also request the compatible source and destination towers by specifying the NET$K_TAG_DESTINATIONTOWERSET and the NET$K_TAG_SOURCETOWERSET item codes in a template item list. $IPC returns this information in protocol tower set format. For more information about protocol tower sets, see the *Section 2.13.1, "Obtaining Local Protocol and Address Information"*.

2.13.4. Obtaining Node Name Information

The IPC$K_FC_BACKTRANSLATE function enables the calling process to obtain the node name associated with the supplied address. You must supply one data item in the input item list and any combination of items in the template list. All data items requested in the template list are returned in the output item list if available. Items not available are returned as null items in the output item list. If no items were available, an error status is returned.

To obtain the node synonym, specify the NET$K_TAG_NODESYNONYM item code in a template item list. To obtain the DECdns opaque full name, specify the NET$K_TAG_NODENAME_INT; to obtain the DECdns full name string, specify the NET$K_TAG_NODENAME item code.

2.13.5. Obtaining Connection Information

Use the IPC$K_FC_GET_PORT_INFORMATION function code to request information about a connection. This request returns information about the SESSION CONTROLPORT entity. This operation returns the same information as does the NCL command: SHOW SESSION CONTROL PORT. You submit the association ID and connection ID in the IPCB. You can request any or all of the following by specifying the appropriate item code in an input item list:

- Network management name assigned to the port – NET$K_TAG_CLIENTNAME.
- Phase IV 6-character node synonym for the source system – NET$K_TAG_NODESYNONYM.
- Item code specifies the name of the target node – NET$K_TAG_NODENAME.
- DECnet-Plus node name in opaque full name format – NET$K_TAG_NODENAME_INT.
- Item code specifies the address of the source node – NET$K_TAG_SOURCEADDRESS.
- Item code indicates the NSAP address of the target node – NET$K_TAG_DESTINATIONADDRESS.
- Whether port is initiating an outgoing or incoming connection – NET$K_TAG_DIRECTION.

2.13.6. Verifying Node Name Information

The IPC$K_FC_VERIFY_NODENAME operation enables you to verify that the nodename associated with an inbound connection is valid.

When the Session Control layer receives an incoming connection request, it stores the address of the source node. When you issue this call, the Session Control layer verifies that the address information associated with the nodename you specify is consistent with the address information for that node object in the DNA$Towers attribute.

You specify the NET$K_TAG_NODENAME, NET$K_TAG_NODENAME_INT, or NET$K_TAG_NODESYNONYM item codes in an input item list. You also specify your association ID and a connection ID. The Session Control layer returns the verification in the IPCB$L_STATUS field of the IPCB if the node name is valid.

2.13.7. Protocol Tower Fields

Address

Protocol-dependent addressing information.

Address Length

The length, in bytes, of the address (not including the length of the *Address Length* field itself). Length is 2 bytes.

#Pairs

The number of protocol/address pairs in the protocol tower. Length is 2 bytes.

Protocol Identifier

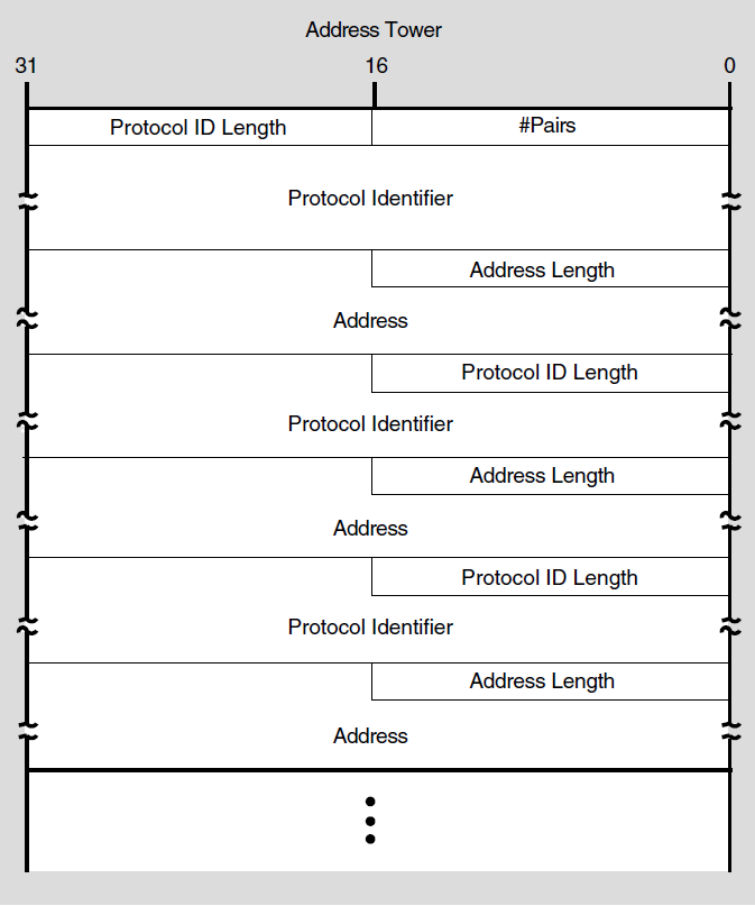
The protocol layer to which the address information refers.

Protocol ID Length

The length, in bytes, of the protocol identifier (not including the length of the *Protocol ID Length* field itself). Length is 2 bytes.

Figure 2.4, "Protocol Tower Data Structure" shows a protocol tower data structure.

Figure 2.4. Protocol Tower Data Structure



2.13.8. Protocol Tower Set Fields

Flag

Signifies whether the information in the tower set is valid. If the low-order bit of the flag byte is set, the information is valid. Length is 1 byte.

Member Array Length

Total length, in bytes, of all the protocol tower set members, not including the length of the *Member Array Length* field. Length is 2 bytes.

Offset Array Length

Length, in bytes, of all the relative offsets in the offset array, not including the length of the *Offset Array Length* or *Member Array Length* field. Length is 2 bytes.

Relative Offset

Number of bytes from the start of the protocol tower set to where the member begins. Length is 2 bytes.

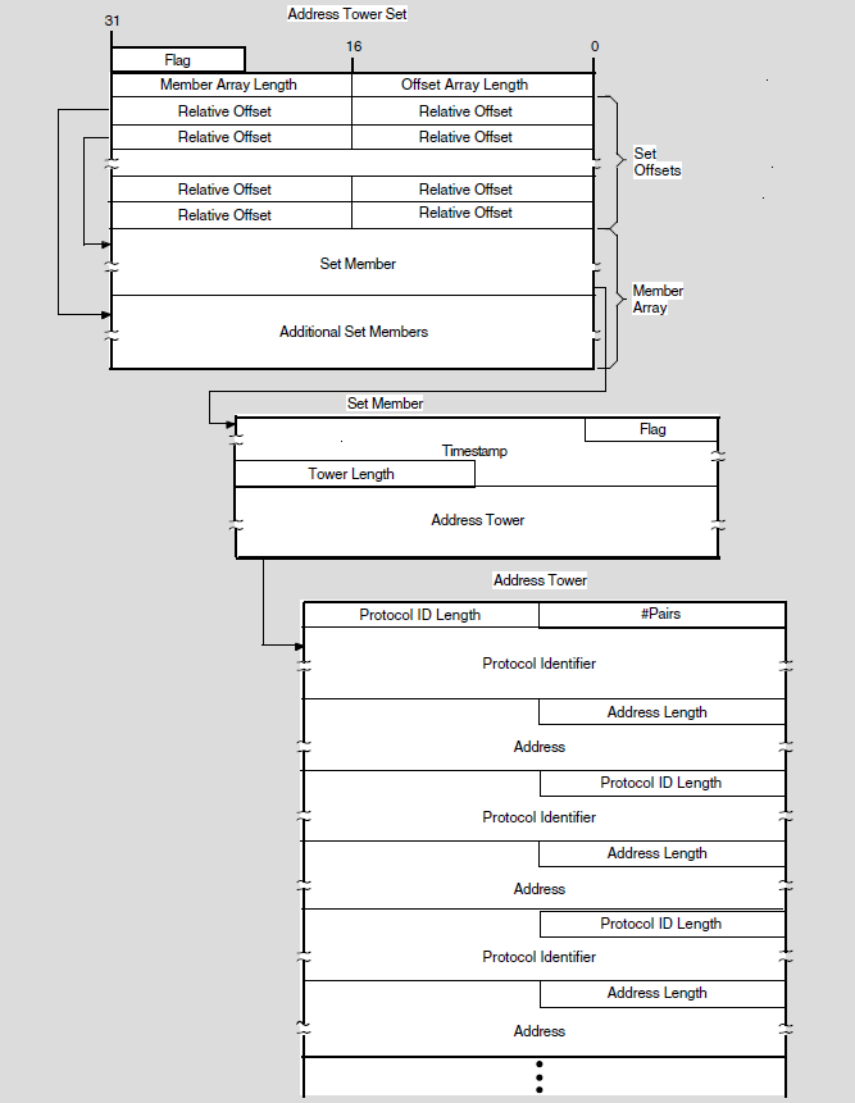
Set Member

The member itself, which includes the following:

Field	Length	Description
Flag	1 byte	Signifies whether the set member is present. If the low-order bit of the flag byte is set, the set member is present.
Timestamp	14 bytes	Time member was added to set.
Tower length	2 bytes	Length, in bytes, of the set member, not including the length of the *Tower Length* field.
Protocol tower		See *Section 2.13.7, "Protocol Tower Fields"*

Figure 2.5, "Protocol Tower Set" presents an illustration of a protocol tower set.

Figure 2.5. Protocol Tower Set



2.14. $IPC Function Codes to Manage Information

Table 2.2, "The $IPC System Service Function Codes to Manage Information" summarizes the function codes you need to manage information.

Table 2.2. The $IPC System Service Function Codes to Manage Information

Function Code	Function
IPC$K_FC_BACKTRANSLATE	Enable the calling process to get the node name or address information associated with the supplied node name or address.
IPC$K_FC_DEREGISTER_OBJECT	Enable the calling process to stop session from maintaining the DNA$Towers attribute for the specified object.
IPC$K_FC_ENUMERATE_LOCAL_TOWERS	Enable the calling process to get the set of local supported protocol towers up to the Session layer.

Function Code	Function
IPC$K_FC_GET_PORT_INFORMATION	Request information about a current connection.
IPC$K_FC_REGISTER_OBJECT	Enable the calling process to ask the Session Control layer to maintain the DNA$Towers attribute of the specified object in the DECdns namespace.
IPC$K_FC_RESOLVE_NAME	Enable the calling process to obtain the set of protocol towers mutually supported by the nodes on which the source and target tasks reside.
IPC$K_FC_VERIFY_NODENAME	Enable the calling process to verify that the node name associated with an inbound connection is valid.

2.15. Receiving Status and Error Reporting

When $IPC completes execution, it places the return status information in register 0 (R0). A successful return status indicates only that the request was queued successfully. $IPC places all completion status information in the interprocess communication context block (IPCB). For example, an $IPC system service transmit operation to a task might be successful (status return is IPC$_NORMAL) yet fail because the link was disconnected. (I/O status return is IPC$_THIRDPARTYABORT.) The return status codes shown in the following sections can be returned both in R0 and in the IPCB.

When DECnet-Plus for OpenVMS returns the status IPC$_NORMAL in the IPCB on a transmit request, it means that the transmit request was queued for transmission on the connection. It does not mean that the transmit request has been received or acknowledged by the remote task. The connection services of DECnet-Plus for OpenVMS provide the guaranteed delivery of transmitted messages to the remote system. If a message cannot be delivered, the user is notified by the disconnection of the connection. The DECnet-Plus for OpenVMS services cannot guarantee the delivery of data received on the remote system to the remote task. For example, the server task may not issue an IPC$K_FC_RECEIVE call. The cooperating tasks must use a common application protocol to ensure that data transmitted by the local task is received by the remote task.

Chapter 3. $IPC Reference Calls

The Interprocess Communication ($IPC) system service is the programming interface to the Session Control layer of the DECnet-Plus for OpenVMS software.

The $IPC system service completes asynchronously; that is, it returns to the caller immediately after queuing the request, without waiting for the operation to complete. The status returned to the caller indicates whether a request was completed successfully.

For synchronous completion, use the $IPCW call. The $IPCW service is identical to the $IPC service in every way except that $IPCW returns to the caller after the operation has completed.

A single system service provides a variety of interprocess communication functions. $IPC has two main parameters:

- A function code identifying the particular service to perform.
- **An Interprocess Communication Context Block (IPCB)**, a data structure that specifies required and optional information. The IPCB is used for both input and output operations. You pass information to and receive information from $IPC in a network item list.

For $IPC, you can synchronize completion (1) by specifying the **astdr** argument to have an AST routine execute when the I/O completes or (2) by calling the Synchronize ($SYNCH) service to await completion of the I/O operation. The $IPCW service completes synchronously; the $IPCW service is the best choice when synchronous completion is required. You can also synchronize by using OpenVMS event flags and the `efn` argument.

3.1. Arguments

Format

`$IPC [efn] ,func ,ipcb [,astadr] [,astprm]`

Returns

OpenVMS Usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Longword condition value. All system services return by immediate value a condition value in R0.

Arguments

efn	
OpenVMS Usage:	ef_number

type:	longword (unsigned)
access:	read only
mechanism:	by value

Number of event flag to be set when $IPC completes.

func	
OpenVMS Usage:	function_code
type:	longword (unsigned)
access:	read only
mechanism:	by value

Function code specifying the function that $IPC is to perform. The `func` argument is a longword containing this function code.

You can specify only one function code in a single call to $IPC. Most function codes require additional information to be passed with the call in the Interprocess Communication Context Block (IPCB) argument.

ipcb	
OpenVMS Usage:	ipc_context_block
type:	structure
access:	modify
mechanism:	by reference

Structure used to pass data to the $IPC system service, and for the $IPC service to return data to the caller. The *ipcb* argument is a longword containing the address of the IPCB structure.

You use the IPCB to pass required and optional information.

astadr	
OpenVMS Usage:	ast_procedure
type:	procedure entry mask
access:	call without stack unwinding
mechanism:	by reference

The AST service routine to be executed when the service completes. The **astadr** argument is the address of the entry point of the caller's AST completion routine.

astprm	
OpenVMS Usage:	user_arg
type:	longword (unsigned)
access:	read only

mechanism:	by value

AST parameter to be passed to the AST service routine. The **astprm** argument is a longword containing the AST parameter.

3.2. IPCB Fields

Table 3–1 through Table 3–19 list the required and optional fields for each of the following function codes.

IPCB$L_STATUS

Completion status of a request.

IPCB$L_STATUS1

Additional status for a completed request.

IPCB$L_FLAGS

Flags associated with a request. The IPCB defines the following symbolic names:

Symbolic Name	Description
IPCB$M_FLAGS_MULT	Signifies that the message is a single segment of a larger message.
IPCB$M_FLAGS_EXPEDITED	Sends or receives expedited data.
IPCB$M_FLAGS_NOPROXY	Overrides the Session Control layer's outgoing proxy characteristics.
IPCB$M_FLAGS_ AUTODISCONNECT	Allows the Transport layer to disconnect automatically if network connectivity has been lost.
IPCB$M_FLAGS_NOVERIFY_NODENAME	Bypasses node name verification.

IPCB$L_ASSOCIATIONID

The identification number of an association. The Session Control layer returns the association ID to a process when it successfully completes an IPC$K_FC_OPEN_ASSOCIATION operation.

IPCB$L_CONNECTIONID

The identification number of a connection. The Session Control layer returns the connection ID to a process when it successfully completes an IPC$K_FC_CONNECT_INITIATE operation or an IPC$K_FC_GET_CONNECTION operation.

IPCB$L_BUFFER_LENGTH

The length of the buffer specified in the IPCB$A_BUFFER field.

IPCB$L_RET_BUFFER_LENGTH

The length of the data that the Session Control layer returns in the IPCB$A_BUFFER field.

IPCB$A_BUFFER

The address of a buffer you supply for receipt or transmission of data or optional user data.

IPCB$L_RET_REPLY_LENGTH

The length of the data that the Session Control layer returns in the IPCB$A_REPLY_BUFFER field.

IPCB$A_REPLY_BUFFER

For connect initiate, the address of a buffer you supply to receive the acceptor reject data from an IPC$K_FC_ACCEPT operation or an IPC$K_FC_REJECT operation.

On a connect accept or connect reject function, the address of a buffer you supply for transmission of optional user data.

IPCB$L_REPLY_LENGTH

The length of the buffer specified in the IPCB$A_REPLY_BUFFER field.

IPCB$L_ASSOCIATION_CONTEXT

A longword that contains information you specify when opening an association to the Session Control layer.

IPCB$L_CONNECTION_CONTEXT

A longword that contains information you specify when identifying a target task or accepting a connection.

IPBC$L_EVENT_TYPE

Indicates which event is completing. The IPCB defines the following event type symbolic names:

- IPCB$K_FC_INCOMING_CONNECT
- IPCB$K_FC_INCOMING_DISCONNECT
- IPCB$K_FC_INCOMING_EXPEDITED

IPCB$Q_INPUTLST_DESC

The descriptor of the fields associated with an input item list, which is passed in network item list format. (Refer to *Section 3.3, "Network Item List Fields"* for more information.) Applications use the IPCB$Q_INPUTLST_DESC to pass required and optional information to $IPC.

The IPCB defines the following input item list fields:

Field	Description
IPCB$W_INPUTLST_LENGTH	Specifies the length of the input item list
IPCB$A_INPUTLST_POINTER	Specifies the address of the input item list

IPCB$Q_TEMPLATETLST_DESC

The descriptor of the fields associated with a template item list, which is passed in network item list format. (Refer to *Section 3.3, "Network Item List Fields"* for more information.) Applications use the IPCB$Q_TEMPLATELST_DESC to specify which items $IPC should return to them (in the output item list) upon completion of the request.

The IPCB defines the following input item list fields:

Field	Description
IPCB$W_TEMPLATETLST_LENGTH	Specifies the length of the input item list
IPCB$A_TEMPLATETLST_POINTER	Specifies the address of the input item list

IPCB$Q_OUTPUTLST_DESC

The descriptor of the fields associated with an output item list, which is passed in network item list format. (Refer to *Section 3.3, "Network Item List Fields"* for more information.) Applications use the IPCB$Q_OUTPUTLST_DESC to pass the address of a buffer that is to receive the output item list upon completion of the $IPC request.

The IPCB defines the following output item list fields:

Field	Description
IPCB$W_OUTPUTLST_LENGTH	Specifies the length of the output item list
IPCB$A_OUTPUTLST_POINTER	Specifies the address of the output item list

IPCB$W_RET_OUTPUTLST_LENGTH

The length of the output item list upon completion of IPC service.

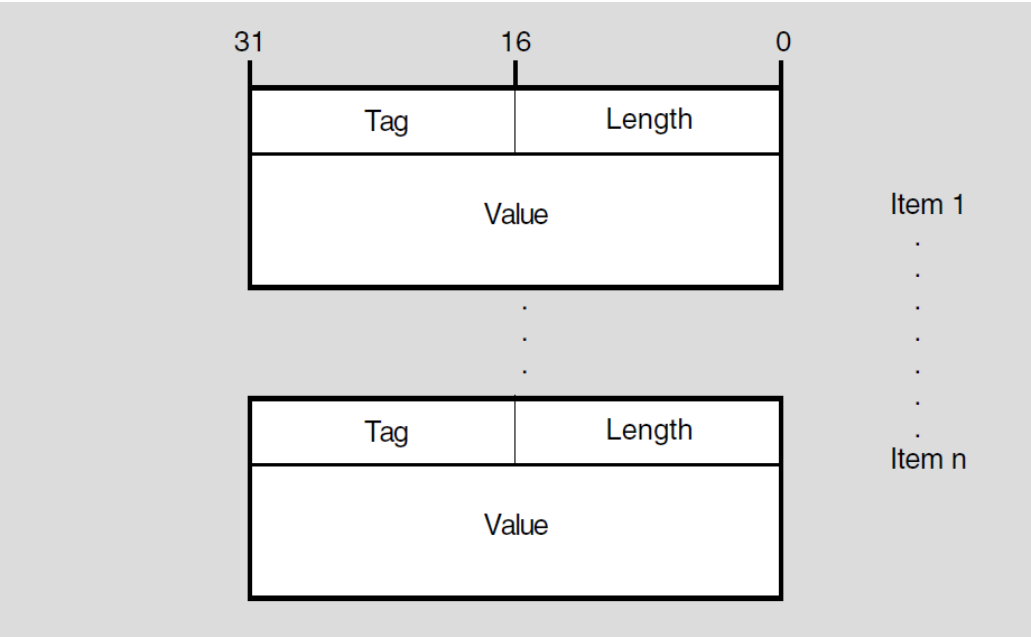
3.3. Network Item List Fields

Network Item Lists

A network item list is a contiguous block of memory containing one or more items, with each item in "tag-length-value" format. The information passed in the network item list affects the action designated by the *func* argument. You pass network item lists in the IPCB$Q_INPUTLST_DESC and the IPCB$Q_TEMPLATELST_DESC fields of the IPCB. You receive information in the IPCB$Q_OUTPUTLST_DESC fields of the IPCB.

Figure 3.1, "Network Item List" shows how the items are formatted:

Figure 3.1. Network Item List



Tag

A 2-byte code indicating the nature and format of the information to be passed. Each item code has a symbolic name; these symbolic names have the format NET$K_TAG_code and are defined in $IPC ITEM CODES section.

Length

The 2-byte length of the body of the item in bytes, including both length and tag fields. For example, an item whose value is a longword has a length field of 8 bytes. Items can be fixed or variable length. All items include the length field.

Data

The actual value of the item in bytes. A null value is allowed. If you use a null value, the default length of the entire item entry is 4 bytes.

Note

If you use a tag that is not valid or omit a required tag, the $IPC operation will fail. The return status code specifying a failure appears in the IPCB$L_STATUS field of the IPCB.

3.4. Function Codes

This section provides a summary of function codes that are valid for the *func* argument. The tables list the function codes, valid input and output IPCB fields, and valid item codes for input and template item lists. The function codes appear in the tables without the IPC$K_FC_prefix. For example, IPC$K_FC_ABORT_CONNECTION appears in the table as ABORT_CONNECTION.

3.4.1. IPC$K_FC_ABORT_CONNECTION

This request terminates the connection. Any pending messages will be lost.

Table 3.1, "ABORT_CONNECTION" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Table 3.1. ABORT_CONNECTION

IPCB Fields	**Item Codes**
Input	**Input Item List**
You must specify both: IPCB$L_ASSOCIATIONID and IPCB$L_CONNECTIONID	None
You can specify: IPCB$A_BUFFER and IPCB$L_BUFFER_LENGTH	None
Output	**Template Item List**
IPCB$L_STATUS	None

3.4.2. IPC$K_FC_BACKTRANSLATE

This request enables the calling process to obtain the node name associated with the supplied address and vice versa. You must supply one data item in the input item list and any combination of items in the template list. All data items requested in the template list are returned in the output item list if available. Items not available are returned as null items in the output item list. If no items are available, an error status is returned.

Table 3.2, "BACKTRANSLATE_ADDRESS" contains possible input and output parameters for the IPC $K_FC_BACKTRANSLATE function.

Table 3.2. BACKTRANSLATE_ADDRESS

IPCB Fields	**Item Codes**
Input	**Input Item List**

You must specify:
• IPCB$Q_INPUTLST_DESC
  ○ IPCB$W_INPUTLST_LENGTH
  ○ IPCB$A_INPUTLST_POINTER
• IPCB$Q_TEMPLATELST_DESC
  ○ IPCB$W_TEMPLATELST_LENGTH
  ○ IPCB$A_TEMPLATELST_POINTER
• IPCB$Q_OUTPUTLST_DESC
  ○ IPCB$W_OUTPUTLST_LENGTH
  ○ IPCB$A_OUTPUTLST_POINTER
	You must specify one:
	• NET$K_TAG_NODESYNONYM
	• NET$K_TAG_NODENAME_INT
	• NET$K_TAG_IV_ADDRESS
	• NET$K_TAG_DESTINATIONADDRESS
	• NET$K_TAG_NODENAME

IPCB Fields	Item Codes
Output	**Template Item List**
IPCB$L_STATUS IPB$L_STATUS1 IPCB$L_RET_OUTPUTLST_LENGTH	You must specify one: • NET$K_TAG_NODESYNONYM • NET$K_TAG_NODENAME • NET$K_TAG_NODENAME_INT • NET$K_TAG_COMPRESSEDNAME • NET$K_TAG_IV_ADDRESS • NET$K_TAG_DESTINATIONADDRESS • NET$K_TAG_DESTINATIONTOWERSET

3.4.3. IPC$K_FC_CLOSE_ASSOCIATION

Closing an association terminates all pending I/O and disconnects all connections for the specified association. No connection can be made to or from the application after it closes the association.

Table 3.3, "CLOSE_ASSOCIATION" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Table 3.3. CLOSE_ASSOCIATION

IPCB Fields	Item Codes
Input	**Input Item List**
You must specify: IPCB$L_ASSOCIATIONID	None
Output	**Template Item List**
IPCB$L_STATUS	None

3.4.4. IPC$K_FC_CONNECT_ACCEPT

This request accepts a received connection request. You can request that DECnet software automatically disconnect the Transport layer connection if it appears that network connectivity has been lost. You can also pass a longword of information you specify to the Session Control layer.

You can return up to 16 bytes of user data with the confirmation of an IPC$K_FC_CONNECT_INITIATE request by specifying the IPCB$A_REPLY_BUFFER and the IPCB$L_BUFFER_LENGTH fields in the IPCB.

Table 3.4, "CONNECT_ACCEPT" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Table 3.4. CONNECT_ACCEPT

IPCB Fields	Item Codes
Input	**Input Item List**
You must specify: IPCB$L_ASSOCIATIONID and IPCB$L_CONNECTIONID	None

IPCB Fields	Item Codes
You can specify: • IPCB$A_REPLY_BUFFER • IPCB$L_REPLY_LENGTH • IPCB$L_CONNECTION_CONTEXT • IPCB$L_FLAGS ○ IPCB$M_FLAGS_AUTODISCONNECT	
Output	**Template Item List**
IPCB$L_STATUS	None

3.4.5. IPC$K_FC_CONNECT_INITIATE

This call requests a connection to the target task. It enables you to do the following:

- Identify the target task.
- Send the request with outgoing proxy disabled.
- Enable the Transport layer to disconnect automatically if network connectivity is lost.
- Specify optional access verification information.
- Request source and target information.
- Pass a longword of information you specify to the Session Control layer.
- Send optional user data.

After successful completion of this request, the Session Control layer returns a connection identification to the calling task in the IPCB.

You can transmit up to 16 bytes of user data with the connection request. The target task can return up to 16 bytes of user data (either to accept or reject the connection).

Table 3.5, "CONNECT_INITIATE" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Table 3.5. CONNECT_INITIATE

IPCB Fields	Item Codes
Input	**Input Item List**

You must specify:
• IPCB$L_ASSOCIATIONID
• IPCB$Q_INPUTLST_DESC
  ○ IPCB$W_INPUTLST_LENGTH
  ○ IPCB$A_INPUTLST_POINTER
	You must specify one:
	• NET$K_TAG_DESTINATIONTOWER
	• NET$K_TAG_NODENAME
	• NET$K_TAG_NODENAME_INT
	• NET$K_TAG_DNSOBJECTNAME

IPCB Fields	Item Codes
	• NET$K_TAG_DNSOBJECTNAME_INT

You can specify:
• IPCB$L_CONNECTION_CONTEXT
• IPCB$Q_TEMPLATELST_DESC
  ○ IPCB$W_TEMPLATELST_LENGTH
  ○ IPCB$A_TEMPLATELST_POINTER
	If you specify NET$K_TAG_DESTINATIONTOWER, you can also specify NET$K_TAG_SOURCETOWER

If you specified a template item list, you must specify:
• IPCB$Q_OUTPUTLST_DESC
  ○ IPCB$W_OUTPUTLST_LENGTH
  ○ IPCB$A_OUTPUTLST_POINTER
	If you specify NET$K_TAG_NODENAME or NET$K_TAG_NODENAME_INT, you must also specify one:
	• NET$K_TAG_ENDUSERID_NAME
	• NET$K_TAG_ENDUSERID_NUMBER
	• NET$K_TAG_ENDUSERID_TASK

You can specify:
• IPCB$A_BUFFER
• IPCB$L_BUFFER_LENGTH
• IPCB$L_FLAGS
  ○ IPCB$M_FLAGS_NOPROXY
  ○ IPCB$M_FLAGS_AUTODISCONNECT
	If you specify NET$K_TAG_DNSOBJECTNAME_INT, you can also specify NET$K_TAG_USERTOWER

If you receive optional user data, you must specify: IPCB$A_REPLY_BUFFER or IPCB$L_REPLY_LENGTH	You can specify: • NET$K_TAG_CLIENTNAME • NET$K_TAG_DESTINATIONACCOUNT • NET$K_TAG_DESTINATIONPASSWORD • NET$K_TAG_DESTINATIONUSER
Output IPCB$L_CONNECTIONID IPCB$L_STATUS	**Template Item List** NET$K_TAG_DESTINATIONTOWER NET$K_TAG_SOURCETOWER
If you receive an output item list: IPCB$W_RET_OUTPUTLST_LENGTH	
If you receive optional user data: IPCB$L_RET_BUFFER_LENGTH	

IPCB Fields	Item Codes
You can specify: NET$K_TAG_DESTINATIONTOWER or NET$K_TAG_SOURCETOWER	

3.4.6. IPC$K_FC_CONNECT_REJECT

This request rejects a received connect request. You can return up to 16 bytes of optional user data with the rejection.

Table 3.6, "CONNECT_REJECT" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Table 3.6. CONNECT_REJECT

IPCB Fields	Item Codes
Input	**Input Item List**
You must specify: IPCB$L_ASSOCIATIONID or IPCB$L_CONNECTIONID You can specify: IPCB$A_REPLY_BUFFER or IPCB$L_REPLY_LENGTH	None
Output	**Template Item List**
IPCB$L_STATUS	None

3.4.7. IPC$K_FC_DEREGISTER_OBJECT

This request enables the calling process to notify the Session Control layer to stop maintaining the DNA $Towers attribute for the specified object in the name service.

Table 3.7, "DEREGISTER_OBJECT" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Table 3.7. DEREGISTER_OBJECT

IPCB Fields	Item Codes
Input	**Input Item List**
IPCB$Q_INPUTLST_DESC IPCB$W_INPUTLST_LENGTH IPCB$A_INPUTLST_POINTER	You must specify: NET$K_TAG_DNSOBJECTNAME or NET$K_TAG_DNSOBJECTNAME_INT You can specify: NET$K_TAG_USERTOWER
Output	**Template Item List**
IPCB$L_STATUS	None

3.4.8. IPC$K_FC_DISCONNECT_CONNECTION

This request synchronously disconnects the connection. The DECnet-Plus for OpenVMS software transmits and acknowledges all pending messages to the remote node before it disconnects the

connection. It does not, however, guarantee that the message reaches the target task. For instance, the target task may not issue a RECEIVE. You can return up to 16 bytes of user data.

Table 3.8, "DISCONNECT_CONNECTION" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Table 3.8. DISCONNECT_CONNECTION

IPCB Fields	**Item Codes**
Input	**Input Item List**
You must specify: IPCB$L_ASSOCIATIONID or IPCB$L_CONNECTIONID You can specify: IPCB$A_BUFFER or IPCB$L_BUFFER_LENGTH	None
Output	**Template Item List**
IPCB$L_STATUS	None

3.4.9. IPC$K_FC_ENUMERATE_LOCAL_TOWERS

This request enables the calling process to obtain information about locally supported protocols and addresses from the Network layer to the Session Control layer. This information is in protocol tower set format.

Table 3.9, "ENUMERATE_LOCAL_TOWERS" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Table 3.9. ENUMERATE_LOCAL_TOWERS

IPCB Fields	**Item Codes**
Input	**Input Item List**
If you specify a template item list, you must specify: ● IPCB$L_TEMPLATELST_DESC ○ IPCB$W_TEMPLATELST_LENGTH ○ IPCB$A_TEMPLATELST_POINTER ● IPCB$L_OUTPUTLST_DESC ○ IPCB$W_OUTPUTLST_LENGTH ○ IPCB$A_OUTPUTLST_POINTER	None
Output	**Template Item List**
IPCB$L_STATUS If you receive an output item list: IPCB$W_RET_OUTPUTLST_LENGTH	You can specify: NET$K_TAG_SOURCETOWERSET

3.4.10. IPC$K_FC_GET_CONNECTION

This request binds an incoming connection request with the calling process. The call completes when the Session Control layer receives a connection request that satisfies the parameters of the calling process. This call binds a request from an IPC$K_FC_CONNECT_INITIATE operation.

A process that receives multiple connection requests must reissue the IPC$K_FC_GET_CONNECTION request to obtain each connection. The process must also accept or reject each individual connection. You can receive up to 16 bytes of optional user data sent by a caller.

When this request completes successfully, the Session Control layer returns a connection identification to the calling task in the IPCB.

Table 3.10, "GET_CONNECTION" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Table 3.10. GET_CONNECTION

IPCB Fields	**Item Codes**
Input	**Input Item List**
You must specify: IPCB$L_ASSOCIATIONID	You can specify: NET$K_TAG_CLIENTNAME

If you specified an input item list:
• IPCB$Q_INPUTLST_DESC
  ○ IPCB$W_INPUTLST_LENGTH
  ○ IPCB$A_INPUTLST_POINTER	

If you specified a template item list:
• IPCB$Q_TEMPLATELST_DESC
  ○ IPCB$W_TEMPLATELST_LENGTH
  ○ IPCB$A_TEMPLATELST_POINTER
• IPCB$Q_OUTPUTLST_DESC
  ○ IPCB$W_OUTPUTLST_LENGTH
  ○ IPCB$A_OUTPUTLST_POINTER	

You can specify: IPCB$A_BUFFER or IPCB$L_BUFFER_LENGTH	
Output	**Template Item List**
IPCB$L_CONNECTIONID IPCB$L_STATUS If you receive an output item list: IPCB$W_RET_OUTPUTLST_LENGTH	You can specify: • NET$K_TAG_NODESYNONYM • NET$K_TAG_NODENAME • NET$K_TAG_NODENAME_INT • NET$K_TAG_ENDUSERID_NUMBER

IPCB Fields	Item Codes
If you receive optional user data: IPCB$L_RET_BUFFER_LENGTH	• NET$K_TAG_ENDUSERID_NAME • NET$K_TAG_ENDUSERID_TASK • NET$K_TAG_DESTINATIONUSER • NET$K_TAG_DESTINATIONACCOUNT • NET$K_TAG_SOURCEUSER • NET$K_TAG_SOURCENAME • NET$K_TAG_SOURCENUMBER • NET$K_TAG_SOURCEADDRESS • NET$K_TAG_SOURCETOWER

3.4.11. IPC$K_FC_GET_PORT_INFORMATION

This call requests information about a specified connection.

Table 3.11, "GET_PORT_INFORMATION" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Table 3.11. GET_PORT_INFORMATION

IPCB Fields	Item Codes
Input	**Input Item List**
You must specify: IPCB$L_ASSOCIATIONID or IPCB$L_CONNECTIONID	None

You must specify:
• IPCB$L_TEMPLATELST_DESC
  ○ IPCB$L_TEMPLATELST_LENGTH
  ○ IPCB$A_TEMPLATELST_POINTER
• IPCB$L_OUTPUTLST_DESC
  ○ IPCB$L_OUTPUTLST_LENGTH
  ○ IPCB$A_OUTPUTLST_POINTER	

Output	**Template Item List**
IPCB$L_STATUS If you receive an output item list: IPCB$W_RET_OUTPUTLST_LENGTH	NET$K_TAG_CLIENTNAME NET$K_TAG_NODESYNONYM NET$K_TAG_NODENAME NET$K_TAG_NODENAME_INT

IPCB Fields	Item Codes
	NET$K_TAG_SOURCEADDRESS NET$K_TAG_DESTINATIONADDRESS NET$K_TAG_DIRECTION

3.4.12. IPC$K_FC_OPEN_ASSOCIATION

This request opens an association between an application and the Session Control layer of a DECnet-Plus for OpenVMS network.

If the application is a server application, the association indicates to the Session Control layer that the application is willing to accept incoming connections. The association also indicates whether user authentication is required on an incoming connection request to a server. If the application is a client application, the association indicates that the application will be issuing requests for connection.

When a task opens an association, the task can indicate to the Session Control layer whether it should stall transmission of normal data messages if it receives an expedited data message before the normal data message.

Note

If the application is defined in the application database, the database parameters will override the parameters specified in this call. Use this call to augment the values in the application database or to open an association for private applications (applications that are not registered in the application database). For more information on registering applications in the application database, refer to *VSI DECnet-Plus for OpenVMS Network Management Guide*.

When opening the association, a server task should do the following:

- Declare the application name.
- Indicate whether node name verification is required.

The Session Control layer passes an association ID (by means of the IPCB) to the task if the association request is successful. The NET$K_TAG_EVENTMASK item code enables events for pick up. This is a longword flags field, valid only for the IPC$K_FC_OPEN_ASSOCIATION function code.

This longword is interpreted by IPC as a bit mask, and each bit set enables a specific type of notification. The bits used by IPC are defined as:

NET$V_EVENT_INCOMING

NET$V_EVENT_EXPEDITED

NET$V_EVENT_DISCONNECTS

Table 3.12, "OPEN_ASSOCIATION" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Table 3.12. OPEN_ASSOCIATION

IPCB Fields	Item Codes
Input	**Input Item List**

IPCB Fields	Item Codes

You can specify:
• IPCB$L_ASSOCIATION_CONTEXT
• IPCB$L_FLAGS
  ○ IPCB $M_FLAGS_NOVERIFY_NODENAME
If you specified an input item list:
• IPCB$Q_INPUTLST_DESC
  ○ IPCB$W_INPUTLST_LENGTH
  ○ IPCB$A_INPUTLST_POINTER
	A server task must specify one:
	• NET$K_TAG_ENDUSERID_NAME
	• NET$K_TAG_ENDUSERID_NUMBER
	• NET$K_TAG_ENDUSERID_TASK
	• NET$K_TAG_EVENTMASK

Output	**Template Item List**
IPCB$L_ASSOCIATIONID IPCB$L_STATUS	None

3.4.13. IPC$K_FC_RECEIVE

This request receives normal and expedited data from a target task. You receive normal data messages by default. To receive expedited data, use the IPCB$M_FLAGS_EXPEDITED function modifier. If the application expects expedited data, it is important that there always be a IPC$K_FC_RECEIVE call with the IPCB$L_FLAGS_EXPEDITED flag set outstanding on all connections under the association.

This call also enables you to receive segmented data. If you post a receive buffer with the IPCB $M_FLAGS_MULT flag set, the DECnet-Plus for OpenVMS software fills the buffer with as much data as will fit and then fills the buffer with an alternate success status (IPC$_MOREDATA). If you post another IPC$K_FC_RECEIVE operation with the IPCB$M_FLAGS_MULT flag set, the DECnet-Plus for OpenVMS software gives you the next segment of the message. If you don't set the IPCB $M_FLAGS_MULT flag and receive a message that is larger than the receive buffer, you lose the end of the message (IPC$_DATAOVERRUN).

You cannot set the IPCB$M_FLAGS_EXPEDITED flag in combination with the IPCB $M_FLAGS_MULT flag.

Table 3.13, "RECEIVE" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Table 3.13. RECEIVE

IPCB Fields	Item Codes
Input	**Input Item List**
You must specify: • IPCB$L_ASSOCIATIONID • IPCB$L_CONNECTIONID • IPCB$A_BUFFER	None

IPCB Fields	Item Codes

• IPCB$L_BUFFER_LENGTH
• IPCB$L_FLAGS
  ○ IPCB$M_FLAGS_EXPEDITED
  ○ IPCB$M_FLAGS_MULT	

Output	**Template Item List**
IPCB$L_STATUS IPCB$L_RET_BUFFER_LENGTH	None

3.4.14. IPC$K_FC_RECEIVE_EVENT

This request receives asynchronous event notification from the DECnet network.

Table 3.14, "RECEIVE_EVENT" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Table 3.14. RECEIVE_EVENT

IPCB Fields	Item Codes
Input	**Input Item List**
You must specify: • IPCB$L_ASSOCIATIONID • IPCB$A_BUFFER • IPCB$L_BUFFER_LENGTH	None
Output	**Template Item List**
IPCB$L_STATUS IPCB$L_RET_BUFFER_LENGTH IPCB$L_EVENT_TYPE	None

3.4.15. IPC$K_FC_REGISTER_OBJECT

This request enables the calling process to request that the Session Control Layer maintain the DNA $Tower attribute of the specified object in the DECdns namespace. Whenever Session detects that the underlying protocol towers supporting Session on the local node have changed, it will update the DNA $Towers attributes in each of the DECdns objects that have been registered with DECdns. You can also request that the Session Control layer maintain user tower information.

You must specify the NET$K_TAG_OBJECTNAME item code in the input item list. You can specify the NET$K_TAG_USERTOWER item code as well.

Table 3.15, "REGISTER_OBJECT" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Note

This request has no effect on DECdns clerk nodes using the Local Naming Option (LNO) and returns an error message.

Table 3.15. REGISTER_OBJECT

IPCB Fields	Item Codes
Input	**Input Item List**

You must specify:
• IPCB$Q_INPUTLST_DESC
  ○ IPCB$W_INPUTLST_LENGTH
  ○ IPCB$A_INPUTLST_POINTER
	You must specify: NET $K_TAG_DNSOBJECTNAME_INT
	
	You can specify: NET$K_TAG_USERTOWER

Output	**Template Item List**
IPCB$L_STATUS	None

3.4.16. IPC$K_FC_RESOLVE_NAME

This request enables the calling process to obtain the set of address towers from DECdns that are supported mutually by the source and target nodes for a specified application object. You can also request the protocol tower elements for the layers above the Session Control layer in the request.

Table 3.16, "RESOLVE_NAME" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Table 3.16. RESOLVE_NAME

IPCB Fields	Item Codes
Input	**Input Item List**

You must specify:
• IPCB$Q_INPUTLST_DESC
  ○ IPCB$W_INPUTLST_LENGTH
  ○ IPCB$A_INPUTLST_POINTER
 	
You can specify:
• IPCB$L_TEMPLATELST_DESC
  ○ IPCB$L_TEMPLATELST_LENGTH
  ○ IPCB$A_TEMPLATELST_POINTER
• IPCB$L_OUTPUTLST_DESC
  ○ IPCB$L_OUTPUTLST_LENGTH
  ○ IPCB$A_OUTPUTLST_POINTER
	You must specify: NET $K_TAG_DNSOBJECTNAME_INT or NET $K_TAG_DNSOBJECTNAME
	
	You can specify: NET$K_TAG_USERTOWER

IPCB Fields	Item Codes
Output	**Template Item List**
IPCB$L_STATUS If you receive an output item list: IPCB $W_RET_OUTPUTLST_LENGTH	You can specify: NET $K_TAG_DESTINATIONTOWERSET or NET $K_TAG_SOURCETOWERSET

3.4.17. IPC$K_FC_SHUT_ASSOCIATION

This request stops any further connections from being accepted by this association with the Session Control layer. All existing connections continue to work until discontinued or aborted by the application.

Table 3.17, "SHUT_ASSOCIATION" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Table 3.17. SHUT_ASSOCIATION

IPCB Fields	Item Codes
Input	**Input Item List**
You must specify: IPCB$L_ASSOCIATIONID	None
Output	**Template Item List**
IPCB$L_STATUS	None

3.4.18. IPC$K_FC_TRANSMIT

This request sends normal and expedited data to the target application. You can also send segmented data with this request. The default is Normal Data Transmit.

This request, along with the IPCB$M_FLAGS_EXPEDITED function modifier, transmits expedited data to the target task.

The IPCB$M_FLAGS_MULT function modifier enables you to transmit segmented data. You can segment the message into n number of segments. Send the first n - 1 messages with the IPCB $M_FLAGS_MULT flag set. Send the final segment without the flag, signaling to the DECnet network that this is the end of this segmented message. The receiver can either specify a single receive operation with a buffer large enough to hold all the segmented messages or issue numerous IPC$K_FC_RECEIVE requests with the IPCB$M_FLAGS_MULT flag set.

You cannot set the IPCB$M_FLAGS_EXPEDITED flag in combination with the IPCB $M_FLAGS_MULT flag.

Table 3.18, "TRANSMIT" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Table 3.18. TRANSMIT

IPCB Fields	Item Codes
Input	**Input Item List**
You must specify: IPCB$L_ASSOCIATIONID or IPCB$L_CONNECTIONID	None

IPCB Fields	Item Codes

You must specify: IPCB$A_BUFFER or IPCB $L_BUFFER_LENGTH
 	
You can specify:
 	
• IPCB$L_FLAGS
  ○ IPCB$_FLAGS_EXPEDITED
  ○ IPCB$_FLAGS_MULT	

Output	**Template Item List**
IPCB$L_STATUS	None

3.4.19. IPC$K_FC_VERIFY_NODENAME

This request verifies that the address information associated with the node name you specify is consistent with the address information for that node object in the DNA$Towers attribute.

Table 3.19, "VERIFY_NODENAME" contains information on optional and required IPCB fields, input item list codes, and template item list codes.

Table 3.19. VERIFY_NODENAME

IPCB Fields	Item Codes
Input	**Input Item List**

You must specify:
 	
• IPCB$L_ASSOCIATIONID
• IPCB$L_CONNECTIONID
• IPCB$Q_INPUTLST_DESC
  ○ IPCB$W_INPUTLST_LENGTH
  ○ IPCB$A_INPUTLST_POINTER
	You can specify:
	
	• NET$K_TAG_NODENAME
	• NET$K_TAG_NODENAME_INT
	• NET$K_TAG_NODESYNONYM
	• NET$K_TAG_SOURCETOWER

Output	**Template Item List**
IPCB$L_STATUS	None

3.5. Item Codes

This section describes the $IPC item codes.

NET$K_TAG_DESTINATIONTOWER

The NET$K_TAG_DESTINATIONTOWER item code specifies the protocol tower that defines the complete destination to the remote node.

The value field of this network item list entry is in protocol tower format. See *Section 2.13.7, "Protocol Tower Fields"* for Protocol Tower Field parameters.

Valid for the IPC$K_FC_CONNECT_INITIATE and IPC$K_FC_GET_CONNECTION function codes.

NET$K_TAG_CLIENTNAME

The NET$K_TAG_CLIENTNAME item code identifies the network management name assigned to the Session Control layer connection.

The value field of this network item list entry is an ASCII character string.

Valid for the IPC$K_FC_CONNECT_INITIATE and IPC$K_FC_GET_PORT_INFORMATION function codes.

NET$K_TAG_DESTINATIONADDRESS

The NET$K_TAG_DESTINATIONADDRESS item code indicates the network service access point (NSAP) address of the target node. This address contains up to 20 bytes of binary data.

The value field of this network item list entry is binary data.

Valid for the IPC$K_FC_BACKTRANSLATE function code.

NET$K_TAG_DESTINATIONACCOUNT

The NET$K_TAG_DESTINATIONACCOUNT item code identifies the destination account and is used for access verification.

The value field of this network item list entry is an ASCII character string.

Valid for the IPC$K_FC_CONNECT_INITIATE, IPC$K_FC_GET_CONNECTION, and IPC $K_FC_GET_PORT_INFORMATION function codes.

NET$K_TAG_DESTINATIONPASSWORD

The NET$K_TAG_DESTINATIONPASSWORD item code specifies the access verification password.

The value field of this network item list entry is an ASCII character string.

Valid for the IPC$K_FC_CONNECT_INITIATE function code.

NET$K_TAG_DESTINATIONUSER

The NET$K_TAG_DESTINATIONUSER item code identifies the destination user name and is used for access verification.

The value field of this network item list entry is an ASCII character string.

Valid for the IPC$K_FC_CONNECT_INITIATE and IPC$K_FC_GET_PORT_INFORMATION function codes.

NET$K_TAG_DESTINATIONTOWERSET

The NET$K_TAG_DESTTOWERSET item code describes the destination protocol towers mutually supported by the source and target nodes.

The value field is in protocol tower set format. See *Section 2.13.8, "Protocol Tower Set Fields"* for protocol tower set field parameters.

NET$K_TAG_DIRECTION

The NET$K_TAG_DIRECTION item code indicates whether the Session Control layerport is opened to initiate an outgoing connection, to receive an incoming connection, or is unknown.

The value field of this network item list entry is a byte value.

Valid for the IPC$K_FC_GET_PORT_INFORMATION function codes.

NET$K_TAG_DNSOBJECTNAME

The NET$K_TAG_OBJECTNAME item code is the DECdns full name string (external name) of the target application object.

The value field of this network item list entry is an ASCII character string.

Valid for the IPC$K_FC_CONNECT_INITIATE and IPC$K_FC_RESOLVE_NAME function codes.

NET$K_TAG_DNSOBJECTNAME_INT

The NET$K_TAG_OBJECTNAME_INT item code is the DECdns opaque full name of the target application object.

The value field of this network item list entry is in DECdns opaque full name format.

Valid for the IPC$K_FC_CONNECT_INITIATE, IPC$K_FC_DEREGISTER_OBJECT, IPC $K_FC_REGISTER_OBJECT, and IPC$K_FC_RESOLVE_NAME function codes.

NET$K_TAG_ENDUSERID_NAME

The NET$K_TAG_ENDUSERID_NAME item code specifies the end user identification information as a DECdns opaque full name.

The value field of this network item list entry is in DECdns opaque full name format.

Valid for the IPC$K_FC_CONNECT_INITIATE, IPC$K_FC_GET_PORT_INFORMATION, and IPC $K_FC_OPEN_ASSOCIATION function codes.

NET$K_TAG_ENDUSERID_NUMBER

The NET$K_TAG_ENDUSERID_NUMBER specifies the numeric identification of the target application. This corresponds to the Phase IV object number.

The value field of this network item list entry is a ASCII character string.

Valid for the IPC$K_FC_CONNECT_INITIATE, IPC$K_FC_GET_PORT_INFORMATION, and IPC $K_FC_OPEN_ASSOCIATION function codes.

NET$K_TAG_ENDUSERID_TASK

The NET$K_TAG_ENDUSERID_TASK item code specifies the string identification of the target application. This corresponds to the Phase IV object task name.

The value field of this network item list entry is an ASCII character string.

Valid for the IPC$K_FC_CONNECT_INITIATE, IPC$K_FC_GET_PORT_INFORMATION, and IPC $K_FC_OPEN_ASSOCIATION function codes.

NET$K_TAG_IV_ADDRESS

The NET$K_TAG_IV_ADDRESS item code specifies the Phase IV 16-bit address.

The value field of this network item list entry is binary data.

Valid for the IPC$K_FC_BACKTRANSLATE function code.

NET$K_TAG_NODENAME

The NET$K_TAG_NODENAME item code specifies the name of the target node. You can specify this item code as either **0** for the local node, Phase IV style address, DECdns full name string, or node synonym.

The value field of this network item list entry is an ASCII character string.

Valid for the IPC$K_FC_CONNECT_INITIATE, IPC$K_FC_GET_PORT_INFORMATION, IPC $K_FC_BACK_TRANSLATE_ADDRESS and IPC$K_FC_VERIFY_NODENAME function codes.

NET$K_TAG_NODENAME_INT

The NET$K_TAG_NODENAME_INT item code specifies the object name of the target node. You must specify this item code as a DECdns opaque full name.

The value field of this network item list entry is in DECdns opaque full name format.

Valid for the IPC$K_FC_CONNECT_INITIATE, IPC$K_FC_GET_PORT_INFORMATION, IPC $K_FC_GET_CONNECTION, and IPC$K_FC_BACK_TRANSLATE_ADDRESS function codes.

NET$K_TAG_NODESYNONYM

The NET$K_TAG_NODESYNONYM item code specifies the synonym name of the target node. You must specify this item code as a Phase IV 6-character node name.

The value field of this network item list entry is an ASCII character string.

Valid for the IPC$K_FC_GET_PORT_INFORMATION and IPC $K_FC_BACK_TRANSLATE_ADDRESS function codes.

NET$K_TAG_SOURCENAME

The NET$K_TAG_SOURCENAME item code specifies the Phase V application object name of the source application as a DECdns opaque full name.

The value field of this network item list entry is in DECdns opaque full name format.

Valid for the IPC$K_FC_GET_PORT_INFORMATION function codes.

NET$K_TAG_SOURCENUMBER

The NET$K_TAG_SOURCENUMBER specifies the numeric identification of the source application. This corresponds to the Phase IV object number.

The value field of this network item list entry is an ASCII character string.

Valid for the IPC$K_FC_GET_PORT_INFORMATION function codes.

NET$K_TAG_SOURCETOWER

The NET$K_TAG_SOURCETOWER item code specifies the protocol tower that the source node used to make the connection.

The value field of this network item list entry is in protocol tower format.

See *Section 2.13.7, "Protocol Tower Fields"* for protocol tower field parameters.

Valid for the IPC$K_FC_CONNECT_INITIATE function code.

NET$K_TAG_SOURCETOWERSET

The NET$K_TAG_SOURCETOWERSET item code signifies the tower set structure that describes the source protocol towers that are mutually supported by the source and target nodes for a specified application object.

The value field of this network item list entry is in protocol tower set format.

See *Section 2.13.8, "Protocol Tower Set Fields"* for protocol tower set field parameters.

Valid for the IPC$K_FC_ENUMERATE_LOCAL_TOWERS and IPC$K_FC_RESOLVE_NAME function codes.

NET$K_TAG_SOURCEUIC

The source user identification code (UIC) identifies the UIC of a process. It consists of 16 bits of UIC group and 16 bits of member. This is stored in binary.

Valid for IPC$K_FC_CONNECT_INITIATE and IPC$K_FC_GET_CONNECTION.

NET$K_TAG_SOURCEUSER

The NET$K_TAG_SOURCEUSER item code identifies the source of a connection request on a IPC $K_FC_GET_CONNECTION operation. It returns the string identification of the source application. This corresponds to the Phase IVobject task name.

The value field of this network item list entry is an ASCII character string.

Valid for the IPC$K_FC_GET_CONNECTION function code.

NET$K_TAG_USERTOWER

The NET$K_TAG_USERTOWER item code identifies the protocol tower structure that specifies the protocol tower elements above the Session Control layer to be used in the application

selection. This item code is used to qualify the NET$K_TAG_DNSOBJECTNAME and NET $K_TAG_DNSOBJECTNAME_INT item codes.

The value field of this network item list entry is in protocol tower format.

See *Section 2.13.7, "Protocol Tower Fields"* for protocol tower field parameters.

Valid for the IPC$K_FC_CONNECT_INITIATE, IPC$K_FC_DEREGISTER_OBJECT, IPC $K_FC_REGISTER_OBJECT, and IPC$K_FC_RESOLVE_NAME function codes.

NET$K_TAG_EVENTMASK

The NET$K_TAG_EVENTMASK item code enables event notification. Valid only for the IPC $K_FC_OPEN_ASSOCIATION function code. The value field of this item list entry is a longword that IPC interprets as a bit mask; each set bit enabling a specific type of event notification. The bits recognized by IPC are defined by:

- NET$V_EVENT_INCOMING
- NET$V_EVENT_EXPEDITED
- NET$V_EVENT_DISCONNECTS

Chapter 4. Queue I/O Request ($QIO) System Service

DECnet-Plus for OpenVMS allows you to perform a variety of operations over the network:

- Retrieve information about the status of the nodes in your network.
- Establish communication with a remote DECnet node through the heterogeneous command terminal facility.
- Access files on remote nodes.
- Perform task-to-task operations.

This chapter describes each of these operations. The primary focus of this chapter, however, is on the use of task-to-task communication in network operations.

Note

$IPC performs task-to-task communication on a system service level and provides functionality that is beyond the capability of $QIO. However, VSI no longer recommends that you use the $IPC system service for Phase V applications. VSI recommends that you use $QIO or the X/Open Transport Interface (XTI).

4.1. 64-Bit Virtual Address Support (Alpha only)

DECnet-Plus for OpenVMS provides support for 64-bit virtual addresses through the use of $QIO system service calls. Applications can use the DECnet-Plus $QIO interface to send messages from and receive messages into buffers residing in extended address space. Existing 32-bit applications will continue to run with the 64-bit-capable DECnet-Plus software. It is not necessary to recompile or relink existing applications that use 32-bit addresses with the DECnet-Plus $QIO interface.

The following table specifies the DECnet-Plus $QIO parameters that accept 64-bit virtual addresses. For further information about using the DECnet-Plus $QIO system service, refer to the *VSI DECnet-Plus for OpenVMS Programming* manual. Also, refer to general 64-bit information in the OpenVMS operating system documentation.

Operation	**$QIO Function Code and Modifier**	**64-Bit Address Parameter(s)**
Connect Initiate	IO$_ACCESS	P2 – 32-bit or 64-bit address of a 32-bit or 64-bit descriptor for the network connect block (NCB).
Connect Accept	IO$_ACCESS	P2 – 32-bit or 64-bit address of a 32-bit or 64-bit descriptor for the network connect block (NCB).
Connect Reject	IO$_ACCESS!IO$M_ABORT	P2 – 32-bit or 64-bit address of a 32-bit or 64-bit descriptor for the network connect block (NCB).

Operation	$QIO Function Code and Modifier	64-Bit Address Parameter(s)
Read	IO$_READVBLK	P1 – 32-bit or 64-bit buffer address.
Write	IO$_WRITEVBLK	P1 – 32-bit or 64-bit buffer address.
Disconnect (Synchronous)	IO$_DEACCESS!IO $M_SYNCH	P2 – 32-bit or 64-bit address of a 32-bit or 64-bit descriptor for optional user data.
Disconnect (Abort)	IO$_DEACCESS!IO $M_ABORT	P2 – 32-bit or 64-bit address of a 32-bit or 64-bit descriptor for optional user data.
Declare Named Object/ Application	IO$_ACPCONTROL	P1 – 32-bit or 64-bit address of a 32-bit or 64-bit descriptor for a NFB$C_DECLNAME NFB block.
		P2 – 32-bit or 64-bit address of a 32-bit or 64-bitstring descriptor for the object/application name.
Declare Numbered Object/ Application	IO$_ACPCONTROL	P1 – 32-bitor 64-bit address of a 32-bit or 64-bit descriptor for an NFB$C_DECLOBJNFB block.

For VAX P.S.I. user operations, refer to the VAX P.S.I. documentation set.

4.2. Establishing Communication with a Remote Node

DECnet-Plus for OpenVMS supports a command terminal facility that permits users to establish communication with a remote node and to use the facilities of that system while physically connected to the local node. By means of this link, you can temporarily become a local user of the remote node and thereby perform functions that the remote node allows its local users to perform from a terminal.

Note that, in addition to communicating with remote OpenVMS nodes, you can communicate with non-OpenVMS nodes that support the Network Architecture (NA) heterogeneous remote command terminal protocol facility (also referred to as the network virtual terminal facility). Consult the DECnet Software Product Description for a description of non–OpenVMS operating systems and their DECnet implementations.

If you want to use the command terminal facility to establish communication with a remote node, enter the DCL command SET HOST in the following format:

```
$ SET HOST nodename
```

where:

nodename	Is a 1- to 6-character name or number specifying the remote node at which you want to log in.

The SET HOST command does not recognize the area prefix in a node number. Therefore, to specify by number a node in another area, you must convert the node number to its decimal equivalent. The algorithm to convert the address to its decimal equivalent is: `area-number * 1024` + node-number

The operating system on the remote node prompts for a user name and password. If the information you supply is valid, you are logged in to the remote node. To return control to your local node, type LOGOUT.

If the remote node is an OpenVMS node, you receive the following message at your terminal after you type LOGOUT:

```
%REM-S-END, control returned to node _NODENAME::
```

This message indicates that control is returned to your local node.

The only special control character used for remote command terminal operations is Ctrl/Y. Except for Ctrl/Y, all control characters are handled as if they were issued at the local node.

Repeated, rapid pressing of Ctrl/Y generates a prompt asking if the remote connection should be broken. If you answer YES to the prompt, control returns to the local node. This technique is useful if for some reason you cannot return to the local node normally.

The following command sequence illustrates the operation of remote command terminals for the network topology example. The name of the local node is BOSTON.

```
$ SET HOST TRNTO
Username: SMITH
Password:

     Welcome to OpenVMS Version 7.1 on node TRNTO
                    .
                    .
                    .

$ LOGOUT
SMITH logged out at 30-NOV-1996 12:31:55:49

%REM-S-END, control returned to node _BOSTON::

$
```

When you are logged in at a remote node, you can use the SET HOST command to establish communication with another node. After logging in to node TRNTO, you could use SET HOST again to log in to another node (for example, node DENVER).

You would again be prompted for a user name and password. If you then supply a valid user name and password for node DENVER, you are logged in.

Note that when you log out of node DENVER, control is returned to node TRNTO. You must log out of node TRNTO to return to your local node, BOSTON.

4.3. Accessing Files on Remote Nodes

DECnet-Plus for OpenVMS allows you to access files on remote nodes in your network as though these files were on your local node. You can use the DECnet-Plus for OpenVMS facilities to access remote

files by means of DCL commands and command procedures, and MACRO and higher-level language programs using OpenVMS RMS or OpenVMS system services directly.

4.3.1. Using DCL Commands and Command Procedures

You can use most DCL commands that perform file operations at a local node to perform these operations on remote nodes. For example, you can use the same DCL commands to obtain directory listings, manipulate files, and execute command procedures on remote nodes. Generally, you need only prefix a nodename followed by two colons to the standard OpenVMS file specification to access the remote file. For example:

```
$ TYPE TRNTO::WORK$:[DOE]LOGIN.COM
```

In this example, the TYPE command requests that the file LOGIN.COM in the directory WORK$:[DOE] at the remote node TRNTO be displayed on your local terminal.

Depending on the file protections that are established on the remote node, you may need to supply an access control string in the DCL command when performing the file operation. For example:

```
$ COPY TRNTO"DOE JOHN"::WORK$:[DOE]LOGIN.COM *.*
```

In this example, an access control string is supplied as part of the request for the COPY operation. For OpenVMS operating systems, the access control string consists of a user name, followed by one or more spaces or tabs, and, optionally, one password and/or one account.

As with DCL, remote file accessing by higher-level languages is accomplished in a way that is transparent to the user. The only additional information you need to specify is the name of the remote node containing the file or files that you want to access. Like DCL, higher-level language programs also employ the OpenVMS RMS services to perform file access operations.

Command descriptions include restrictions that apply to individual commands and command qualifiers used in network operations. Unless otherwise stated, you can assume that a particular DCL command is supported for network operations.

4.3.2. Using Higher-Level Language Programs

You can use various higher-level languages to write programs that access remote files using the standard I/O statements of these languages. Regardless of the programming language used, you access remote files exactly as you would access local files.

In the following example, assume you want to design a FORTRAN program to transfer files from a local node to a remote node. You can identify the source and destination files by defining the logical names SRC and DST, respectively. You can use these DCL commands by entering the following commands:

```
$ DEFINE SRC TRNTO::INVENTDISK$:[STOCKROOM.PAPER]INVENTORY.DAT
$ DEFINE DST BOSTON::ARCDISK$:[ARCHIVE]TRNTO_INVENTORY.DAT
```

After you make the logical name assignments, the FORTRAN program can open the files by way of those logical names. You can use the following FORTRAN open calls:

```
OPEN (UNIT=1,NAME='SRC',TYPE='OLD',ACCESS='SEQUENTIAL',
     FORM='FORMATTED')
```

```
OPEN (UNIT=2,NAME='DST',TYPE='NEW',ACCESS='SEQUENTIAL',
     FORM='FORMATTED')
```

This FORTRAN program fragment uses standard I/O statements to transfer records from one file to another. In this example, the access mode is sequential.

As shown in the next example, you can design a FORTRAN program to transfer a file from the local node to a line printer on the remote node. You can define logical names for the source and destination, as follows:

```
$ DEFINE SRC TRNTO::INVENTDISK$:[STOCKROOM.PAPER]INVENTORY.DAT
$ DEFINE DSTLPR BOSTON::LPA0:
```

After you make the logical name assignments, the FORTRAN program can open the file and access the line printer by way of those logical names, as follows:

```
OPEN (UNIT=1,NAME='SRC',TYPE='OLD',ACCESS='SEQUENTIAL',
     FORM='FORMATTED')

OPEN (UNIT=2,NAME='DSTLPR',TYPE='NEW',ACCESS='SEQUENTIAL',
     FORM='FORMATTED',CARRIAGECONTROL='LIST',
      RECORDTYPE='VARIABLE')
```

This FORTRAN program fragment uses the standard I/O statements to transfer records from the source file to the destination line printer. The access mode of the file is sequential.

Examples of complete higher-level language programs designed to access remote files are included in the appropriate sections of the programming manuals for each VAX language.

4.3.3. Using RMS Services from MACRO Programs

The OpenVMS operating system provides a programming interface for remote file access using higher-level languages, including VAX MACRO. The MACRO programs can use OpenVMS Record Management Services (RMS) calls or OpenVMS system service calls. This section describes how you can use RMS to access remote files. The OpenVMS system services, which you can also use for remote file access, are described more completely in *Section 4.5.4, "Using System Service Calls in MACRO Programs"*.

For remote file processing, RMS integrates the network software necessary to translate standard RMS calls, which provides a transparent user interface to the network.

Using the RMS facilities, you can perform remote file-handling operations on entire files or access individual records, through programmed RMS service calls in a VAX MACRO application. All you need to do is supply the name of the remote node in your file specification.

As in the previous FORTRAN examples, you can use DCL commands to make logical name assignments to the source and destination files that you want to manipulate, for example:

```
$ DEFINE SRC TRNTO::INVENTDISK$:[STOCKROOM.PAPER]INVENTORY.DAT
$ DEFINE DST BOSTON::ARCDISK$:[ARCHIVE]TRNTO_INVENTORY.DAT
```

Before you can open either the source (SRC) or destination (DST) file with the RMS $OPEN statement, however, you must allocate the appropriate file access blocks (FABs) and record access blocks (RABs) in your program. To do this, you can use the following RMS structures:

```
                    .
                    .
                    .
SRC_FAB:
         $FAB    FAC=GET,-
                 FOP=SQO,-
                 FNM=SRC

SRC_FAB:
         $RAB    FAB=SRC_FAB,-
                 RAC=SEQ,-
                    .
                    .
                    .
```

These statements define the source file FAB and RAB control blocks. You must also define the destination file FAB and RAB control blocks, as follows:

```
                    .
                    .
                    .
DST_FAB:
        $FAB    FAC=PUT,-
                FOP=SQO,-
                FNM=DST,-
                ORG=SEQ,-
                RFM=VAR,-
                RAT=CR

DST_RAB:
        $RAB    FAB=DST_FAB,-
                RAC=SEQ,-
                    .
                    .
                    .
```

After defining the source and destination FABs and RABs, you can open the files for remote file processing. Note that, if your program accesses files sequentially, you can specify the sequential-only (SQO) option of the file options (FOP) field of the FAB. Specifying FOP=SQO enables RMS and the remote file access listener (FAL) to enter into file-transfer mode. In file-transfer mode there is no wait for message acknowledgment and, consequently, there is a significant increase in file-transfer performance.

Note that DECnet-Plus for OpenVMS does not support the use of RMS for operations on a remote magnetic tape volume.

4.4. Performing Task-to-Task Operations

Task-to-task communication is a feature common to all DECnet implementations. It allows two programs or tasks running under the same or different operating systems to communicate with each other regardless of the programming languages used. For example, a FORTRAN task running on the OpenVMS operating system at node BOSTON could exchange messages with a MACRO task running on the RSX–11M operating system at node DALLAS. Although these programs use different programming languages and run under different operating systems, the DECnet software translates system-dependent language calls into a common set of network protocol messages.

4.4.1. Transparent and Nontransparent Task-to-Task Communication

DECnet-Plus for OpenVMS supports both transparent and nontransparent task-to-task communication. Transparent communication provides the means for a DCL command procedure or a user program (written in either VAX MACRO or in a higher-level language) to communicate with other command procedures or user programs over the network, with no knowledge of the DECnet-Plus for OpenVMS software. Nontransparent communication allows the programmer to use system service options to perform network-specific functions.

There are important differences between these two forms of communication. Transparent communication is a form of device-independent I/O in OpenVMS in which you move data with little concern for the way the operation is accomplished. Likewise, transparent communication allows you to move data across the network without necessarily knowing that you are using DECnet software. Nontransparent communication, on the other hand, is a form of device-dependent I/O, in that you are interested in specific characteristics of the device that you want to access. A nontransparent task, in turn, can use network-specific features to monitor the communication process.

Note

While it is possible for a single task to create and maintain both transparent and nontransparent connections, each connection should be processed separately. That is, transparent-specific RMS and system services apply to transparent links, and nontransparent-specific system services apply to nontransparent links.

4.4.1.1. Transparent Communication

Transparent communication provides the basic functions necessary for a task to communicate with another task over the network. These functions include the initiation and completion of a logical link connection, the orderly exchange of messages between both tasks, and the controlled termination of the communication process. To perform these functions, you can write your cooperating tasks in any of the higher-level languages supported over the network, in VAX MACRO (using RMS service calls or system service calls), or by using DCL commands.

One way to view transparent communication is to look at the programming required to develop such an application. Transparent access provides the functions necessary to communicate over the network using standard I/O operations. When accessing the network transparently, you may use standard I/O statements of the higher-level language or straightforward RMS or system service calls to access a sequential record-oriented device. System service calls are described in *Section 4.5, "Performing Transparent Task-to-Task Operations"*.

4.4.1.2. Nontransparent Communication

Nontransparent communication provides the same functions as transparent communication plus additional system service and I/O features supported by DECnet-Plus for OpenVMS. In particular, a nontransparent task can create and use an OpenVMS mailbox to receive information that is not available to a transparent task with transparent communication. You can make use of network-specific features such as optional user data on connects and disconnects, and *interrupt messages*. Also, nontransparent tasks can receive and process multiple inbound connection requests. (See the description in *Section 4.6.1.5, "Completing the Establishment of a Logical Link"*.)

Note that on a OpenVMS Cluster node, nontransparent tasks that can receive multiple inbound connection requests should not use the cluster alias node address for outgoing connections. They should

also not be enabled to receive incoming connections directed to the cluster alias node. Incoming links directed to a cluster alias node address can be assigned to any of the nodes in the cluster that accept that alias node address, without knowledge of the nodes on which a declared task may be running.

In general, nontransparent tasks can use a mailbox to receive information about particular network operations. There are four types of mailbox messages:

- Messages that result from the use of certain system service calls(including optional user data carried on logical link creation or termination)
- Interrupt messages
- Logical link status messages
- Network system messages

Nontransparent functions that indirectly cause mailbox messages to be placed in the receiver's mailbox include calls for initiating, completing and terminating logical links. *Figure 4.1, "Mailbox Messages"*illustrates how nontransparent tasks use mailboxes.

Table 4.3, "System Mailbox Messages" (later in this chapter)provides a list of mailbox messages and their meanings.

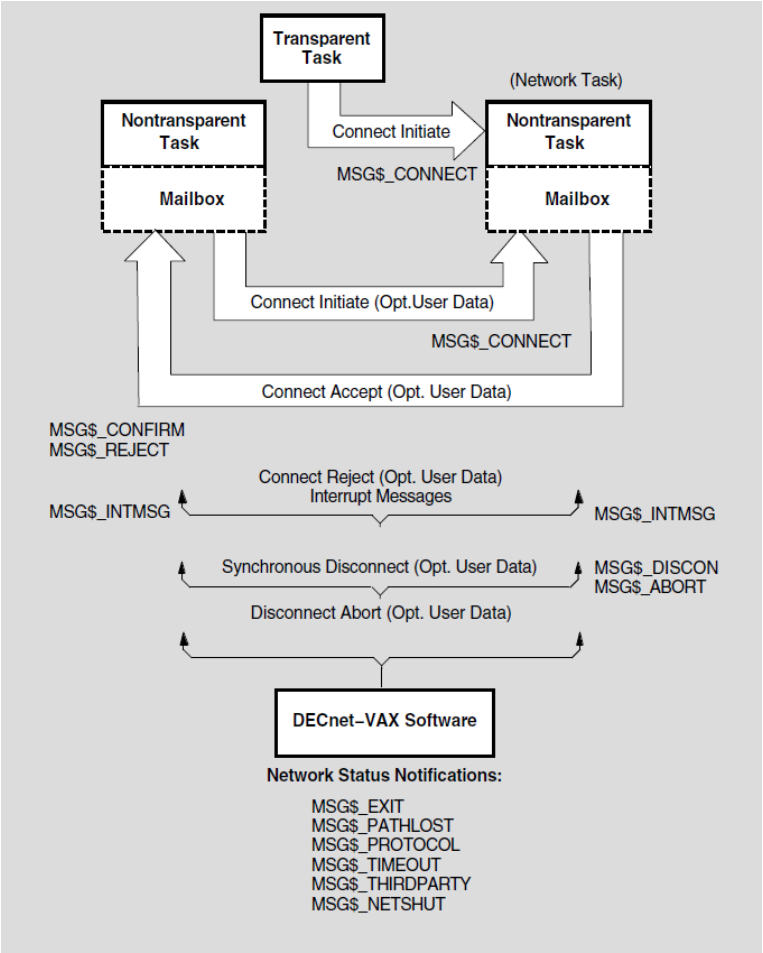
A nontransparent task can receive **network status notifications** in the mailbox. These notifications apply to physical and logical link conditions over the network. For example, DECnet-Plus for OpenVMS software can notify a nontransparent task of the following conditions:

- Third-party disconnections
- Network software- and hardware-related problems
- Processes exiting before I/O completion
- Connection request timeouts

4.4.2. Task Specification Strings in Task-to-Task Applications

Whether you are performing a transparent or nontransparent task-to-task operation, you must use a **task specification string** to identify the remote task with which you want to communicate. A task specification string is a quoted string that identifies the target task to which you attempt a logical link connection.

Figure 4.1. Mailbox Messages



To establish a logical link connection with a target task addressed as object type 0, use either of the following forms of task specification string:

- “TASK= *taskname*”
- “0= *taskname*”

where:

taskname	Can be from 1 to 12 characters.

Note that “0” and “TASK” are equivalent. (If the remote node is not an OpenVMS system, the maximum length of the task name may be different.)

If the remote node is an OpenVMS operating system, the *task name* usually represents the file name of a command procedure to be executed at the remote node. The *task name* may also represent a specific image to be run. The command procedure invoked at the remote node can complete the logical link itself (using a DCL OPEN command), or it can include a DCL RUN command to execute a program that completes the logical link.

The examples that follow illustrate two uses of the task specification string. The first example identifies the task TEST2 by using the "TASK=" form for specifying target tasks. The second example is the same as the first, except that access control information is provided and the alternative "0=" form for specifying a task is used.

```
BOSTON::"TASK=TEST2"BOSTON"SMITH JOHN"::"0=TEST2"
```

In this example, TEST2 refers to SYS$LOGIN:TEST2.COM for the default DECnet account (DECNET) at the remote OpenVMS node. Note that only the file name component of the command file specification is used in the task name string in this example. When naming the target task, you can specify a more complete file specification. For example, you can include a device name or a file type.

4.4.3. Functions Required for Performing Task-to-Task Operations

Several functions are necessary for performing a task-to-task operation. The number of functions, of course, depends on whether you intend to access the network transparently or non-transparently.

Even a transparent task-to-task application requires a minimum number of operations to initiate and complete a logical link connection, to exchange messages, and to terminate the logical link. These operations are actually a subset of a larger group of functions defined for nontransparent communication. The entire set of functions is as follows:

- **Initiating a logical link connection**
  - Requesting a logical link to a remote task
  - Declaring a network name and processing multiple connection requests
- **Completing a logical link connection**
  - Rejecting a logical link connection request
  - Accepting a logical link connection request[1]
- **Exchanging messages**
  - Sending and receiving data messages[1]
  - Sending and receiving interrupt messages
- **Terminating a logical link**
  - Synchronously disconnecting the logical link
  - Aborting the logical link[1]

Nontransparent tasks can use any or all of these functions to extend the basic capabilities offered under transparent communication.

4.4.3.1. Initiating a Logical Link Connection

Whether you access the network transparently or nontransparently, you must establish a communication link to the remote node on which the target task runs before any message exchange can take place. You establish the link by issuing a source task call that requests a logical link connection. The **source task** is the task that initiates a logical link connection request; the **target task** is the task with which you want to communicate.

[1]This operation represents the minimum subset for transparent task-to-task communication.

The interaction between the source task and the target task that takes place before the logical link is established is called a **handshaking sequence**. Upon receiving a call that requests a logical link connection, the local DECnet-Plus for OpenVMS node initiates a handshaking sequence with the target task. The following information is supplied in a connection request:

- An I/O channel. The I/O channel (more commonly referred to as the channel) serves as the path over which messages are sent and received by the source task.
- The identification of the target node. Every node in a network has an identifier that distinguishes it from all other nodes in the network. Transparent communication uses a task specification string to indicate the name of the target node. Nontransparent communication requires a user-generated data structure called the **network connect block(NCB)**, which also includes a task specification string.
- An object type descriptor.
- Access control information (optional).
- Optional user data. Nontransparent tasks have the option of sending up to 16 bytes of data to the target task (see the following information about NCBs).

You should be aware that after you issue a call that uses either a task specification string or an NCB, you access the network and, by definition, the DECnet-Plus for OpenVMS software.

4.4.3.2. Completing the Logical Link Connection

As part of the handshaking sequence, the target task completes the logical link connection in two steps. First, the DECnet software at the remote node processes the inbound logical link connection request. Second, the target task either accepts or rejects the link. These steps are performed differently, depending on whether the target task uses transparent or nontransparent I/O.

When a logical link request is received, a procedure called NET$SERVER.COM is executed, which in turn invokes the image NET$SERVER. This program works in conjunction with the network ACP (NET $ACP) and uses DCL to invoke the image or command procedure defined for the requested object. For example, the specified task is invoked for object 0 and FAL is invoked for object 17.

When the logical link is terminated, the “object” program (for example, FAL) also terminates. However, the process is not deleted. Instead, control is returned to NET$SERVER, which communicates with NET $ACP to inquire for another incoming logical link request. This inquiry process continues until NET $SERVER encounters a timeout condition (the default is 5 minutes).

The system manager can specify the time that NET$SERVER waits for another logical link request. The logical name NETSERVER$TIMEOUT, when defined, determines the amount of time NET$SERVER waits before reaching the timeout condition. Note that the equivalence name must be in the standard OpenVMS delta time format, for example, 0:10:0, representing 10 minutes.

You may define a number of NET$SERVER processes that never time out. This is useful on systems that are the target of significant amounts of network activity, such as mail or public file access. Two benefits may be gained: improved response time for the user initiating the network access, because there is no waiting for a new process to be created, and reduced overhead on the target system by virtue of fewer process creations.

To allow for permanent servers, define the logical name NETSERVER$SERVERS_*username* in the login procedure for the account receiving the network connects. The translation of the logical name should be the number of permanent servers you want. For example, to define two permanent servers for a default account named CML$SERVER, enter the following command:

```
$ DEFINE NETSERVER$SERVERS_CML$SERVER 2
```

You should put this command in the login command procedure of the default account; in this case, the SYS$LOGIN directory for CML$SERVER. You could also define it as a system logical name in the site-dependent system startup command procedure. The account must have WRITE access to its SYS $LOGIN directory. Note that you gain very little by defining only one permanent server, because a number of functions such as wildcard file copy require multiple logical links, each of which requires its own server.

If you use this mechanism, you should understand the interaction between proxy access and NET $SERVER processes. The proxy database is read by NET$ACP before a process has been created. For this reason, any incoming connection that may have a proxy account on the local system will not be given to an existing NET$SERVER process that was created for a different user.

In the following discussion, the remote node is assumed to be an OpenVMS operating system. If the remote node on which your target task runs is not an OpenVMS operating system, you should refer to the DECnet documentation for that system.

Completing the Connection Transparently

If the target task is transparent, the DECnet software at the remote node checks the access control information supplied in the connection request call.

Before you access the remote node, the system manager must have created the appropriate account in the user authorization file (UAF) (refer to the information on access control). In addition, the command procedure file (*taskname.*COM) starting the remote task must exist in the default directory associated with the account identified by the access control information. For a description of the command procedure *taskname*.COM, see *Section 4.7.1, "DCL Command Procedure for Task-to-Task Communication"*, which contains examples of command procedures designed for task-to-task communication.

Command procedures for objects existing in the Session Application database (which is created using NCL commands) are located in the SYS$SYSTEM directory. The VSI-supplied FAL.COM procedure is an example of such a command procedure. (Note that the command procedure is bypassed if the application definition specifies an EXE file.)

Completing the Connection Nontransparently

If the target task is nontransparent, then one of several things may occur. If the task has not declared itself a **network task** (and is therefore eligible to accept only one connection request at a time), then the DECnet software at the remote node performs the access checking procedure. After it starts, the target task retrieves the connection information by translating the logical name SYS$NET using the $TRNLNM system service call (see *Section 4.6, "Performing Nontransparent Task-to-Task Operations"*).

If the target task declares itself as an active network task, then DECnet-Plus for OpenVMS software places all connection requests addressed to the task in the mailbox associated with the channel being used. The first message in the mailbox is the NCB from the original connection request that started the task. This message appears in the mailbox after channel assignment and name declaration occur. After the task declares a network name or number, subsequent inbound connection requests are not checked by the remote node to verify access control. (Note that if the task is started without being part of a DECnet operation, access control is never checked.) *Section 4.6, "Performing Nontransparent Task-to-Task Operations"* describes in more detail the nontransparent process of completing the logical link connection.

After examining the incoming connection request, the target task either accepts or rejects the request, and optionally can send 1 to 16 bytes of data back to the source task at the same time that it responds

to the logical link connection request. Furthermore, a library routine, LIB$ASN_WTH_MBX, which assigns a channel and associates a unique mailbox, can be used when accepting the connection.

4.4.3.3. Exchanging Messages

When you access the network transparently or nontransparently, DECnet-Plus for OpenVMS sends data messages over a logical link in response to a set of send and receive calls issued by the source and target tasks. For higher-level language tasks, use standard read and write statements to send and receive data messages. (In *Example 4.2, "FORTRAN Task-to-Task Communication"*, the two FORTRAN tasks use read and write statements to exchange information. The equivalent RMS service calls are $GET and $PUT.)

After DECnet-Plus for OpenVMS creates a logical link, the two tasks are ready to exchange messages. This exchange can take place only if the two tasks cooperate in the transmission process. In other words, for each message sent by a task, the receiving task must issue a corresponding call to receive the message. Also, you must decide which task will disconnect the link. In addition, if the tasks are nontransparent, they must agree on whether or not the optional data will be passed. In the context of an established logical link, the task sending a message is the transmitter and the task receiving it is the receiver. Because logical links are inherently full duplex, each task may be a transmitter and a receiver simultaneously.

DECnet-Plus for OpenVMS distinguishes between two types of message: data messages and mailbox messages. Data messages are the normal mode of information exchange for both transparent and nontransparent communication. Mailbox messages such as interrupt messages, messages resulting from some DECnet operation (including optional user data), and network status notifications can be used only in nontransparent communication.

Nontransparent communication frequently involves using a mailbox to obtain network-specific information. A task may receive three types of message in its mailbox:

- Messages that DECnet generates when the task initiates certain network operations. An OpenVMS task issues system service calls to initiate these operations. For example:
  - When one task requests a logical link connection, a notification message(and optional user data) may be placed in the mailbox of the target task.
  - When a target task accepts or rejects the logical link connection request, a notification message (and optional user data) is placed in the mailbox of the source task.
  - When one task synchronously disconnects or aborts a logical link, a notification message (and optional user data) is placed in the mailbox of the task from which it is disconnecting.
- Network status notification messages that inform a task of some unusual network occurrence (such as a third-party disconnect).
- Interrupt messages sent by the other task.

4.4.3.4. Terminating a Logical Link Connection

The termination of a logical link signals the end of the communication between tasks.

In transparent communication using higher-level language statements, RMS service calls, or system service calls, either task can break the link. To terminate the link properly, the receiver, and not the transmitter, of the final message should issue the $CLOSE service to break the link. The link termination process is complete when the other task issues a link termination request. In transparent communication using system service calls, the $DASSGN system service call causes the link to be terminated.

Issuing the $CANCEL service call followed by the $DASSGN service call causes all pending operations to abort, then closes the link and deassigns the channel.

In nontransparent communication using system service calls, you can terminate I/O operations over a channel in one of three ways:

- **Synchronous Disconnect ($QIO)—**Specifies that all messages sent by the local task are required to be received and acknowledged by the remote End Communication Layer (ECL) before the logical link is disconnected. You should use this type of disconnect when the user of the logical link's services wants to ensure that the transmission of messages has completed before taking down the logical link. Note, however, that this service cannot guarantee the delivery of the received data to the remote task.

- **Disconnect Abort ($QIO)—**Specifies that all messages sent by the local task are not required to be received or acknowledged by the remote ECL before the logical link is disconnected. You should use this type of disconnect when the local task wants to reset the logical link to a known state. To ensure that the transmitted messages have been received and acknowledged by the remote ECL, the local task may issue the system service$CANCEL on the channel before issuing the disconnect abort. Note, however, that these services cannot guarantee the delivery of the received data to the remote task.

- **Deassign Channel and Terminate Link ($DASSGN)—**Specifies that all messages sent by the local task are not required to be received or acknowledged by the remote ECL before the logical link is disconnected. You should use this type of disconnect when the local task wants to break a logical link and deassign the channel to the network immediately.

Note that after either a synchronous disconnect or a disconnect abort of a nontransparent link, you can issue a new connection request because you did not deassign the I/O channel but merely deaccessed the link. For further information about these system service calls, see *Section 4.6, "Performing Nontransparent Task-to-Task Operations"*.

When a connection to a nontransparent task terminates the connection, a notification message indicating that the link is disconnected is placed in the mailbox of the affected task. A nontransparent task can send up to 16 bytes of optional user data, with the disconnect request. This optional user data is placed in the mailbox of the nontransparent task on the receiving end of the disconnect message.

Disconnect operations cannot guarantee to both partners that communication is complete. Therefore, VSI recommends that the communicating tasks agree on a protocol for terminating communication. In general, the receiver, not the transmitter, of the final message should disconnect the logical link.

Transparent communication allows you to create a logical link between tasks, send and receive data messages, and terminate the logical link at the end of the message dialog. The discussion covers general concepts implicit in DECnet-Plus for OpenVMS task-to-task communication and assumes familiarity with the QIO-related material in the *VSI OpenVMS System Services Reference Manual*. The use of higher-level language statements and RMS service calls in transparent task-to-task communication is described in *Section 4.5, "Performing Transparent Task-to-Task Operations"*.

4.5. Performing Transparent Task-to-Task Operations

This section describes the system service calls and functions you can use to perform transparent task-to-task communication over the network. You can perform these operations using any of the following methods:

- DCL commands and command procedures
- Higher-level language programs using appropriate language I/O statements
- MACRO or higher-level language programs using OpenVMS RMS calls or OpenVMS system service calls

See *Section 4.7, "Designing Tasks"* for examples of transparent task-to-task operations.

4.5.1. Using DCL Commands and Command Procedures

To perform transparent task-to-task operations, you can use DCL commands to construct and execute command procedures.

For example, to display information about another system, you can design a command procedure that can be invoked as a remote task. Assume that a procedure called SHOWBQ.COM is designed to return status information about jobs entered in batch queues on the system where it executes. Assume also that SHOWBQ.COM resides on node TRNTO. You can use SHOWBQ.COM for task-to-task communication by entering a task specification string in a TYPE command. For example:

```
$ TYPE TRNTO"BROWN JUNE"::"TASK=SHOWBQ"
```

See *Section 4.7.1, "DCL Command Procedure for Task-to-Task Communication"* for an example of a command procedure used for task-to-task communication.

4.5.2. Using Higher-Level Language Programs

This section contains examples of higher-level language calls that you can use for transparent task-to-task communication. Each higher-level language call contains a task specification string as part of its statement.

Higher-level language tasks can use standard file opening statements to request a logical link connection to a remote task. The following examples show how to specify a target task, TEST4, running on node TRNTO, in various languages supported on the OpenVMS operating system.

FORTRAN	OPEN(UNIT=7,NAME= 'TRNTO:: "TASK=TEST4 " ',TYPE= 'NEW ')
BASIC	OPEN 'TRNTO:: "TASK=TEST4 " 'AS FILE #7
PL/1	OPEN FILE(OUTPUT) TITLE('TRNTO:: "TASK=TEST4 " ');
Pascal	OPEN(PARTNER, 'TRNTO:: "TASK=TEST4 " ',NEW);
COBOL	SELECT PARTNER ASSIGN TO "TRNTO:: " "TASK=TEST4 " " ".OPEN OUTPUT PARTNER.
C	F1 = OPEN("TRNTO:: \ "TASK=TEST4 \ " ",2);

To complete the logical link, the target task performs a file opening operation using the logical name SYS$NET to establish a communications path back to the source task. The following examples show how to specify SYS$NET from higher-level language calls.

FORTRAN	OPEN(UNIT=2,NAME= 'SYS$NET ',TYPE= 'OLD ')
BASIC	OPEN "SYS$NET " ASFILE #2
PL/1	OPEN FILE(INPUT) TITLE('SYS$NET ');
Pascal	OPEN(PARTNER, 'SYS$NET ',OLD);

COBOL	SELECT PARTNER ASSIGN TO "SYS$NET ". OPEN INPUT PARTNER.
C	F2 = OPEN("SYS$NET ",2);

Section 4.7.2, "FORTRAN Program for Task-to-Task Communication" provides an example of a FORTRAN program designed for transparent task-to-task communication.

4.5.3. Using RMS Service Calls in MACRO Programs

You can write a MACRO program or a higher-level language program to perform transparent task-to-task communications, using RMS service calls. This section describes how to use RMS service calls in a MACRO program.

Note that the RMS $OPEN statement is equivalent to the higher-level language statements described in *Section 4.5.2, "Using Higher-Level Language Programs"*.

After you define the appropriate FAB and RAB control blocks, you can use the $OPEN statement to specify the target task, TEST4, running on node TRNTO. You can initiate the link by specifying the following call, in your MACRO program:

```
    TARGET:
           $FAB    FAC=<GET,PUT>,-
                   ORG=SEQ,-
                   FNM=<NODE::"TASK=TEST4">
           $OPEN   FAB=TARGET
```

To complete the logical link, the target task performs a file-opening operation using the logical name SYS$NET to establish a communications path back to the source task. For example:

```
REQUESTER:
           $FAB    FAC=<GET,PUT>,-
                   ORG=SEQ,-
                   FNM=<SYS$NET>
           $OPEN   FAB=REQUESTER
```

As in the case of the target task, the appropriate FABs and RABs must already be declared, if the RMS OPEN call is to succeed. On inbound connections, DECnet-Plus for OpenVMS automatically makes the logical name assignment to SYS$NET.

4.5.4. Using System Service Calls in MACRO Programs

You can write MACRO programs or higher-level language programs to perform transparent task-to-task communications, using system service calls. This section focuses on MACRO programs using system service calls for performing these operations.

Table 4.1, "System Service Calls for Transparent Communication" summarizes these calls and their network-related functions. *Section 4.5.5, "Summary of System Service Calls for Transparent Operations"* presents the format of these calls in more detail.

Table 4.1. System Service Calls for Transparent Communication

Call	Function
$ASSIGN	Request a logical link connection
$DASSGN	Terminate a logical link
$QIO (IO$_READVBLK)	Receive a message

Call	Function
$QIO (IO$_READVBLK!IO$M_MULTIPLE)	Receive a message in multiple receive requests
$QIO (IO$_WRITEVBLK)	Send a message
$QIO (IO$_WRITEVBLK!IO$M_MULTIPLE)	Send a message in multiple write requests

These calls allow you to perform task-to-task communication in much the same way as you would perform normal I/O operations. Use the $ASSIGN call to assign a logical link I/O channel to a device, which in this case is a task that behaves like a full-duplex record-oriented device. You can perform read and write operations with this task either synchronously or asynchronously. To exchange messages, use the Queue I/O (QIO) requests supported by DECnet-Plus for OpenVMS. When all communication completes, use the $DASSGN system service call to deassign the channel and thereby disconnect the logical link.

4.5.4.1. Requesting a Logical Link

To request a logical link and assign an I/O channel, use the $ASSIGN system service. When you issue this call, you must include a task specifier for the remote node on which the cooperating task runs. The task specifier identifies the remote node and the target task to which you want to establish a logical link.

For example, you could establish a logical link to target task TEST2 on node TRNTO to perform task-to-task communication. To create this link, code the following VAX MACRO statements in your source program.

```
TARGET:     .ASCID  /TRNTO::"TASK=TEST2"/
NETCHAN:    .BLKW   1       ; Channel number returned here
                .
                .
                .
            $ASSIGN_S      DEVNAM=TARGET,CHAN=NETCHAN
```

For debugging or for symmetry, you can develop and run the target task on the local node. Use the local node name (or node number 0) plus two colons to connect to the local node. This practice applies to DCL, higher-level languages and RMS, as well as system services.

After you establish a logical link, you refer to the assigned channel in any succeeding call in the MACRO program, either to send or receive messages, or to deassign the channel and terminate the logical link.

Until the connection operation completes, the process is in local event flag (LEF) wait state in kernel mode. Therefore, pressing Ctrl/Y does not return the process to DCL status. The maximum amount of time that the process will wait in this state is specified by the OUTGOING TIMER parameter of the NCL command SET SESSION CONTROL. If this timer cannot be set to an acceptable value, tasks that accept commands from the terminal should use $QIO (IO$_ACCESS) instead of the transparent $ASSIGN call to initiate logical links.

4.5.4.2. Completing the Logical Link Connection

The target task completes the logical link by specifying the logical name SYS$NET as the *devnam* argument for the $ASSIGN system service. For example:

```
LOGNAM:     .ASCID  /SYS$NET/
NETCHAN:    .BLKW   1       ; Channel number returned here
                .
                .
                .
            $ASSIGN_S      DEVNAM=LOGNAM,CHAN=NETCHAN
```

Issue this call in the target task to complete the logical link connection. The target task also specifies a channel to be used in subsequent system service calls.

The remote node is assumed to be an OpenVMS operating system. If the remote node on which the target task runs is other than OpenVMS, you should refer to the related DECnet documentation.

4.5.4.3. Exchanging Messages

After DECnet-Plus for OpenVMS software establishes a logical link with the target task, either task can then send or receive messages. However, they must cooperate with each other: For each message sent with the $QIO (IO$_WRITEVBLK), the other task must issue a corresponding $QIO (IO $_READVBLK) to receive the message.

On logical links, DECnet-Plus for OpenVMS supports sending and receiving data messages that are larger than the maximum size allowed by the $QIO system service. You do this by allowing write and read requests to be fragmented across multiple $QIO requests. To fragment writes and reads, you must include the modifier IO$M_MULTIPLE on the write or read $QIO call.

When you supply the modifier on a write message request $QIO (IO$_WRITEVBLK!IO $M_MULTIPLE), it indicates that more data will be supplied for this message. To indicate the last fragment of the message being sent, you should issue the write request without a modifier$QIO (use the QIO called IO$_WRITEVBLK).

When you supply the modifier on a read message $QIO (IO$_READVBLK!IO$M_MULTIPLE), if the received data message contains more than enough data to fill the buffer supplied with the read request, then SS$_BUFFEROVF is returned. This is not an error status. The next read posted receives the next fragment of the data message. If the received message fits into the buffer posted, then SS$_NORMAL is returned. Tasks that require fragmentation should always supply the IO$M_MULTIPLE on read requests.

If you do not use the read multiple request to receive a data message, then you must ensure that the tasks allocate enough buffer space for receiving the messages. If the tasks do not, SS$_DATAOVERUN error occurs. You must also ensure that the end of the dialog can be determined.

One of the two tasks must disconnect the logical link. To terminate a logical link properly, the receiver, and not the transmitter, of the final message should break the link.

DECnet-Plus for OpenVMS does not provide an automatic timeout of read or write requests. If the task needs to stop a read or write request on a logical link, it must do so by disconnecting or aborting the logical link.

4.5.4.4. Terminating the Logical Link

Use the $DASSGN system service call to deassign the channel and break off the logical link with the cooperating task. This call terminates all pending calls for sending and receiving messages, aborts the link immediately, and frees the channel associated with that logical link.

4.5.4.5. Status and Error Reporting

When a system service completes execution, a status value is returned (does not apply to the $EXIT service). The $ASSIGN, $DASSGN, and $QIO system services place the return status information in register 0 (R0). For the$QIO system service, a successful return status indicates only that the request was queued successfully. All I/O completion status information is placed in the I/O status block (IOSB). For example, a $QIO system service read request to a task might be successful (status return is SS $_NORMAL) yet fail because the link was disconnected. (I/O status return is SS$_LINKABORT.) The return status codes shown in the following sections may be returned both in R0 and in the IOSB.

When DECnet-Plus for OpenVMS returns the status SS$_NORMAL in the I/O status block on a write request, it means that the write was queued for transmission on the logical link. It does not mean that the write request has been received or acknowledged by the remote task. The logical link services of DECnet-Plus for OpenVMS provide the guaranteed delivery of transmitted messages to the remote node. If a message cannot be delivered, the user is notified by the disconnection of the logical link. The DECnet-Plus for OpenVMS services cannot guarantee the delivery of data received on the remote node to the remote task. It is the responsibility of cooperating tasks to agree on a protocol to ensure that data transmitted by the local task is received by the remote task.

The *VSI OpenVMS System Services Reference Manual* and the *Guide to OpenVMS Programming Resources* both provide more information about $QIO system services.

4.5.5. Summary of System Service Calls for Transparent Operations

The following sections describe the OpenVMS system services you can use for transparent task-to-task communication. Each description covers the use of the call, its format, the arguments associated with the call, and the return status information.

4.5.5.1. $ASSIGN

The $ASSIGN system service assigns a channel to refer to the logical link. You can then use the channel returned in the *chan* argument in any succeeding call to send or receive a message, or to deassign the channel and thereby terminate the logical link.

Format

```
$ASSIGN devnam, chan, [acmode]
```

Arguments

devnam	Address of a quadword descriptor of a character string that identifies the remote task. The string contains either of the following: • A task specification string if the call is by the source task. Both the string and its descriptor must be in read/write storage. • The SYS$NET logical name if the call is by the target task.
chan	Address of a word that is to receive the assigned channel number. You use this channel number to send a message to a remote task, receive a message from a remote task, or to abort the logical link.
acmode	Access mode to be associated with this channel. The most privileged access mode used is the access mode of the caller. You can perform I/O operations on the channel only from equal or more privileged access modes.

Return Status

SS$_CONNECFAIL	The connection to a network object timed out or failed.
SS$_DEVOFFLINE	The physical link is shutting down.
SS$_FILALRACC	A logical link already exists on the channel.
SS$_INSFMEM	There is not enough system dynamic memory to complete the request.

SS$_INVLOGIN	The access control information was found to be invalid at the remote node.
SS$_IVDEVNAM	The task specifier has an invalid format or content.
SS$_LINKEXIT	The network partner task was started, but exited before confirming the logical link (that is, $ASSIGN to SYS$NET).
SS$_NOLINKS	No logical links are available. The maximum number of logical links as set for the NCP executor MAXIMUM LINKS parameter was exceeded.
SS$_NOPRIV	The issuing task does not have the required privilege to perform network operations or to confirm the specified logical link.
SS$_NOSUCHNODE	The specified node is unknown.
SS$_NOSUCHOBJ	The network object number is unknown at the remote node; or for a TASK= connect, the named DCL command procedure file cannot be found at the remote node.
SS$_NOSUCHUSER	The remote node could not recognize the login information supplied with the connection request.
SS$_PROTOCOL	A network protocol error occurred, most likely because of a network software error.
SS$_REJECT	The network object rejected the connection.
SS$_REMOTE	The service completed successfully. (A logical link was established with the target task.)
SS$_REMRSRC	The link could not be established because system resources at the remote node were insufficient.
SS$_SHUT	The local or remote node is no longer accepting connections.
SS$_THIRDPARTY	The logical link connection was terminated by a third party (for example, the system manager).
SS$_TOOMUCHDATA	The task specified too much optional or interrupt data.
SS$_UNREACHABLE	The remote node is currently unreachable.

4.5.5.2. $QIO (Sending a Message to a Target Task)

The $QIO system service with a function code of IO$_WRITEVBLK or IO$_WRITEVBLK!IO $M_MULTIPLE sends a message to a target task. The $QIO call initiates an output operation by queuing a request to the channel associated with the logical link. Alternatively, you could use the $QIOW system service to perform the same operation but also wait for I/O completion.

Format

$QIO [efn], chan, func, [iosb], [astadr], [astprm], p1, p2$QIOW

Arguments

efn	Number of the event flag to be set at request completion.

chan	Word containing the channel number associated with the logical link. Use the same channel number returned in the$ASSIGN call.
func	IO$_WRITEVBLK or IO$_WRITEVBLK!IO$M_MULTIPLE.
iosb	Address of a quadword I/O status block that is to receive the completion status.
astadr	Entry point address of an asynchronous system trap (AST) routine that executes when the I/O operation completes. If specified, the AST routine executes at the access mode from which the $QIO service was requested.
astprm	AST parameter to be passed to the AST completion routine.
p1	Buffer address.
p2	Buffer length in bytes.

Return Status

SS$_NORMAL	The service completed successfully.
SS$_ABORT	The I/O request has been aborted by a $DASSGN or $CANCEL call.
SS$_CANCEL	The I/O on this channel has been canceled.
SS$_FILNOTACC	No logical link is associated with the channel.
SS$_INSFMEM	Enough memory to buffer the message could not be allocated.
SS$_LINKABORT	The network partner task aborted the logical link.
SS$_LINKDISCON	The network partner task disconnected the logical link.
SS$_LINKEXIT	The network partner task exited.
SS$_PATHLOST	The path to the network partner task node was lost.
SS$_PROTOCOL	A network protocol error occurred. This is most likely due to a network software error.
SS$_THIRDPARTY	The logical link connection was terminated by a third party (for example, the system manager).

4.5.5.3. $QIO (Receiving a Message from a Target Task)

The $QIO system service with a function code of IO$_READVBLK receives a message from a target task. The $QIO call initiates an input operation by queuing a request to the channel associated with the logical link. Alternatively, you could use the $QIOW system service to perform the same operation but also wait for I/O completion.

Format

```
$QIO [efn], chan, func, [iosb], [astadr], [astprm], p1, p2$QIOW
```

Arguments

efn	Number of the event flag to be set at request completion.
chan	Word containing the channel number associated with the logical link. Use the same channel number returned in the $ASSIGN call.
func	IO$_READVBLK or IO$_READVBLK!IO$M_MULTIPLE.

iosb	Address of a quadword I/O status block that is to receive the completion status.
astadr	Entry point address of an AST routine that executes when the I/O operation completes. If specified, the AST routine executes at the access mode from which the $QIO service was requested.
astprm	AST parameter to be passed to the AST completion routine.
p1	Buffer address.
p2	Buffer length in bytes.

Return Status

SS$_NORMAL	The service completed successfully.
SS$_ABORT	The I/O request has been aborted by a $DASSGN or$CANCEL call.
SS$_CANCEL	The I/O on this channel has been canceled.
SS$_DATAOVERUN	More bytes were sent than could be received in the supplied buffer. This status will not be returned when IO$M_MULTIPLE is used on the read request.
SS$_FILNOTACC	No logical link is associated with the channel.
SS$_INSFMEM	Enough memory to buffer the message could not be allocated.
SS$_LINKABORT	The network partner task aborted the logical link.
SS$_LINKDISCON	The network partner task disconnected the logical link.
SS$_LINKEXIT	The network partner task exited.
SS$_PATHLOST	The path to the network partner task node was lost.
SS$_PROTOCOL	A network protocol error occurred. This is most likely due to a network software error.
SS$_THIRDPARTY	The logical link connection was terminated by a third-party (for example, the system manager).
SS$_BUFFEROVF	Data could not fit in the buffer supplied. Supply another read request to receive the next fragment of received data message.

4.5.5.4. $DASSGN (Disconnecting a Logical Link)

The $DASSGN system service terminates all pending operations to send and receive data, disconnects the logical link immediately, and frees the channel associated with that link. Either task can terminate the logical link by calling $DASSGN.

Format

`$DASSGN chan`

Argument

chan	Word containing the channel number to the logical link you want disconnected. Use the same channel number returned in the $ASSIGN call.

Return Status

SS$_NORMAL	The service completed successfully.
SS$_IVCHAN	The process specified an invalid channel.
SS$_NOPRIV	The specified channel was not assigned or was assigned from a more privileged access mode.

4.6. Performing Nontransparent Task-to-Task Operations

This section describes the system service calls and functions that you use for nontransparent task-to-task communication. In general, the principles of nontransparent task-to-task communication are similar to those of transparent communication.

If you want to perform nontransparent communication operations, you can write VAX MACRO programs using OpenVMS system services designed specifically for DECnet-Plus for OpenVMS. You can also write programs in one of the higher-level languages, provided the language supports the DECnet-Plus for OpenVMS services. These DECnet-Plus for OpenVMS services are described in detail throughout this section.

DECnet-Plus for OpenVMS also provides additional services with extensions that allow you to use network-specific features for nontransparent network operations, such as the following:

- Creating and using mailboxes for receiving messages, including network status notifications.
- Declaring a task as a network task, thus enabling it to process multiple inbound logical link connection requests.
- Sending connection requests, optionally with user data.
- Accepting or rejecting a connection request, optionally with user data.
- Communicating between a transparent and a nontransparent task.
- Sending or receiving an interrupt message.
- Aborting or synchronously disconnecting a logical link, optionally with user data.

The general concepts implicit in DECnet-Plus for OpenVMS task-to-task communication are covered in *Section 4.5, "Performing Transparent Task-to-Task Operations"*. You should also be familiar with the material in the *VSI OpenVMS System Services Reference Manual* and the *VSI OpenVMS I/O User's Reference Manual*.

4.6.1. Using System Services for Nontransparent Operations

Nontransparent task-to-task communication over the network uses a set of system service calls available under the OpenVMS operating system. *Table 4.2, "System Service Calls for Nontransparent Communication"* summarizes these calls and their network-related functions. The $QIO calls are distinguished by function code.

Table 4.2. System Service Calls for Nontransparent Communication

Call	Function
$ASSIGN	Assign an I/O channel
$CANCEL	Cancel I/O on a channel
$CREMBX	Create a mailbox
$DASSGN	Abort the logical link connection (deassigning an I/O channel)
$GETDVI	Get information on device or volume
$QIO (IO$_ACCESS)	Request a logical link connection
$QIO (IO$_ACCESS)	Accept a logical link connection request
$QIO (IO$_ACCESS!IO$M_ABORT)	Reject a logical link connection request
$QIO (IO$_ACPCONTROL)	Assign a network name to a task eligible to accept multiple inbound connection requests
$QIO (IO$_DEACCESS!IO$M_ABORT)	Abort the logical link connection
$QIO (IO$_DEACCESS!IO$M_SYNCH)	Synchronously disconnect a logical link
$QIO (IO$_READVBLK)	Receive a message
$QIO (IO$_READVBLK!IO$M_MULTIPLE)	Receive a message in multiple receive requests
$QIO (IO$_WRITEVBLK)	Send a message
$QIO (IO$_WRITEVBLK!IO$M_MULTIPLE)	Write a message in multiple write requests
$QIO (IO$_WRITEVBLK!IO$M_INTERRUPT)	Send an interrupt message
$TRNLNM	Translate logical names

4.6.1.1. Assigning a Channel to _NET: and Creating a Mailbox

To prepare for nontransparent task-to-task communication, you need to assign a channel just as you would for transparent communication. In addition, you can create a mailbox to take advantage of optional network protocol features.

You must assign a channel to the pseudodevice _NET:; use the $ASSIGN system service call for this purpose. This call normally contains a reference to a mailbox, thereby associating it with the channel assigned to _NET:. If you use a mailbox, you must create the mailbox before assigning a channel to _NET:. It is important to note that this use of the $ASSIGN system service differs from its use for transparent communication. Assigning a channel to _NET: does not transmit a logical link connection request to the remote node. Instead, the channel to _NET: provides a communication path to DECnet software. You must use a separate $QIO call (IO$_ACCESS function using the same channel) to request a logical link to the remote task. Refer to *Section 4.6.2.1, "$ASSIGN (I/O Channel Assignment)"* for details about the$ASSIGN system service.

To take advantage of optional network protocol features, you can create a mailbox to receive messages on behalf of logical link operations. For example, the mailbox receives a message indicating whether the cooperating task accepted or rejected a connection request issued by the source task. Use the $CREMBX system service to create a mailbox for these purposes. In the event that your application does not need the information supplied in the mailbox, you need not create a mailbox.

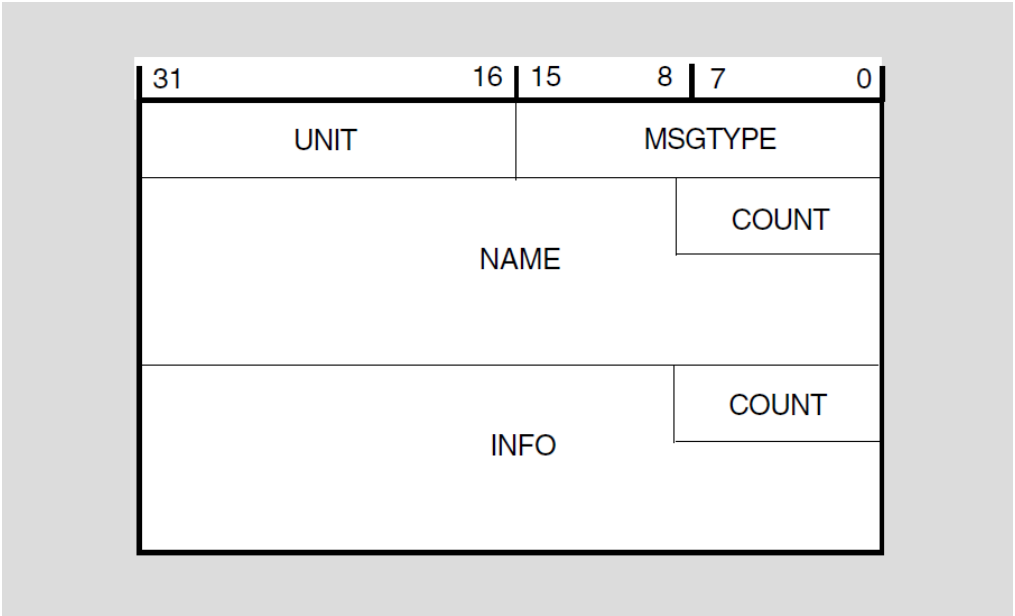
For convenience, you can use the Run-Time Library routine LIB$ASN_WTH_MBX to create a temporary mailbox, assign a channel to it, and assign a channel to _NET:. This routine creates a unique mailbox on each call to the routine. Multiple copies of a task using this routine, in effect, use different

mailboxes. If you were to create a mailbox with a logical name within the task, then all copies of that task would use the same mailbox and thereby interfere with each other's mailbox messages.

4.6.1.2. Mailbox Message Format

The mailbox receives information specific to nontransparent communication with a remote task. *Figure 4.2, "Mailbox Message Format"* illustrates the general format of the mailbox message.

Figure 4.2. Mailbox Message Format



Notes on *Figure 4.2, "Mailbox Message Format"*

MSGTYPE	Contains a code that identifies the message type.
UNIT	Contains the binary unit number of the device for which the message applies.
COUNT NAME	Contains a counted ASCII string that gives the name of the device for which the message applies. The $ASSIGN system service creates devices having names beginning with NET.
COUNT INFO	Contains a counted ASCII string of information, which depends on the message type.

All system mailbox messages contain, in the first word of the message, a constant that identifies the sender of the message. These constants have symbolic names (defined in the $MSGDEF macro) in the following format:`MSG$_sender`

Table 4.3, "System Mailbox Messages" summarizes the system mailbox messages that pertain to nontransparent task-to-task communication.

Table 4.3. System Mailbox Messages

Symbolic Name	Meaning
MSG$_TRMUNSOLIC	Unsolicited terminal data

Symbolic Name	Meaning
MSG$_CRUNSOLIC	Unsolicited card reader data
MSG$_ABORT	Network partner aborted link
MSG$_CONFIRM	Network connect confirm
MSG$_CONNECT	Network inbound connect initiate
MSG$_DISCON	Network partner disconnected; hang-up
MSG$_EXIT	Network partner exited prematurely
MSG$_INTMSG	Network interrupt message; unsolicited data
MSG$_PATHLOST	Network path lost to partner
MSG$_PROTOCOL	Network protocol error
MSG$_REJECT	Network connect reject
MSG$_THIRDPARTY	Network third party disconnect
MSG$_TIMEOUT	Network connect timeout
MSG$_NETSHUT	Network shutting down

4.6.1.3. Requesting a Logical Link Connection

After you assign the I/O channel, you can request a logical link connection to the target task. Use the $QIO system service with a function code of IO$_ACCESS. You must identify the target task in the $QIO call. Use a network connect block (NCB) to specify the target task identification string. In addition, you can optionally send one to 16 bytes of data in the NCB. The format of the NCB is discussed in *Section 4.6.1.4, "Using the Network Connect Block"*.

After the source task issues the connection request, it can issue a $QIO call with a function code of IO$_READVBLK to read its mailbox. Checking the contents of the mailbox is one way to determine whether the target task accepted or rejected the connection request. The mailbox can contain a variety of information, including either the MSG$_CONFIRM or MSG$_REJECT messages, and possibly optional data in the mailbox buffer.

If specified, the IOSB argument of the $QIO (IO$_ACCESS) call will also contain the result of the connection request operation. *Section 4.6.2.2, "$QIO (Requesting a Logical Link Connection)"* provides a complete list of I/O status messages for this call.

Note that you must read the mailbox to inspect any optional data sent from the target task upon accepting or rejecting the connection request.

4.6.1.4. Using the Network Connect Block

The network connect block (NCB) is a user-generated data structure that contains the information necessary to request a logical link connection or to accept or reject a logical link connection request. The NCB must be in read/write storage.

The NCB identifies a specific task using a task specification string. This task specification string specifies either an object name or an object number. The following are valid task specification strings:

```
"TASK=TEST2"TASK=157"0=TEST2"
```

For an inbound call with an NCB, the task name portion of the task specification string is a process ID if the remote node is an OpenVMS operating system; if not, then the task name portion is a system-

specific string that identifies an executable unit (for example, job or task). The task specification string must be enclosed in quotation marks. Note that the final quotation mark of the task specification string follows the last item within the NCB. *Section 4.4.2, "Task Specification Strings in Task-to-Task Applications"* provides additional information about task specification strings.

Example 4.1, "Network Connect Block Format" shows an NCB you could use when issuing a connection request call. The significance of the information contained in the NCB block varies, depending on the type of call in which it is used. If the call is an outbound connection request with no optional data, items 2, 3, 4, and 5 of the block are not required. If the call is a connect accept operation and no optional data is sent, then items 4 and 5 are not required. Item 5 is meaningful only to the receiver of a connection request.

Example 4.1. Network Connect Block Format

```
1.  With optional data (outbound connect):
                                        ❶       ❷
NCB:    .ASCII  ?TRNTO::"TASK=TEST2/?
             .WORD   0❸
OPTDATA:
             .ASCIC /USERINFO/
             .BLKB 17-<.-OPTDATA>❹
             .ASCII /"/
                 ❺
2.  Without optional data (outbound connect):
                                           ❶
NCB:     .ASCII   ?TRNTO::"TASK=TEST2"?
```

Item	Function
❶	A valid task specification string.
❷	The slash character (/).
❸	One word. This word must be 0 for a connection request operation. For a connect accept or reject operation, this word contains an internal DECnet link identifier.
❹	Up to 16 bytes of optional data sent as a counted string. This string is stored in a fixed-length field that is 17 bytes long. DECnet-Plus for OpenVMS software ignores unused bytes.
❺	A destination descriptor. This descriptor indicate show the connection was issued and is meaningful only to the task or object to which the connection is made. This information is useful where one program serves many functions and needs to know how it was invoked. The maximum length for the destination descriptor is 19 bytes. The format is as follows: a. If byte 0 contains 0, then byte 1 is the binary value of the object number. b. If byte 0 contains 1, then byte 1 is the binary object number, and bytes 2 through 18 contain a counted task name.

	c. If byte 0 contains 2, then byte 1 is the binary object number; bytes 2 through 5 contain a UIC, the first two bytes of which contain a binary group code and the second two bytes contain a binary user code; and bytes 6 through 18 contain a counted task name.

4.6.1.5. Completing the Establishment of a Logical Link

A nontransparent target task completes the logical link connection in one of several ways, depending upon whether the task can process multiple inbound connection requests or just a single request. Furthermore, a nontransparent target task has the option of accepting or explicitly rejecting a logical link request.

Receiving Connection Requests

This section describes what happens when you receive single and multiple connection requests. The remote node is assumed to be OpenVMS. If the remote node on which your target task runs is other than OpenVMS, you should refer to the related DECnet documentation.

When a remote node receives a call requesting a logical link, the DECnet-Plus for OpenVMS software constructs an NCB from the information contained in the call. At this point, one of two things occurs. If a task already running on the remote node has declared a network name or object number which is the same as the one identified in the constructed NCB, the software puts the NCB into that task's mailbox. If not, DECnet-Plus for OpenVMS must create a process to execute the task. The DECnet-Plus for OpenVMS software either uses a compatible netserver process (if one exists) or creates a netserver process (if one does not already exist) to execute NET$SERVER.COM, which in turn runs NET$SERVER.EXE.

If the task running on the remote node has not declared a network name or network object, SYS$NET is equated to the NCB, and LOGIN.COM (if it exists) is invoked, which in turn starts the *taskname*.COM command file. The name of this command file is determined as follows:

- If the connection request identifies a numbered (nonzero) object, then the appropriate record is located in the configuration database and the name of the file is found in this record. (The file is assumed to reside in SYS$SYSTEM.)
- If the connection request identifies a named object with type 0 (TASK), then the file name is assumed to be the name of the task connected to (with a file type of COM) and is assumed to reside in the default directory associated with the access control information.

When executing, the target task can determine whether to accept or explicitly reject the connection request. You can program the target task to base this assessment on the information contained in the NCB.

A nontransparent target task can accept only one connection request at a time, unless it declares itself as a network task. The target task may retrieve the connection information by translating the logical name SYS$NET using the$TRNLNM system service. After the task retrieves the logical name, it may decide whether to accept or explicitly reject the connection request.

Note that you need to translate SYS$NET only if you require the following information:

- The optional data in the network connect block
- The identity of the connector

- The descriptor by which the connection was made

A target task can accept multiple inbound connection requests only if it declares itself a known network task. To make this declaration, you must first use the $ASSIGN call to assign an I/O channel to _NET:. Then, use the $QIO system service with the function code IO$_ACPCONTROL to assign a network name or object number to the task, making it eligible to process multiple inbound connection requests. This system service requires SYSNAM privilege. You must associate a mailbox with the channel if a name or number is to be declared.

You should program tasks that have declared names or object numbers to terminate when their mailboxes receive a MSG$_NETSHUT message. You must restart such tasks when the network comes back up.

After you declare the target task as an active network task, DECnet places all connection requests addressed to the task in the mailbox associated with the channel over which the ACP control function was issued. The target task retrieves connection requests from the mailbox by issuing the $QIO system service call with the function code IO$_READVBLK. Note that the first message in the mailbox is the NCB from the original connection request that put the task into a state of execution. After the task declares a network name or object number, subsequent inbound connection requests are not checked for their access control information.

Note that you can start tasks that declare names or object numbers apart from the first inbound connection (that is, by a RUN command). However, if the network task is started separately from a DECnet operation, access control is never checked.

Accepting or Rejecting a Connection Request

The target task can either accept or reject a connection request. To accept a connection request, thus completing the logical link connection, use the $QIO system service with the function code IO $_ACCESS. To reject the connection request, use the $QIO system service with the function code IO $_ACCESS!IO$M_ABORT. When it either accepts or rejects the connection request, the target task can also send 1 to 16 bytes of optional data within a modified NCB back to the source task.

Exchanging Data Messages and Interrupt Messages

The exchange of data messages between the two cooperating tasks is performed in the same way for both nontransparent and transparent communication. (Refer to *Section 4.5.4.3, "Exchanging Messages"* for information about exchanging messages on DECnet-Plus for OpenVMS logical links.)

The exchange of interrupt messages applies only to nontransparent communication. Either task can send a 1- to 16-byte interrupt message. You can use this method to send a message to a target task outside the normal flow of data messages. DECnet-Plus for OpenVMS places the received interrupt message in the target task's mailbox. Use the $QIO system service with the function code IO$_WRITEVBLK!IO $M_INTERRUPT to send the interrupt message. If the target task needs to be notified that an interrupt message has been placed in its mailbox, then it should issue a $QIO system service read request to the mailbox. The task may also specify an AST on the $QIO request to cause the execution of a special routine to handle the received interrupt message. (AST routines are described in the *VSI OpenVMS System Services Reference Manual*.)

4.6.1.6. Disconnecting or Aborting the Logical Link

A nontransparent task can terminate communication with a remote task either by disconnecting the link (synchronous disconnect or disconnect abort) or by deassigning the channel. In the first instance, you can issue a new connection request on the same channel because you do not deassign it. If you specifically

use the IO$_DEACCESS, as opposed to the $DASSGN method of terminating a link, you can send an optional message of 1 to 16 bytes of data with the $QIO call.

To disconnect a logical link synchronously, issue the $QIO system service with the function code IO $_DEACCESS!IO$M_SYNCH. The channel is then free for subsequent communication with either the same or a different remote task.

A synchronous disconnect may be useful for master/slave communication, in which one task always sends data and its partner task always receives data. If the receiving task is notified of a synchronous disconnection, then all the data that was sent has been received. (The sending task, on the other hand, is not guaranteed that its partner received the data.) Because this notification is the only guarantee provided by this operation, VSI discourages using this operation in favor of a user-defined protocol to ensure completion of communication. In general, the receiver of the final message should break the logical link.

To abort the logical link, issue the $QIO system service with the function code IO$_DEACCESS!IO $M_ABORT. This type of disconnect indicates that all messages transmitted by the local transmitter may not have been received or acknowledged by the remote ECL before the logical link was disconnected. You should use this type of disconnect when the local task needs to reset the logical link to a known state. If the local task needs to ensure that the transmitted messages have been received and acknowledged by the remote ECL, the task can issue the system service $CANCEL on the channel before issuing the disconnect abort. Note that this does not guarantee the delivery of the received data to the remote task. It is the responsibility of cooperating tasks to agree on a protocol to ensure that the received data is delivered to the remote task.

Note that after either a synchronous disconnect or a disconnect abort, you can issue a new connection request if you did not deassign the I/O channel.

If you issue the $CANCEL system service to a channel over which a network name or object has been declared, the declaration is removed from the network database.

4.6.1.7. Terminating the Logical Link

You can issue the $DASSGN system service call to deassign the channel and terminate the logical link immediately. You issue the call only after all communication between the tasks is complete. The call releases the I/O channel, disassociates the mailbox from the channel, and terminates the logical link immediately. This operation is equivalent to using $CANCEL followed by $QIO IO$_DEACCESS!IO $M_ABORT.

The same status and error-reporting considerations apply to both nontransparent and transparent task-to-task communication. Refer to *Section 4.5.4.5, "Status and Error Reporting"* for information about status and error reporting.

4.6.2. System Service Calls for Nontransparent Operations

The following sections describe the OpenVMS system services you can use for nontransparent task communication over the network. Each description covers the use of the call, its format, the arguments associated with the call, and the return status information.

The following system services are not described in detail here, because their use does not change in a networking context.

- $CANCEL (Cancel I/O on Channel)

- $CREMBX (Create Mailbox and Assign Channel)
- $GETDVI (Get Device/Volume Information)

Note that $GETDVI performs the same function as the Get I/O Channel Information ($GETCHN) system service. However, VSI recommends that you use the $GETDVI system service.

After you issue a $CANCEL on a DECnet-Plus for OpenVMS logical link, the only valid operation is to disconnect or abort the logical link.

4.6.2.1. $ASSIGN (I/O Channel Assignment)

The $ASSIGN system service assigns a channel to refer to a logical link. You use this channel in all $QIO calls that communicate with a remote task. In addition, you can use the $ASSIGN system service call to associate a mailbox with the channel.

Format

`$ASSIGN devnam, chan, [acmode], [mbxnam]`

Arguments

devnam	Address of a quadword descriptor of a character string containing the string _NET: or a logical name for _NET:.
chan	Address of a word that is to receive the assigned channel number.
acmode	Access mode to be associated with this channel. The most privileged access mode used is the access mode of the caller. You can perform I/O operations on the channel only from equal or more privileged access modes.
mbxnam	Address of a character string descriptor for the physical name of the mailbox to be associated with the channel. This mailbox remains associated with the channel until the channel is deassigned ($DASSGN).

Return Status

SS$_NORMAL	The service completed successfully.
SS$_INSFMEM	There is not enough system dynamic memory to complete the request.
SS$_NOPRIV	The issuing task does not have the required privileges to create the channel.
SS$_NOSUCHDEV	The network device driver is not loaded (for example, the DECnet-Plus for OpenVMS software is not running currently on the local node).

4.6.2.2. $QIO (Requesting a Logical Link Connection)

The $QIO system service with the function code IO$_ACCESS requests a logical link connection to a target task. You can send 1 to 16 bytes of optional data to the target task at the same time that you issue the $QIO system service.

Format

`$QIO [efn], chan, func, [iosb], [astadr], [astprm], [p1], p2`

Arguments

efn	Number of the event flag to be set at request completion.
chan	Channel number associated with the logical link. Use the same channel number returned in the $ASSIGN call.
func	IO$_ACCESS.
iosb	Address of a quadword I/O status block that is to receive the completion status.
astadr	Entry point address of an AST routine that executes when the I/O operation completes. If specified, the AST routine executes at the access mode from which the $QIO service was requested.
astprm	AST parameter to be passed to the AST completion routine.
p1	Not used (omit the argument).
p2	Address of a quadword descriptor of the NCB (see *Section 4.6.1.4, "Using the Network Connect Block"*). Both the descriptor and the NCB must be in read/write storage.

Return Status

SS$_NORMAL	The service completed successfully.
SS$_CONNECFAIL	The connection to a network object timed out or failed.
SS$_DEVOFFLINE	The physical link is shutting down.
SS$_FILALRACC	A logical link is already accessed on the channel (that is, a previous connection is active on the channel).
SS$_INSFMEM	There is not enough system dynamic memory to complete the request.
SS$_INVLOGIN	The access control information was found to be invalid at the remote node.
SS$_IVDEVNAM	The NCB has an invalid format or content.
SS$_LINKEXIT	The network partner task was started, but exited before confirming the logical link (that is, $ASSIGN to SYS$NET).
SS$_NOLINKS	No logical links are available. The maximum number of logical links as set for the executor MAXIMUM LINKS parameter was exceeded.
SS$_NOPRIV	The issuing task does not have the required privileges to create a logical link to the designated target.
SS$_NOSUCHNODE	The specified node is unknown.
SS$_NOSUCHOBJ	The network object number is unknown at the remote node; or for a TASK= connect, the named DCL command procedure file cannot be found at the remote node.
SS$_NOSUCHUSER	The remote node could not recognize the login information supplied with the connection request.

SS$_PROTOCOL	A network protocol error occurred. This error is most likely due to a network software error.
SS$_REJECT	The network object rejected the connection.
SS$_REMRSRC	The link could not be established because system resources at the remote node were insufficient.
SS$_SHUT	The local or remote node is no longer accepting connections.
SS$_THIRDPARTY	The logical link was terminated by a third party (for example, the system manager).
SS$_TOOMUCHDATA	The task specified too much optional or interrupt data.
SS$_UNREACHABLE	The remote node is currently unreachable.

4.6.2.3. $QIO (Accepting Logical Link Connection Request)

The $QIO system service with the function code IO$_ACCESS accepts a logical link connection request from a source task. You can send 1 to 16 bytes of optional data to the source task at the same time that you issue the $QIO system service.

Format

```
$QIO [efn], chan, func, [iosb], [astadr], [astprm], [p1], p2
```

Arguments

efn	Number of the event flag to be set at request completion.
chan	Channel number associated with the logical link. Use the same channel number returned in the $ASSIGN call.
func	IO$_ACCESS.
iosb	Address of a quadword I/O status block that is to receive the completion status.
astadr	Entry point address of an AST routine that executes when the I/O operation completes. If specified, the AST routine executes at the access mode from which the $QIO service was requested.
astprm	AST parameter to be passed to the AST completion routine.
p1	Not used (omit the argument).
p2	Address of a quadword descriptor of the NCB (see *Section 4.6.1.4, "Using the Network Connect Block"*). Both the descriptor and the NCB must be in read/write storage.

Return Status

SS$_NORMAL	The service completed successfully.
SS$_DEVALLOC	The process cannot access the logical link specified in the NCB because that link is intended for another process.
SS$_EXQUOTA	The process does not have sufficient quota to complete the request.

SS$_INSFMEM	There is not enough system dynamic memory to complete the request.
SS$_IVDEVNAM	The NCB has an invalid format or content.
SS$_LINKABORT	The network partner task aborted the logical link.
SS$_LINKDISCON	The network partner task disconnected the logical link.
SS$_LINKEXIT	The network partner task exited.
SS$_NOSUCHNODE	The specified node is unknown.
SS$_PATHLOST	The path to the network partner task node was lost.
SS$_PROTOCOL	A network protocol error occurred. This error is most likely due to a network software error.
SS$_THIRDPARTY	The logical link connection was terminated by a third party (for example, the system manager).
SS$_TIMEOUT	The connection request did not complete within the required time.
SS$_UNREACHABLE	The remote node is currently unreachable.

4.6.2.4. $QIO (Rejecting a Logical Link Connection Request)

The $QIO system service with the function code IO$_ACCESS!IO$M_ABORT rejects a logical link connection request. You can send 1 to 16 bytes of optional data to the target task at the same time that you issue the $QIO system service.

Format

```
$QIO [efn], chan, func, [iosb], [astadr], [astprm], [p1], p2
```

Arguments

efn	Number of the event flag to be set at request completion.
chan	Channel number associated with the logical link. Use the same channel number returned in the $ASSIGN call.
func	IO$_ACCESS!IO$M_ABORT.
iosb	Address of a quadword I/O status block that is to receive the completion status.
astadr	Entry point address of an AST routine that executes when the I/O operation completes. If specified, the AST routine executes at the access mode from which the $QIO service was requested.
astprm	AST parameter to be passed to the AST completion routine.
p1	Not used (omit the argument).
p2	Address of a quadword descriptor of the NCB (see *Section 4.6.1.4, "Using the Network Connect Block"*). Both the descriptor and the NCB must be in read/write storage.

Return Status

SS$_NORMAL	The service completed successfully.

SS$_DEVALLOC	The process cannot access the logical link specified in the NCB because that link is intended for another process.
SS$_EXQUOTA	The process does not have sufficient quota to complete the request.
SS$_IVDEVNAM	The NCB has an invalid format or content.
SS$_LINKABORT	The network partner task aborted the logical link.
SS$_LINKDISCON	The network partner task disconnected the logical link.
SS$_LINKEXIT	The network partner task exited.
SS$_NOSUCHNODE	The specified node is unknown.
SS$_TIMEOUT	The connection request did not complete within the required time.
SS$_PATHLOST	The path to the network partner task node was lost.
SS$_PROTOCOL	A network protocol error occurred. This error is most likely due to a network software error.
SS$_THIRDPARTY	The logical link connection was terminated by a third party (for example, the system manager).
SS$_UNREACHABLE	The remote node is currently unreachable.

4.6.2.5. $QIO (Sending a Message to a Target Task)

The $QIO system service with the function code IO$_WRITEVBLK or IO$_WRITEVBLK!IO $M_INTERRUPT or IO$_WRITEVBLK!IO$M_MULTIPLE sends a message to a target task. Refer to *Section 4.5.5.2, "$QIO (Sending a Message to a Target Task)"* for the format of this call, its arguments, and possible return status codes.

4.6.2.6. $QIO (Receiving a Message from a Target Task)

The $QIO system service with the function code IO$_READVBLK or IO$_READVBLK!IO $M_MULTIPLE receives a message from a target task. Refer to *Section 4.5.5.3, "$QIO (Receiving a Message from a Target Task)"* for the format of this call, its arguments, and possible return status codes.

4.6.2.7. $QIO (Sending an Interrupt Message to a Target Task)

The $QIO system service with the IO$_WRITEVBLK!IO$M_INTERRUPT function code sends a 1- to 16-byte interrupt message to a target task. If the remote node is an OpenVMS operating system, the message is placed in the mailbox associated with the target task.

Format

```
$QIO [efn], chan, func, [iosb], [astadr], [astprm], p1, p2
```

Arguments

efn	Number of the event flag to be set at event completion.
chan	Channel number associated with the logical link. Use the same channel number returned in the $ASSIGN call.
func	IO$_WRITEVBLK!IO$M_INTERRUPT.

iosb	Address of a quadword I/O status block that is to receive the completion status.
astadr	Entry point address of the AST routine that executes when the I/O operation completes. If specified, the AST routine executes at the access mode from which the $QIO service was requested.
astprm	AST parameter to be passed to the AST completion routine.
p1	Buffer address.
p2	Buffer length (1 to 16 bytes).

Return Status

SS$_NORMAL	The service completed successfully.
SS$_ABORT	The I/O request has been aborted by a $DASSGN or$CANCEL call.
SS$_FILNOTACC	No logical link is associated with the channel.
SS$_INSFMEM	Enough memory to buffer the message could not be allocated.
SS$_LINKABORT	The network partner task aborted the logical link.
SS$_LINKDISCON	The network partner task disconnected the logical link.
SS$_LINKEXIT	The network partner task exited.
SS$_NOSOLICIT	DECnet could not accept an interrupt message at this time.
SS$_TOOMUCHDATA	The task specified too much interrupt data.
SS$_PATHLOST	The path to the network partner task node was lost.
SS$_PROTOCOL	A network protocol error occurred. This error is most likely due to a network software error.
SS$_THIRDPARTY	The logical link connection was terminated by a third party (for example, the system manager).

4.6.2.8. $QIO (Synchronously Disconnecting a Logical Link)

The $QIO system service with the function code IO$_DEACCESS!IO$M_SYNCH synchronously disconnects the logical link. All pending messages are transmitted to the remote node before the link is disconnected.

You can send 1 to 16 bytes of optional data to the task from which you are disconnecting at the same time you issue this $QIO system service.

Format

```
$QIO [efn], chan, func, [iosb], [astadr], [astprm], [p1], [p2]
```

Arguments

efn	Number of the event flag to be set at event completion.
chan	Channel number associated with the logical link. Use the same channel number returned in the $ASSIGN call.

func	IO$_DEACCESS!IO$M_SYNCH.
iosb	Address of a quadword I/O status block that is to receive the completion status.
astadr	Entry point address of the AST routine that executes when the I/O operation completes. If specified, the AST routine executes at the access mode from which the $QIO service was requested.
astprm	AST parameter to be passed to the AST completion routine.
p1	Not used (omit the argument).
p2	Address of a descriptor of a counted ASCII string of optional user data. Both the string and its descriptor must be in read/write storage.

Return Status

SS$_NORMAL	The service completed successfully.
SS$_FILNOTACC	No logical link is associated with the channel.

4.6.2.9. $QIO (Aborting a Logical Link)

The $QIO system service with the function code IO$_DEACCESS!IO$_ABORT terminates the logical link. Note, however, that the DEACCESS function completes only after all pending I/O operations complete, even if you specify IO$_ABORT. First, issue the $CANCEL system service call to cancel I/O operations on the logical link and then issue this call to terminate the logical link.

You can send 1 to 16 bytes of optional data to the task from which you are disconnecting at the same time that you issue this $QIO system service call.

Format

```
$QIO [efn], chan, func, [iosb], [astadr], [astprm], [p1], [p2]
```

Arguments

efn	Number of the event flag to be set at event completion.
chan	Channel number associated with the logical link. Use the same channel number returned in the $ASSIGN call.
func	IO$_DEACCESS!IO$M_ABORT.
iosb	Address of a quadword I/O status block that is to receive the completion status.
astadr	Entry point address of the AST routine that executes when the I/O operation completes. If specified, the AST routine executes at the access mode from which the $QIO service was requested.
astprm	AST parameter to be passed to the AST completion routine.
p1	Not used (omit the argument).
p2	Address of a quadword descriptor of a counted string of optional user data. Both the string and its descriptor must be in read/write storage.

Return Status

SS$_NORMAL	The service completed successfully.

SS$_FILNOTACC	No logical link is associated with the channel.

4.6.2.10. $QIO (Declaring a Network Name or Object Number)

DECnet-Plus for OpenVMS Alpha supports the following ACPCONTROL functions:

- NFB$C_DECLOBJ – declare object by number
- NFB$C_DECLNAME – declare object by name

The $QIO system service with the function code IO$_ACPCONTROL assigns a network name or object number to the task, thereby making it eligible to process multiple inbound connection requests. You must associate a mailbox with the I/O channel. All inbound connection requests are placed in the mailbox associated with the channel over which this I/O function is issued. You need the SYSNAM privilege to declare a name or object number.

MACRO programmers should be aware that, whenever a logical link is established, you should obtain its device unit number (for example, 18 from _NET18:) by using the $GETDVI system service, because unit numbers and not channel numbers appear in mailbox messages. Use this system service call where a single mailbox is being used for many logical links. The unit number could be used as a key into a database that keeps track of multiple links.

Format

```
$QIO [efn] ,chan ,func ,[iosb] ,[astadr] ,[astprm] ,p1 ,p2
```

Arguments

efn	Number of the event flag to be set at event completion.
chan	Word containing the channel number associated with the logical link. Use the same channel number assigned in the$ASSIGN call.
func	IO$_ACPCONTROL.
iosb	Address of a quadword I/O status block that is to receive the completion status.
astadr	Entry point address of the AST routine that executes when the I/O operation completes. If specified, the AST routine executes at the access mode from which the $QIO service was requested.
astprm	AST parameter to be passed to the AST completion routine.
p1	Address of a quadword descriptor of a 5-byte block consisting of a function type (one byte) and a longword parameter. The function type is a symbol defined by the $NFBDEF macro in SYS$LIBRARY:LIB.MLB. The format of the 5-byte block for declaring a name is: `.BYTE NFB$C_DECLNAME.LONG 0`. The format of the 5-byte block for declaring an object number is: `.BYTE NFB $C_DECLOBJ.LONG object-number`. The object number is a number reserved for customer use in the range of 128 to 255. This 5-byte buffer and its descriptor should be in read/write storage.
p2	Address of a quadword descriptor of the network name (maximum of 12 characters). You should not supply this argument for the DECLOBJ function. Both the name and its descriptor must be in read/write storage.

Return Status

SS$_NORMAL	The service completed successfully.
SS$_BADPARAM	One of the QIO parameters has an invalid value.
SS$_ILLCNTRFUNC	The control function is invalid.
SS$_NOMBX	A name or object number is being declared using a channel without an associated mailbox.
SS$_NOPRIV	The issuing process does not have the SYSNAM privilege.

4.6.2.11. $DASSGN (Terminating a Logical Link)

The $DASSGN system service terminates all pending operations to send and receive data, aborts the logical link immediately, and frees the channel associated with that link. Refer to *Section 4.5.5.4, "$DASSGN (Disconnecting a Logical Link)"* for the format of this call, its arguments, and possible return status codes.

4.7. Designing Tasks

The following sections contain a command procedure and two user program examples designed to perform task-to-task communications over the network.

The command procedure and the two user program examples illustrate transparent operations.

4.7.1. DCL Command Procedure for Task-to-Task Communication

As described in *Section 4.5, "Performing Transparent Task-to-Task Operations"*, you can write command procedures in DCL to execute transparent task-to-task operations. You can use the following command procedure, called SHOWBQ.COM, to perform such an operation. You can use SHOWBQ.COM for task-to-task communication by entering a task specification string in a TYPE command. For example:

```
$ TYPE TRNTO"BROWN JUNE"::"TASK=SHOWBQ"
```

In this command procedure, SYS$OUTPUT is equated to SYS$NET in user mode to allow the SHOW QUEUE image to communicate over the logical link by opening SYS$OUTPUT. When the SHOW QUEUE image exits, the temporary definition of SYS$OUTPUT is deleted. In other words, only one DCL image can use the logical link as the communication path to the requester at the other node.

```
        SHOWBQ.COM
$ !
$ !     This  command procedure  returns status information about
$ !     jobs  entered in  batch  queues  on  the  system where it
$ !     executes.   It  may  be   run  interactively as  a command
$ !     procedure,  submitted  as a  local or remote batch job, or
$ !     invoked  as a  "remote task"  to display information about
$ !     another system.  For example:
$ !
$ !     $ @SHOWBQ
$ !     $ SUBMIT SHOWBQ
$ !     $ SUBMIT/REMOTE node::SHOWBQ
$ !     $ TYPE node::"TASK=SHOWBQ"
$ !
```

```
$ IF F$MODE() .EQS. "NETWORK" THEN GOTO NET
$ SHOW QUEUE/BATCH/BRIEF/ALL$ EXIT$NET:
$ DEFINE/USER SYS$OUTPUT SYS$NET$ SHOW QUEUE/BATCH/BRIEF/ALL
$ EXIT
```

4.7.2. FORTRAN Program for Task-to-Task Communication

Example 4.2, "FORTRAN Task-to-Task Communication" shows an example of VAX FORTRAN transparent communication. In the FORTRAN source task that initiates the logical link request, you use a standard open statement to specify the remote task to which you want to connect. In turn, the remote task issues an open statement to complete the logical link connection. A coordinated set of read and write operations enable the exchange of information over the link. To terminate the connection, the source task executes a close statement to break the logical link. When the remote task receives this close statement, it issues a close statement, which completes the link termination process. The remainder of this section discusses the statements that you would use to develop such an application.

Example 4.2. FORTRAN Task-to-Task Communication

```
        PROGRAM TEST3.FORC

C       This program prompts the user for the part number of an item
C       in inventory and responds with the number of units in stock.
C       The information is obtained by communicating with a program
C       (TEST4) on another node that has access to the inventory data.C
C       Before running this program, the logical name TASK must be
C       defined to refer to the target program. For example:C
C       $ DEFINE TASK "TRNTO::""TASK=TEST4"""
C       $ RUN TEST3
C
        CHARACTER PARTNO*5
        INTEGER PARTCOUNT
C
100     FORMAT (/,'$Enter part number: ')
200     FORMAT (A)
300     FORMAT (I4)
400     FORMAT ('0Inventory for part number ',A,' is: ',I4)
C
C       Establish a logical link with the target task.
C
   ❶  OPEN (UNIT=1,NAME='TASK',ACCESS='SEQUENTIAL',
     1       FORM='FORMATTED',CARRIAGECONTROL='NONE',TYPE='NEW')
C
C       Prompt the user for a part number, send it to the target task,
C       read reply of quantity on hand, and finally display the answer
C       for the user. Repeat the cycle until the user enters 'EXIT' for
C       a part number.
C
10      TYPE 100
        ACCEPT 200, PARTNO
        IF (PARTNO .EQ. 'EXIT') GOTO 20
    ❷ WRITE (1,200) PARTNO
        READ (1,300) PARTCOUNT
        TYPE 400, PARTNO, PARTCOUNT
        GOTO 10C
C       Disconnect the logical link.
```

```
C
20  ❸ CLOSE (UNIT=1)
        END
$!
$ !     ************************************************************
$ !     TEST4.COM
$ !
$ !     This command procedure executes the program TEST4 after
$ !     being started by a task-to-task connection request issued
$ !     by TEST3.
$ !
❹ $ RUN SYS$LOGIN:TEST4.EXE
$ EXIT
        PROGRAM TEST4.FOR
C
C       Test4 is the target program that communicates with TEST3.
C       For each part number received from the source task, the
C       number of units in stock is determined, and this value is
C       returned.
C
C       To complete the logical link with its initiator, this program
C       uses the logical name SYS$NET as the file specification in an
C       open statement.
C        CHARACTER PARTNO*5        INTEGER PARTCOUNT
C
100     FORMAT (A)
200     FORMAT (I4)
C
C       Complete the logical link connection.
C    ❺ OPEN (UNIT=1,NAME='SYS$NET',ACCESS='SEQUENTIAL',
     1       FORM='FORMATTED',CARRIAGECONTROL='NONE',TYPE='OLD')
C
C       Process requests until end-of-file is reached. (This is the
C       error condition returned when the source task breaks the
C       logical link connection.)
C
10  ❷ READ    (1,100,END=20) PARTNOC
C       Perform appropriate processing to obtain the part count value
C       and transmit this back to the source task.
C
        CALL INSTOCK (PARTNO,PARTCOUNT)
    ❷ WRITE (1,200) PARTCOUNT
        GOTO 10C
C       Disconnect the logical link.
C
20  ❸ CLOSE   (UNIT=1)
        END
```

Notes on *Example 4.2, "FORTRAN Task-to-Task Communication"*

❶ The source task, TEST3, requests a logical link connection to the target task, TEST4.

❷ TEST3 and TEST4 send and receive data messages.

❸ TEST3 disconnects the logical link and thereby terminates the communication process.

❹ When the remote node receives a connection request, the command procedure TEST4.COM is executed. This command procedure must reside in the default directory associated with the account

being accessed. TEST4.COM contains a RUN statement that causes the program TEST4.EXE to be executed.

❺ TEST4 completes the logical link connection through SYS$NET. Note that the unit numbers in the source and target programs need not be the same.

Because DECnet-Plus for OpenVMS translates system-dependent language calls into the same set of messages that permit task-to-task communication, any task programmed in VAX MACRO or one of the higher-level languages can communicate with a remote task programmed in the same or a different language. More specifically, for communication between tasks that run on OpenVMS nodes, the language in which you access the network has no effect on the communication process. The VAX FORTRAN source task in *Example 4.2, "FORTRAN Task-to-Task Communication"* could just as easily communicate with a MACRO task on node TRNTO.

Chapter 5. Introduction to OSI Transport Programming

The OSI transport service enables an application running on an OpenVMS system to exchange data with an application running on another host. A **host** is:

- An OpenVMS system running OSI transport, or
- Another computer running software that implements the OSI transport Protocol, and uses lower layers corresponding to those that OSI transport supports.

See *VSI DECnet-Plus for OpenVMS Introduction and User's Guide* for a description of how the OSI transport service operates. This chapter discusses the OSI transport programming interface functions.

Applications on an OpenVMS system communicate with OSI transport using standard OpenVMS system service calls. These calls are to the $ASSIGN, $DASSGN, $QIO and $QIOW system services. *Table 5.1, "Summary of System Service Calls in MACRO Format"* at the end of this chapter shows all the system service calls used by OSI transport.

The OSI transport service interface is similar to the DECnet-Plus for OpenVMS service interface; these applications running over DECnet-Plus for OpenVMS can be used over OSI transport, with few alterations.

Note

The abbreviation $QIO(W) refers to cases where either a $QIO or a $QIOW call can be used.

5.1. An Overview of the OSI Transport Programming Interface

OpenVMS OSI transport service is an interface into the OSI transport protocol engine. In the following chapters, when possible, a distinction is made between the two. OSI transport is used to describe only the OSI transport protocol engine. OpenVMS OSI transport service, or transport service, is used primarily to describe the interface. However, where information applies to both the interface and the protocol engine, "transport service" is used.

OpenVMS OSI transport service is implemented as an OpenVMS pseudodevice driver. This means that a task will communicate with OpenVMS OSI transport service using OpenVMS channels, $QIO(W) system service calls, and OpenVMS mailboxes.

Before a task can communicate with OpenVMS OSI transport service, it must assign a logical OpenVMS channel to the OpenVMS OSI transport service pseudodevice. Before it can exchange data with the remote host, a transport service connection must be established using that channel.

A user on an OpenVMS OSI transport service system issues a transport service connection by making a connection request to OpenVMS OSI transport service. This connection request must provide OpenVMS OSI transport service with certain information. In particular:

- A network address for the remote host. OpenVMS OSI transport service uses this address to deliver the connection request.

- A transport service access point identifier (TSAP-ID). This uniquely identifies the responding user on the remote system.

The responding user can either accept the connection request or reject it. The responding user can also get more information about the connection request before making a decision.

If the connection is accepted, OpenVMS OSI transport service informs the initiating user that the transport service connection is established. The communicating users can now start to transfer data.

Users can transfer two types of data: normal and expedited. Normal data is used for the main data exchange traffic. Expedited data is sent outside the normal data flow and is typically reserved for emergency messages.

Once the data has been transferred satisfactorily, either user can disconnect the transport service connection.

Once a connection has been disconnected, the OpenVMS channel to OpenVMS OSI transport service can be re-used for another transport service connection.

5.2. The OpenVMS OSI Transport Service Device, Channels and Mailboxes

Before an application can use OpenVMS OSI transport service, it must assign an OpenVMS channel to the OpenVMS OSI transport service pseudodevice. This pseudodevice has the name VMS OSIT $DEVICE.

A VMS OSIT$DEVICE channel connects only one transport service at a time. Similarly, each transport service connection can only use one channel at a time. However, once a transport service connection has been disconnected, the channel can be re-used for another transport service connection.

Usually a mailbox is associated with the VMS OSIT$DEVICE channel. The OpenVMS OSI transport service uses the mailbox to deliver:

- Status messages about outbound connection requests and established transport service connections (for example, a disconnect message)
- Details of inbound connection requests from the remote host
- Expedited data

A transport service user does not have to use a mailbox if it will not receive inbound connections, or receive expedited data. Otherwise, it should use a mailbox.

One mailbox may be shared among several VMS OSIT$DEVICE channels.

5.3. Using $QIO or $QIOW System Service Calls

Tasks make requests to the OpenVMS OSI transport service using $QIO or $QIOW system service calls over the OpenVMS channel assigned to the OpenVMS OSI transport service.

The task may make asynchronous requests using $QIO calls or synchronous requests using $QIOW calls.

- $QIO calls return control to the initiating user once a request has been accepted or rejected by the OSI transport service. When the OpenVMS OSI transport service has accepted or rejected a request, a status code is returned in Register 0 (R0).

  If the OpenVMS OSI transport service accepts the request, it then attempts to process it. When it either succeeds or fails, it returns a status code in the I/O status block (IOSB) for the $QIO call. If the $QIO call specified an AST routine, it will be called at this point; alternatively, an event flag could be set.

- $QIOW calls return control to the initiating user only when the request has actually been processed by the OpenVMS OSI transport service, or if the OpenVMS OSI transport service does not accept the call. At that point, the OpenVMS OSI transport service returns a status code in the IOSB.

Whether you use $QIO or $QIOW calls depends on whether your task has useful work it might do while waiting for a request to be completed.

One typical use of $QIO calls is to make $QIO (IO$_READVBLK) calls, with an AST parameter, on the mailbox. This allows status messages to be delivered without affecting other processing.

5.4. NCBs and Item Lists

In certain $QIO (W) calls, you are required to use either a **network connect block (NCB)** or an **item list** to supply information to the OpenVMS OSI transport service. For example, a connection request uses an NCB or an item list to provide essential addressing information, as well as other optional information.

An item list has two main advantages over an NCB:

- You can supply more information in an item list; in particular, information needed to negotiate options and protocol classes with the remote host.
- You can use an item list to request detailed information about an inbound connection request.

5.5. Issuing an Outbound Connection Request

Once it has assigned a channel to the OpenVMS OSI transport service, a task can issue a connection request. It does this by issuing a $QIO(W)(IO$_ACCESS) call.

In this call, the initiating user must supply:

- The address for the remote host tells the OpenVMS OSI transport service which network to use to deliver the connection request, and what address to use on that network.
- The TSAP-ID uniquely identifies the responding user on the remote system. The TSAP-ID you provide must be known to the system with which you are trying to communicate.
- User data is optional information for the remote user. It is typically used to provide helpful information, for example, the amount of data that will follow.

The connection request supplies the address, TSAP-ID, and user data in either an NCB or an item list.

5.5.1. The Status of an Outbound Connection Request

An initiating user can get information about the status of the connection request in two ways:

- From the IOSB identified in the $QIO(W)(IO$_ACCESS) call. This will contain a status code to show that the connection request has been successfully processed, or why it has failed.
- From the mailbox, if you specified one. When a connection request is accepted or rejected, the OpenVMS OSI transport service sends a mailbox message to the initiating user.

5.6. Receiving an Inbound Connection Request

Before a transport service user can receive an inbound connection request, it must associate itself with a TSAP. A TSAP uniquely identifies a transport user. Inbound connection requests usually contain a The OpenVMS OSI transport service locates the task by matching the TSAP-ID in the connection request with one associated with a transport service user.

There are two types of TSAP association: **active** and **passive**.

- An active association is created by an executing task. The task issues a $QIO call asking the OpenVMS OSI transport service to associate it with a TSAP. Once this has been done, there is an active association between the TSAP, identified by its TSAP-ID and the process identifier (PID) for the task. When an inbound connection request arrives that names this TSAP-ID, the OpenVMS OSI transport service uses the TSAP-ID to locate the executing task.
- A passive association is created when the system manager creates an OpenVMS OSI transport service application entity. The entity associates the name of an image or command file with a TSAP-ID. The OpenVMS OSI transport service activates this file when an inbound connection request arrives that names the relevant TSAP-ID.

5.6.1. Examining an Inbound Connection Request

When an inbound connection request arrives, the OpenVMS OSI transport services ends information about it to the transport service user identified by the TSAP-ID. This information is in the form of an NCB. The NCB contains the address of the initiating user and any user data supplied.

The task can use the information in the NCB to decide whether to accept or reject the connection request, or to examine the connection request further to get more details.

To obtain more details, a responding task should make a $QIO(W)(IO$_SENSEMODE) call. This will return the protocol classes and the options proposed in the inbound connection request. This call is typically used to get information for class or options negotiation.

A task must supply an item list with a $QIO(W)(IO$_SENSEMODE) call. One of the items must be the **transport service connection identifier** (**TC-ID**) for the connection. To find the item list, the task must analyze the NCB supplied by the OpenVMS OSI transport service for the inbound connection request.

5.6.2. Accepting an Inbound Connection Request

If the inbound connection request is satisfactory, a task accepts it by issuing a $QIO(W) call with the function code IO$_ACCESS. The accept call must supply either an item list or an NCB:

- If it supplies an item list, one of the items must be the TC-ID supplied in the NCB for the inbound connection request.
- If it supplies an NCB, it should use the NCB supplied by the OpenVMS OSI transport service for the inbound connection request.

The task may also supply user data in the NCB or item list.

Note that if the responding user is only passively associated with a TSAP, no VMS OSIT$DEVICE channel will have been assigned. This is because passively associated tasks only start running after a connection request arrives. Therefore, the passively associated task must first assign a channel before it can accept or reject a connection request.

5.6.3. Rejecting an Inbound Connection Request

A responding user might find the inbound connection request unacceptable. For example, a connection request may have requested an unacceptable option. To reject an inbound connection request, the task issues a $QIO(W)(IO$_ACCESS) call with the function code IO$M_ABORT.

The reject call must supply either an item list or an NCB:

- If it supplies an item list, one of the items must be the TC-ID supplied in the NCB for the inbound connection request.
- If it supplies an NCB, it should use the NCB supplied by the OpenVMS OSI transport service for the inbound connection request.

The task may also supply user data in the NCB or item list. This might contain a reason for the disconnection.

5.7. Exchanging Data

Once a transport service connection has been established on a VMS OSIT$DEVICE channel, either the initiating or responding user may transfer data. The main traffic is usually carried using normal data. Expedited data is reserved for sending short messages out of the normal data flow.

A task sends normal data by issuing $QIO or $QIOW(IO$_WRITEVBLK) calls to the OpenVMS OSI transport service. It receives normal data by issuing $QIO or $QIOW(IO$_READVBLK) calls to the OpenVMS OSI transport service.

A task sends expedited data by issuing a $QIO(W)(IO$_WRITEVBLK!IO$M_INTERRUPT) call to the OpenVMS OSI transport service. It reads IO$M_INTERRUPT expedited data by issuing a $QIO(W)(IO$_READVBLK) call to the mailbox. This is because the mailbox associated with the VMS OSIT $DEVICE channel receives the expedited data. If your task does not issue this call, it will not receive any expedited data.

5.8. Canceling I/O on a Channel

A task can cancel outstanding I/O requests at any time during the lifetime of a transport service connection by issuing a $CANCEL call on the channel used for the connection.

$CANCEL does the following:

- Immediately cancels outstanding $QIO(W)(IO$_WRITEVBLK) and $QIO(W)(IO$_READVBLK) calls using the specified channel
- Deletes any actively associated TSAPs that were created using that channel
- Disconnects the connection

5.9. Disconnecting a Transport Service Connection

Either end of the OpenVMS OSI transport service connection may disconnect at any time. A transport service user on an OpenVMS OSI transport service system can do this by issuing a $QIO(W)(IO $_DEACCESS) call on the VMS OSIT$DEVICE channel associated with the transport service connection. The user doing the disconnecting may supply optional data with the disconnection. However, it is not guaranteed that it will be delivered.

5.9.1. Receiving a Disconnection

An OpenVMS OSI transport service user receiving a disconnection is informed by a mailbox message. The mailbox message may also contain optional user data supplied in the disconnection request. If a task issues a $QIO(W) call after the disconnect, the call will fail, with a completion code in the IOSB that indicates the cause of the failure.

If there is no associated mailbox, the task will not be informed directly of the disconnection. However, when the next $QIO call fails, the IOSB completion code will give the reason.

When the task receives the disconnection message, it should also issue a $QIO(W)(IO$_DEACCESS); this will make the VMS OSIT$DEVICE channel ready for re-use.

5.9.2. Results of Disconnection

Once a $QIO(W)(IO$_DEACCESS) on a VMS OSIT$DEVICE channel has been issued, a task may use that channel to set up another transport service connection, without using another $ASSIGN call.

Any $QIO requests outstanding when the transport service connection is disconnected will be aborted, with an abort status in the IOSB.

Note that issuing a $QIO(W)(IO$_DEACCESS) on a VMS OSIT$DEVICE channel does not affect any active associations with TSAPs made on that channel.

5.10. Deassigning a Channel

If a task has no further use for the VMS OSIT$DEVICE channel, then it should deassign the channel by issuing a $DASSGN system service call.

Deassigning a channel automatically ends the connection. It also deletes any active association with TSAPs made on that channel.

5.11. System Service Calls

Table 5.1, "Summary of System Service Calls in MACRO Format" shows the system service calls you can use with OSI transport.

Table 5.1. Summary of System Service Calls in MACRO Format

Purpose of the Call		Call (function code!modifier)
Assign a channel to VMS OSIT $DEVICE		$ASSIGN

Purpose of the Call		**Call (function code!modifier)**
Cancel I/O on a channel		$CANCEL
Create a mailbox		$CREMBX
Deassign a channel from VMS OSIT$DEVICE after concluding a transport connection		$DASSGN
Request an outbound transport service connection		$QIO (IO$_ACCESS)
Examine an inbound transport service connection		$QIO (IO$_SENSEMODE)
Accept an inbound transport service connection		$QIO (IO$_ACCESS)
Reject an inbound transport service connection		$QIO (IO$_ACCESS !IO $M_ABORT)
Associate a task with a TSAP		$QIO (IO$_ACPCONTROL)
Conclude a transport service connection		$QIO (IO$_DEACCESS !IO $M_ABORT)
Read normal data		$QIO (IO$_READVBLK)
Read a mailbox message		$QIO (IO$_READVBLK)
Send normal data		$QIO (IO$_WRITEVBLK)
Send expedited data		$QIO (IO$_WRITEVBLK !IO $M_INTERRUPT)
Translate a logical name		$TRNLNM

Chapter 6. Programming Guidelines

This chapter explains how to use the OpenVMS system services to perform task-to-task communication:

- Assign an I/O channel to OpenVMS OSI transport service
- Associate a mailbox with that channel
- Initiate an outbound transport service connection
- Accept or reject an inbound transport service connection request
- Examine an inbound transport service connection request
- Exchange data with a remote transport service user
- Disconnect a transport service connection
- Deassign the I/O channel

Appendix E, "Programming Examples" offers you:

- Access to a directory of example programs written in a variety of programming languages.
- One language example script (written in the C programming language).

The example script is divided into separate program modules. Each module illustrates one or more of the functions needed in your application.

The modules in the example script are referenced throughout this chapter.

6.1. Including Definitions of Transport Service Symbols

Before an application can use OpenVMS OSI transport service, it must contain definitions for OpenVMS OSI transport service-specific symbols. If it uses a mailbox, it should also include definitions of mailbox message types. You include these definitions in an application by using standard library files.

6.1.1. OpenVMS OSI Transport Service-specific Symbols

OpenVMS OSI transport service-specific symbols begin with the prefix VMS OSIT$. These symbols are contained in library files with the name SYS$LIBRARY:VMS OSIT. *x*, where *x* identifies a programming language. For example, OpenVMS OSI transport service-specific symbols for MACRO programs are in SYS$LIBRARY:VMS OSIT.MAR.

You must make the OpenVMS OSI transport service-specific symbols available in your program by including the appropriate SYS$LIBRARY:VMS OSIT file.

6.1.2. Mailbox Message Types

Mailbox message types are defined by the $MSGDEF macro. A list of the mailbox message types is in *Appendix B, "Mailbox Message Types"*.

6.1.3. Mailbox Messages

See *Appendix B, "Mailbox Message Types"* for a description of the structure of a mailbox message, and a list of mailbox message types.

See *Section E.1.25, "Report Mailbox Message Type"* for an example routine to report a mailbox message type and take the appropriate action.

6.2. Assigning a Channel and Setting Up a Mailbox

Before a task can establish a transport service connection, it must assign a channel to communicate with OpenVMS OSI transport service. Each transport service connection needs its own channel to OpenVMS OSI transport service. If your task will have several transport service connections operating at the same time, it must assign a channel for each of them.

If a task is to receive inbound connection requests, it will usually associate a mailbox with the channel or channels to OpenVMS OSI transport service. The mailbox is the standard OpenVMS mailbox facility.

A task can associate a different mailbox with each channel, or share one mailbox among several channels.

OpenVMS OSI transport service uses the mailbox to deliver:

- Status messages; for example, connect, confirm, failure, error, or disconnect messages
- Inbound connection request details for tasks actively associated with a TSAP (see *Section 6.5.1, "Transport Service Access Points"*)
- Expedited data

A task does not need a mailbox if all of the following are true:

- It will not receive expedited data.
- It will not receive multiple inbound connection requests.
- It does not need to receive mailbox status messages.

6.2.1. Assigning a Channel to OpenVMS OSI Transport Service

OpenVMS OSI transport service is implemented as the device driver for the pseudodevice, VMS OSIT $DEVICE. To communicate with OpenVMS OSI transport service, your task must assign an OpenVMS channel to VMS OSIT$DEVICE. You use this channel to establish a transport service connection, transfer data, and end the connection. You need a separate VMS OSIT$DEVICE channel for each transport service connection.

When you assign a channel to VMS OSIT$DEVICE, the following happens:

- OpenVMS creates a new pseudodevice called OS *n*, where *n* is a unique **unit number** for the channel.

  You can see the unit numbers for OS channels by entering the DCL command, SHOW DEVICE OS.

- OpenVMS assigns a **channel number** to the channel. This is different from the unit number.

- If you have specified a mailbox device name, OpenVMS associates a mailbox with the channel (see *Section 6.2.4, "Associating One Mailbox with Several Channels"*).

You must never explicitly assign a channel to the pseudodevice OS *n*. You must always identify the OpenVMS OSI transport service device by the device name VMS OSIT$DEVICE.

6.2.2. Assigning a Channel without Creating a Mailbox

You can assign a channel to VMS OSIT$DEVICE using the system service $ASSIGN. If you use $ASSIGN, you need to create a mailbox separately, using the system service $CREMBX.

You need to supply the following in the $ASSIGN call:

- The name of the device, VMS OSIT$DEVICE

- If you want a mailbox, the name of a previously created mailbox

See *Section E.1.7, "Assign a Channel to OSI Transport"* for an example routine to assign a channel to VMS OSIT$DEVICE, and associate it with a previously created mailbox.

See *Section E.1.8, "Create Mailbox and Post a Read"* for an example routine to create a mailbox separately, using $CREMBX.

6.2.3. Assigning a Channel and Creating a Mailbox

You can assign a channel to VMS OSIT$DEVICE, create a mailbox, and associate the mailbox with the channel in one operation. You do this by using the run-time library routine LIB$ASN_WTH_MBX. This routine:

- Creates a temporary mailbox, and assigns a channel to it.

- Assigns a channel to VMS OSIT$DEVICE.

- Associates the mailbox channel with the channel to VMS OSIT$DEVICE.

You supply the device name, VMS OSIT$DEVICE.

LIB$ASN_WTH_MBX returns two channel numbers: one for the mailbox channel and one for the channel to VMS OSIT$DEVICE.

Each LIB$ASN_WTH_MBX call creates a new, unique mailbox. This is because this routine does not allow you to use logical names; it simply assigns a channel number. If you create a mailbox with a logical name (using the $CREMBX system service), other tasks might use the same logical name when creating their mailboxes. The result would be that these tasks all use the same mailbox, and interfere with each other's mailbox messages.

If you want to use $CREMBX to create a mailbox, do not supply a logical name. $CREMBX will then simply return a mailbox channel number.

Once you have created the mailbox, you can associate it with other VMS OSIT$DEVICE channels, using the $ASSIGN system service. See *Section 6.2.2, "Assigning a Channel without Creating a Mailbox"* for details.

6.2.4. Associating One Mailbox with Several Channels

You can associate the same mailbox with a number of VMS OSIT$DEVICE channels. You might want to do this if your task will create a number of transport service connections. Follow these steps:

1. Use LIB$ASN_WTH_MBX to create a mailbox and associate it with a VMS OSIT$DEVICE channel. Alternatively, use $CREMBX to create a mailbox, and $ASSIGN to assign a VMS OSIT $DEVICE channel and associate it with the mailbox.

   This will return two channel numbers: one for the mailbox channel and one for the VMS OSIT $DEVICE channel.

2. Use the system service $GETDVI to find the physical device name of the mailbox assigned. Supply the number of the mailbox channel to $GETDVI.

   $GETDVI will return the **mailbox physical device name**.

3. Each time you use $ASSIGN to assign another channel to VMS OSIT$DEVICE, supply the mailbox physical device name in the *mbxnam* argument.

6.2.5. Reading the Mailbox

To read the mailbox, you issue either a $QIO(IO$_READVBLK) or a $QIOW(IO$_READVBLK) call. You need to supply the mailbox channel number.

You usually do not wait for a task to read the mailbox asynchronously, that is, by issuing a $QIO read call. This call should specify an AST routine to process information written to the mailbox. The task can continue processing immediately after it has issued the $QIO call; it does not need to wait until OpenVMS OSI transport service delivers a message to the mailbox. The AST routine deals with the mailbox message.

You should always have a $QIO(IO$_READVBLK) call, with an AST routine, outstanding on the mailbox. If you do this, messages will be read as soon as OpenVMS OSI transport service deposits them. If you do not, you may not receive some mailbox messages.

A task can also read the mailbox synchronously, by issuing a $QIOW read call. In this case, the task will wait until OpenVMS OSI transport service delivers a message to the mailbox, and only return to processing when a message is delivered.

See *Section E.1.24, "Read Mailbox"* for an example routine that reads the mailbox. See *Section E.1.26, "Wait for Mailbox Message and Read Mailbox"* for an example AST routine that waits for a mailbox message and then reads the mailbox.

6.2.6. Reading a Mailbox Associated with Several Channels

If you have one mailbox associated with several VMS OSIT$DEVICE channels, you need to match each mailbox message to the relevant channel. You do this by matching OS unit numbers to OpenVMS channel numbers.

Each mailbox message contains an OS unit number in the form OS *n*. This number uniquely identifies a VMS OSIT$DEVICE channel. However, to OpenVMS, the channel is identified by the OpenVMS channel number.

You need to provide a data structure to match OS unit numbers to OpenVMS channel numbers. One way of doing this would be to have an array of unit numbers and channel numbers.

You can get unit numbers and channel numbers as follows:

- When you assign a channel to VMS OSIT$DEVICE, $ASSIGN returns the OpenVMS channel number.
- Each time you assign a channel to VMS OSIT$DEVICE, use the $GETDVI system service to obtain the OS unit number.

6.2.7. Removing an Associated Mailbox

Once a mailbox is associated with an VMS OSIT$DEVICE channel, it remains associated until the VMS OSIT$DEVICE channel is deassigned, using the system service $DASSIGN.

If you do not want to re-use the mailbox, you should deassign the mailbox channel after you have deassigned the VMS OSIT$DEVICE channel.

6.3. Issuing $QIO and $QIOW Calls to OpenVMS OSI transport service

Tasks make requests to OpenVMS OSI transport service by issuing $QIO or $QIOW system service calls on the channel to VMS OSIT$DEVICE.

A $QIO call is used for **asynchronous** requests. It returns control to the task as soon as OpenVMS OSI transport service has validated the request and queued it.

A $QIOW call is used for **synchronous** requests. It only returns control to the task when the service is complete. For example, for an outbound connection request, it will only return control when the request has been accepted or rejected by the responding user.

You might want to use a $QIO call if your application has other processing to do while waiting for the service to complete.

The $QIO(W) calls you can use with OpenVMS OSI transport service are fully described in *Chapter 8, "System Service Calls Using Network Control Blocks"* and *Chapter 9, "System Service Calls Using Item Lists"*. Sections *Section 6.3.1, "Input/Output Status Block (IOSB)"* to *Section 6.3.4, "NCBs"* briefly describe some of the parameters you need to supply.

6.3.1. Input/Output Status Block (IOSB)

You usually specify an address for an input/output status block (IOSB) in a$QIO(W) call. When the call has been processed, the IOSB holds a **completion status code**, which either shows that the call completed successfully, or gives the reason why it failed.

Although the IOSB address is optional, it is strongly recommended that you use it, since it provides a way to assess accurately the success or failure of the $QIO(W) call.

The structure of the IOSB differs slightly depending on the type of call used. See *Appendix C, "Structure of an IOSB"* for details of the structure of IOSBs returned by $QIO(W)calls to OpenVMS OSI transport service.

6.3.2. Item Lists and NCBs

Some $QIO calls to OpenVMS OSI transport service require you to supply information in the form of an **item list** or a **network connect block** (NCB). This is mainly addressing information. You need to supply an item list or NCB whenever you request, accept, reject, or examine a transport service connection.

Item lists are preferable to NCBs because they allow you to:

- Include information needed to negotiate OSI transport protocol options and classes with the remote OSI transport entity.
- Examine the protocol options and classes actually selected for the established transport service connection.

However, if you want your application to be able to run over either DECnet-VAX or OpenVMS OSI transport service, you must use NCBs.

You identify an item list in a $QIO(W) call by giving the address of a descriptor for the item list in *p1*. You identify an NCB in a $QIO(W) call by giving the address of a descriptor for an NCB in *p2*. You can use either *p1* or *p2* as a parameter to a $QIO(W) call, but not both.

6.3.3. Item Lists

Item lists are data structures containing one or more items. There are two kinds of item list:

- An **input item list** contains data you supply to OpenVMS OSI transport service.
- An **output item list** is used by OpenVMS OSI transport service to return data about a transport service connection.

6.3.3.1. Input Item Lists

You use input item lists to provide:

- Addressing information
- The protocol classes and options you wish OpenVMS OSI transport service to negotiate with the remote transport service entity
- Optional user data
- A transport service connection identifier (TC-ID) when dealing with inbound connection requests

You can use input item lists in the following $QIO(W) calls:

$QIO(W)(IO$_ACCESS)	Request outbound connection
$QIO(W)(IO$_SENSEMODE)	Examine inbound connection request
$QIO(W)(IO$_ACCESS)	Accept inbound connection request
$QIO(W)(IO$_ACCESS!IO$M_ABORT)	Reject inbound connection request

The length of the input item list is given by the descriptor in *p1*of the $QIO(W) call. There is no restriction on the length of an input item list.

See *Section E.1.18, "Build Input Item List"* for an example routine to build an input item list.

6.3.3.2. Output Item Lists

In your $QIO call, you can specify a buffer for an output item list in *p3*. OpenVMS OSI transport service uses this buffer to supply information about an inbound connection request or about an established connection. You specify an output item list by giving the address of a descriptor for the buffer to hold the output item list.

You can only specify an output item list if you specify an input item list.

OpenVMS OSI transport service places output items in the output item list buffer until it runs out of space. If you want to be sure of receiving all the output items, you should specify a large buffer. You can specify the maximum size buffer for an output item list by using the literal VMS OSIT $K_MAX_OUTPUT_ITEM_LIST in *p3*. This literal is defined in your library file of OpenVMS OSI transport service-specific symbols.

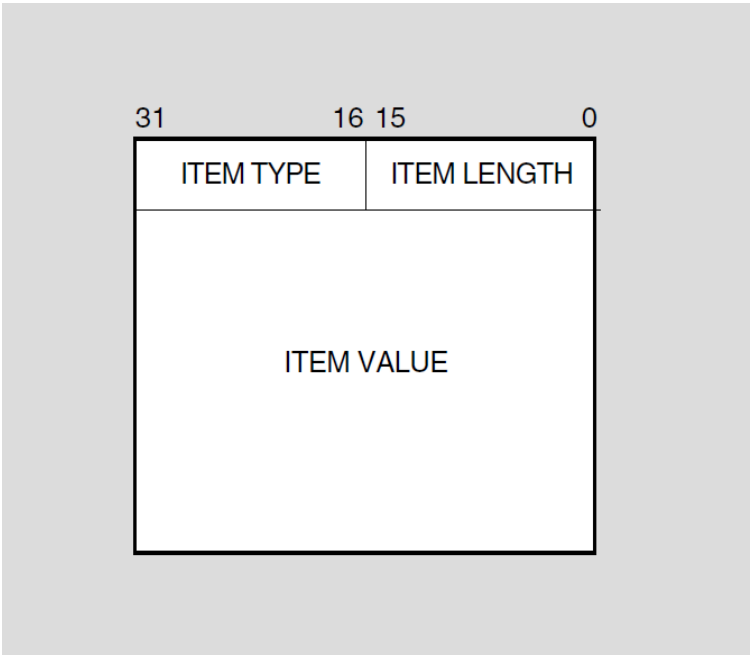
You can specify an output item list in these calls:

$QIO(IO$_ACCESS)	Request outbound connection
$QIO(IO$_SENSEMODE)	Examine inbound connection request
$QIO(IO$_ACCESS)	Accept inbound connection request

6.3.3.3. Structure of an Item in an Item List

Figure 6.1, "Structure of an Item" shows the structure of an item.

Figure 6.1. Structure of an Item



The fields in an item are:

length	The total length of the body of an item in bytes, including the TYPE field and the LENGTH field. Items may be of fixed or variable length.
type	A 16-bit code indicating the type of the item.

value	The actual content of the item. For example, for item type VMS OSIT $K_ITEM_CLASS, the value would be the class of protocol that you would accept for a connection.

6.3.4. NCBs

There are two kinds of NCB (network connect block):

- Outbound NCB. A task uses this to provide addressing information and optional user data. A task can use NCBs in these $QIO(W) calls:

$QIO(W)(IO$_ACCESS)	Request outbound connection
$QIO(W)(IO$_ACCESS)	Accept inbound connection request
$QIO(W)(IO$_ACCESS!IO$M_ABORT)	Reject inbound connection request

- Inbound NCB. OpenVMS OSI transport service supplies this to a task. When an inbound connection request arrives, OpenVMS OSI transport service supplies information about it in the form of an NCB. The task reads the NCB, and uses the information in it when accepting, rejecting, or examining the inbound connection request.

For the structure of an outbound NCB, see *Section 6.4.4, "Supplying an NCB in a Connection Request"*. For the structure of an inbound NCB, see *Section 6.5.3, "Examining the NCB"*.

6.4. Initiating an Outbound Connection

A task initiates an outbound connection by sending a connection request to a responding user on a remote host. The following sections describe how to initiate a connection and examine the response from the responding user.

6.4.1. $QIO and $QIOW Calls for Connection Requests

To request a transport service connection, issue either a $QIO or a $QIOW call with the function code IO$_ACCESS.

The $QIO call returns control to the task as soon as OpenVMS OSI transport service has validated the request and queued it. The $QIOW call only returns control to the task when OpenVMS OSI transport service has sent the connection request to the responding user, and received a connection confirm or reject.

You should specify an address for an IOSB in a connection request. The IOSB will show whether the connection request has been accepted or rejected. If it has been rejected it will show why. The IOSB will also show if the connection request has failed for some other reason.

See *Section E.1.6, "Initiate Outbound Connection Request"* for an example routine to send a connection request. See *Section E.1.20, "Build Input Item List for a Connection Request"* for an example routine to build an item list for a connection request.

6.4.2. Supplying an Input Item List in a Connection Request

Table 6.1, "Item Types Used in a Connection Request" shows the item types you can use in an input item list for a connection request. It also shows which are mandatory (M) and which are optional (O). See *Chapter 9, "System Service Calls Using Item Lists"* for a more detailed explanation of item types.

An explanation of the OpenVMS OSI transport service address (VMS OSIT$K_ITEM_ADDRESS and OSIT$K_ITEM_DESTINATION_NSAP) is in *Section 6.4.5, "Addressing the Remote Host"*. An explanation of TSAPs in outbound connection requests (VMS OSIT$K_ITEM_CALLED_TSAP, VMS OSIT$K_ITEM_CALLING_TSAP) is in *Section 6.4.8, "TSAPs in Outbound Connection Requests"*. An explanation of ISIT$K_ITEM_SEND_IMPLEMENTATION is in *Section 6.4.4, "Supplying an NCB in a Connection Request"*.

See *Section E.1.20, "Build Input Item List for a Connection Request"* for an example routine for building an input item list for a connection request.

Table 6.1. Item Types Used in a Connection Request

Item Type	Meaning	M or O
VMS OSIT$K_ITEM_ADDRESS	An OpenVMS OSI transport service address for the remote host. Note: If this tag is used, the tag OSIT $K_ITEM_DESTINATION_NSAP cannot be present.	M
VMS OSIT $K_ITEM_DESTINATION_NSAP	An OpenVMS OSI transport service address for the remote host. Note: If this tag is used, the tag OSIT $K_ITEM_ADDRESS cannot be present.	M
VMS OSIT$K_ITEM_CALLED_TSAP	The TSAP identifier (TSAP-ID) of the responding user.	O
VMS OSIT$K_ITEM_CALLING_TSAP	The TSAP-ID of the initiating user.	O
VMS OSIT$K_ITEM_CHECKSUM	Indicates whether or not checksums will be included in data units sent over this transport connection.	O
VMS OSIT$K_ITEM_CLASS	The classes of transport protocol to be allowed for this transport connection.	O
VMS OSIT$K_ITEM_EXPEDITED	Indicates whether expedited data is to be available on this transport connection.	O
VMS OSIT$K_ITEM_EXTENDED	Indicates whether extended format should be used to build OSI transport protocol data units (TPDUs) on this transport service connection.	O
VMS OSIT $K_ITEM_SEND_IMPLEMENTATION	Indicates whether the OpenVMS implementation ID for OSI transport should be sent in the outgoing connect.	O
VMS OSIT$K_ITEM_OPTIONS	Protocol options allowed on this transport connection. The options are expedited, extendedm, checksum, and send implementation.	O
VMS OSIT $K_ITEM_PROTOCOL_TYPE	Always VMS OSIT$K_VMS OSI_PROTOCOL.	M
VMS OSIT$K_ITEM_SECURITY	Access control information used to start up responding user. If the remote host is an OpenVMS OSI transport service host, this is a user name and password.	O

Item Type	Meaning	M or O
VMS OSIT$K_ITEM_USER_DATA	Optional data you send, for example, to provide useful information for the responding user.	O

6.4.3. Supplying an Output Item List Buffer in a Connection Request

If you want an output item list, specify the address of a descriptor for the buffer to hold it, in *p3* of the connection request. OpenVMS OSI transport service will return the actual transport characteristics negotiated for this transport connection in the output item list.

See also *Section 6.3.3.2, "Output Item Lists"*, *Chapter 9, "System Service Calls Using Item Lists"*, and *Chapter 10, "Negotiating Protocol Classes and Options"*.

6.4.4. Supplying an NCB in a Connection Request

You can use an NCB instead of an item list in a connection request. If your application is going to use DECnet-Plus for OpenVMS as well as OpenVMS OSI transport service, you must use an NCB.

The structure of an NCB varies slightly according to the information it holds and the language in which it is written. The general form of an NCB is as follows:

```
HOST"ACC-INFO"::"APPL-ID/WORD-ZERO USER-DATA"
```

The fields are:

HOST	This is the address of a remote host. This is either in the form of an OpenVMS OSI transport service address (see *Section 6.4.5, "Addressing the Remote Host"*) or a logical name for an OpenVMS OSI transport service address.
ACC-INFO	This is optional access control information. If the remote host is an OpenVMS OSI transport service host, this is an OpenVMS user name and password.
::	This is a delimiter between the details relating to the host and those relating to the task.
TASK-ID	This identifies the responding user on the remote host. You supply a TSAP identifier (TSAP-ID). See *Section 6.4.8, "TSAPs in Outbound Connection Requests"* for more information.
WORD-ZERO	This is one word that you set to zero for outbound connection requests and OSI transport service sets for inbound connection requests. If you are going to specify USER-DATA, you need to define WORD-ZERO. You do this by using a slash character (/) to denote the start of the word. If you are not going to specify USER-DATA, do not define this word; finish the TSAP-ID with a single set of double quotes (").

USER-DATA	This is a byte-counted string. Specify the first byte of the string as the byte count. You may include up to 32 bytes of user data in any form.

You can omit parts of the NCB if they are not relevant to the connection request.

The following is an example of an NCB written in VAX C:

```
char ncb_buffer [ ] ="LOCNET%AA0047040058::\"TSAP=SERVER_24/\0\0\7USER_AB
\"";
```

6.4.5. Addressing the Remote Host

You must supply an address to identify your remote host, either in the input item list or the NCB. This address must be in the form of a **OpenVMS OSI transport service address**.

An OpenVMS OSI transport service address has two parts: an OpenVMS OSI transport service template name and a network address. The format of an OpenVMS OSI transport service address is:

template-name%network-address

where:

- *template-name* is the name of the OpenVMS OSI transport service template you wish to use to make the test connection. Each type of network service provided by OpenVMS OSI transport service has an associated OpenVMS OSI transport template, which specifies information that is not supplied in the NCB or input item list. The OpenVMS OSI transport service template specifies which type of network service (CLNS or CONS) is to be used. OpenVMS OSI transport templates are set up by the system manager.
- *network-address* is the network address of the target system.

The form of the *network-address* depends on the type of network service used for the test connection. The OpenVMS OSI transport service template named in the first part of the OpenVMS OSI transport service address specifies the type of network service to be used; the network service may be one of:

- CLNS with Internet/ES–IS (NSAP)
- CLNS with Null Internet
- CONS with DTE
- CONS NSAP

  If *template-name* specifies a CLNS OpenVMS OSI transport service template that uses Internet/ES-IS, *network-address* must be an NSAP address. You may specify the NSAP address in either of the following forms (through the NCB and item list interface and logical table).

- In Phase V NA format; for example:

  ```
  CLNSTEMP%49::00-40:08-00-2B-56-87-01:21
  ```

- In full hexadecimal format; for example:

  ```
  CLNSTEMP%49004008002B56870121
  ```

  These are all examples of an OpenVMS OSI transport service address used by CLNS OpenVMS OSI transport service template CLNSTEMP, where the NSAP address of the target system is

49004008002B56870121. Refer to *VSI DECnet-Plus for OpenVMS Network Management Guide* for more information about NSAPs.

You can use either tag OSIT$K_ITEM_ADDRESS or OSIT$K_ITEM_DESTINATION_NSAP to describe the OpenVMS OSI transport service address.

If *template-name* specifies a CLNS OpenVMS OSI transport service template that uses Null Internet, *network-address* must be a LAN address. You may specify the LAN address in either of the following forms (through the NCB and item list interface and logical table):

Note the LAN address may be specified with or without the hyphens that separate each pair of hex digits in the address:

```
NULLTEMP%AA000400E302
```

or:

```
NULLTEMP%AA-00-04-00-E3-02
```

Both of these OpenVMS OSI transport service addresses are used by CLNS OpenVMS OSI transport service template NULLTEMP, where the LAN address of the target system is AA000400E302.

You can use either tag OSIT$K_ITEM_ADDRESS or OSIT$K_ITEM_DESTINATION_NSAP to describe the OpenVMS OSI transport service address.

If *template-name* specifies a CONS OpenVMS OSI transport service template, the *network-address* must be an X.25 address. You may specify the network-address in either of the following forms (through the NCB interface and through the logical table OSIT$NAMES):

The network-address consists of a DTE address (from 1 to 12 digits), possibly followed by a subaddress (if the target system supports subaddressing). The total X.25 address must not exceed 15 digits. For example, the following is an OpenVMS OSI transport service address used by CONS OpenVMS OSI transport service template CONSTEMP, where the DTE address of the target system is 1287389634:

```
CONSTEMP%1287389634
```

The tag OSIT$K_ITEM_ADDRESS may be used to describe the OpenVMS OSI transport service address when using CONS with a DTE address.

If *template-name* specifies a CONS OpenVMS OSI transport service template, you may specify the network-address in the following forms (through the item list interface). The network-address is an NSAP address. For example:

```
CONSTEMP%12370050521234567821
```

To make any VOTS QIO connection over CONS network service, the OSI transport CONS NSAP attribute has to be set to at least one NSAP. It can be the same as any CLNS NSAP.

When using CONS with an NSAP address, you must use the tag OSIT $K_ITEM_DESTINATION_NSAP to describe the remote host address. This tag is only accessible through the item list interface. The NSAP should be encoded as follows:

- Number of semi-octets in the NSAP (1 byte) hexadecimal.

- NSAP itself up to 32 octets hexadecimal.

In the previous example, the number of semi-octets is 12 hexadecimal and the NSAP is 370050521234567821.

Note

If you only use DTE addressing, you do not have to set the OSI transport CONS NSAP attribute.

6.4.5.1. Changes in DECnet and OSI Programming Interface

The $QIO programming interfaces in DECnet-Plus for OpenVMS have been extended to accept an IP name and IP address. This means that any DECnet or OSI application that assigns a channel to the NET or OS device will now be able operate over an IP only network backbone.

The NA $QIO interface will accept IP names and addresses in the following formats:

- Domain name as defined in BIND (for example, bansha.zko.dec.com)
- Address in dotted decimal quad (for example, 1.2.3.4)
- Address in ASCII HEX (for example, 01020304)

In addition, the OSI applications can also specify an optional template. Templates are defined in the OSI transport template subentity in NCL. The VOTS (OSI) $QIO formats:

- Domain name as defined in BIND (for example, bansha.zko.dec.com or osit $rfc1006%bansha.zko.dec.com)
- Address in dotted decimal quad (for example, 1.2.3.4 or osit$rfc1006%1.2.3.4)
- Address in ASCII HEX (for example, 01020304 or osit$rfc1006%01020304)

6.4.5.2. Changes in OSI Programming Interface

The OSI $QIO programming interface (VOTS) in DECnet-Plus for OpenVMS will determine the network service to use by following the steps below:

- Look for the OSIT$K_ITEM_NETWORK_SERVICE tag in the Connect initiate (IO$_ACCESS) call. If this tag is found its value will determine the network service type. Note that this tag was not used before DECnet/OSI Version 6.0.
- Looks for the presence of a template as described in *Section 7.3, "Return Status Codes"*. If a template is specified by the caller, the network service characteristic for that template will be used.
- Use the network service characteristic specified in the DEFAULT OSI transport template. To see the value of your DEFAULT OSI transport template, issue the command:

```
$ MCR NCL show osi transport template default network service
```

6.4.6. Using Logical Names for OpenVMS OSI Transport Service Addresses

OpenVMS OSI transport service will logically translate OpenVMS OSI transport service addresses. This means that tasks using OpenVMS OSI transport service may supply a logical name for the OpenVMS OSI transport service address.

To translate logical names, OpenVMS OSI transport service searches logical name tables as follows:

1. First, it searches logical name tables following the conventions defined by OpenVMS.
2. Then, it searches the OpenVMS OSI transport service logical name table, VMS OSIT$NAMES.

VSI recommends that you use VMS OSIT$NAMES for OpenVMS OSI transport service logical names that are used systemwide.

The VMS OSIT$NAMES logical name table is created automatically when OpenVMS OSI transport service is started.

6.4.6.1. Adding OpenVMS OSI Transport Service Logical Names to VMS OSIT$NAMES

To add a logical name to VMS OSIT$NAMES, use the DCL command:

```
$ DEFINE/TABLE=OSIT$NAMES logical name OpenVMS OSI transportservice address
```

Once you have defined a logical name, you can use it for any subsequent operations.

A process must have SYSNAM privilege to define a logical name in VMS OSIT$NAMES.

6.4.7. Access Control Information in Outbound Connection Requests

You can include access control information in a connection request. This information can be used by the remote host to control access to the responding user.

For a remote host running OpenVMS OSI transport service, access control information consists of a user name and password. Other remote hosts may require different types of access control information.

You can supply access control information either in an NCB or in an input item list:

- In an NCB, you supply access control information in the field ACC-INFO; see *Section 6.4.4, "Supplying an NCB in a Connection Request"*.
- In an input item list, you supply access control information in the item VMS OSIT $K_ITEM_SECURITY; see *Section 6.4.3, "Supplying an Output Item List Buffer in a Connection Request"*.

6.4.8. TSAPs in Outbound Connection Requests

A transport service access point (TSAP) uniquely identifies a transport user. You can provide a called TSAP identifier (TSAP-ID) in an outbound connection request to identify the responding user on the remote host. You can provide a calling TSAP identifier in an outbound connection request to identify the initiating user on the local host.

A called TSAP is equivalent to a task specification string for a DECnet object.

6.4.8.1. TSAP Identifiers in Input Item Lists

In an input item list, you identify:

- The TSAP-ID of the responding task in the item VMS OSIT$K_ITEM_CALLED_TSAP
- The TSAP-ID of the initiating task in the item VMS OSIT$K_ITEM_CALLING_TSAP

Use either of the following formats for the TSAP-IDs:

- A string of up to 32 ASCII characters.
- A string of an even number of packed hexadecimal digits. This can be up to 64 digits long; this is 32 bytes.

You may use any characters in the TSAP-ID.

6.4.8.2. TSAP Identifiers in NCBs

In an NCB, you identify the TSAP-ID of the responding task in the TASK-ID (you cannot identify the initiating task in an NCB). The format you use depends on whether this task will only run over OpenVMS OSI transport service, or may run over either DECnet or OpenVMS OSI transport service.

- If a task will only run over OpenVMS OSI transport service, the format is:

  TSAP= *TSAP-ID*

  where *TSAP-ID* is either a string of up to 32 ASCII characters, or a string of an even number of packed hexadecimal digits, up to 64 digits long.

- If a task is to run over either DECnet or OpenVMS OSI transport service, the format is either of the following:

  TASK= *taskname*

  0= *taskname*

  where *taskname* is the TSAP-ID. Use a string of up to 16 ASCII characters.

Do not use the following characters when you specify a TSAP-ID in an NCB:

- The slash character (/) in an ASCII TSAP-ID, or the byte “2F” (hexadecimal for the slash character) in a hexadecimal TSAP-ID
- The double quotes character (") in an ASCII TSAP-ID or the byte “22” (hexadecimal for the double quotes character) in a hexadecimal TSAP-ID

This is because these characters are used in NCBs as special delimiters.

6.4.9. Send Implementation ID in Item Lists

The item OSIT$K_ITEM_SEND_IMPLEMENTATION enables a user to influence whether the OSI transport implementation ID will be sent in the OSI transport connect request TPDU. This item has some inherent risks if set to false. OSI transport uses the OSI transport implementation ID to determine if the connection is to another Network Architecture (NA) implementation of OSI transport, and will allow some relaxing of stricter ISO 8073 rules in certain cases.

For example, NA implementations allow zero length data TPDUs in Class 0 connection, and allows more than 32 bytes of connect data in a Connect Request TPDU. If the OSIT $K_ITEM_SEND_IMPLEMENTATION is set to false, the OSI transport does not know this is a NA implementation and will adhere strictly to the ISO 8073 rules for OSI transport. Any violations of this protocol results in a protocol violation and a teardown of the connection. This is important if an application is written to allow the NA additions. If the OSIT$K_ITEM_SEND_IMPLEMENTATION is set to false, the connection does not work.

Note

This parameter is also settable in the OSI transport template. If the application does not specify OSIT $K_ITEM_SEND_IMPLEMENTATION, the template parameter is used. The same situation as described above exists with the template specification of SEND IMPLEMENTATION.

6.4.10. Connection Status

When a task makes a connection request, it needs to be informed whether the remote task accepts or rejects the connection request. It also needs to find out if the connection request has failed for some reason.

A task can get information about the status of its connection request in two ways:

- Reading the IOSB specified in the connection request.
- Reading the mailbox. If you have set up a mailbox, you should set your mailbox read to be activated when a connection response arrives.

A task should always check the contents of the IOSB after a connection request, even if it has a mailbox. This is because the mailbox will not contain a message for certain types of connection request failures.

If the task specified an output item list in the connection request, it canal so find out the actual protocol classes and options negotiated for the connection.

6.4.10.1. Reading the IOSB

To find out whether a connection request was accepted or rejected, a task should read the IOSB associated with the connection request $QIO(IO$_ACCESS) call. See *Section E.1.5, "AST Routine to Check Status of Outbound Connection Request"* for an example AST routine to check the status of an outbound connection request.

If the request was accepted, the completion status code in the IOSB is SS$_NORMAL. If the request was rejected, the status code is SS$_REJECT.

A request may fail for some reason other than because it was rejected, for example, because there is no network connection available. In this case, a failure status code will be returned in R0, in the IOSB, or in both. See *Chapter 8, "System Service Calls Using Network Control Blocks"* and *Chapter 9, "System Service Calls Using Item Lists"* for possible reasons for failure, and associated status codes.

6.4.10.2. Reading the Mailbox

To read the mailbox, issue a $QIO(IO$_READVBLK) call on the mailbox channel; see *Section 6.2.5, "Reading the Mailbox"*. You should always have a read call with an AST routine outstanding for the mailbox.

If the request was accepted, the mailbox will contain the message MSG$_CONFIRM. If the request was rejected, the mailbox will contain MSG$_REJECT. There may also be optional user data in the mailbox, if this was supplied by the responding user.

6.4.10.3. Reading the Output Item List

After you issue a connection request, you may want to find out the actual transport service classes and options negotiated for the connection. You can only do this if you specify both an input item list and a buffer for an output item list in your connection request.

If the responding user accepts the connection request, the output item list will contain the actual protocol classes and options negotiated for the connection. If the responding user rejects the connection request, there will be no output item list. See *Section 6.3.3.2, "Output Item Lists"* and *Chapter 9, "System Service Calls Using Item Lists"* for more information.

Your application will need to include a routine to read and analyze the output item list; OpenVMS OSI transport service does not do this for you. See *Section E.1.21, "Display Output Item List"* for an example routine to display an output item list.

6.5. Inbound Connection Requests

6.5.1. Transport Service Access Points

In order for a transport service user to receive inbound connection requests, it must be associated with a TSAP. A TSAP provides a unique identity for a transport user. In this section, a transport service user is defined as a task using OpenVMS OSI transport service.

There are two types of TSAP association: **active** and **passive**:

- An active association is initiated by an executing task.

  The task issues a $QIO(W) call requesting OpenVMS OSI transport service to associate it with a TSAP, and specifying a TSAP-ID. (This is similar to the declaration of a task as a network object in DECnet.)

  OpenVMS OSI transport service creates an active association between the task(identified by its PID) and the TSAP (identified by its TSAP-ID). When OpenVMS OSI transport service creates an active association, it also creates an OpenVMS OSI transport service application entity so that the active association is visible to network management. OpenVMS OSI transport service deletes this entity when the active association is deleted.

- A passive association is created by the system manager.

  The system manager creates an OpenVMS OSI transport service application entity. Characteristics of this entity specify a TSAP-ID and the name of an image (.EXE) or command (.COM) file. OpenVMS OSI transport service creates a passive association between the file (identified by its name) and the TSAP (identified by its TSAP-ID). The OpenVMS OSI transport service application entity must also specify access control information, in the form of a user name.

An inbound connection request typically contains a TSAP-ID that identifies a local user. When an inbound connection request arrives, OpenVMS OSI transport service matches its TSAP-ID with the TSAP-ID specified in one of the local OpenVMS OSI transport service application entities.

If the TSAP association is active, OpenVMS OSI transport service sends connection request details to the mailbox for the channel used to create the TSAP. If the TSAP association is passive, OpenVMS OSI transport service starts up a process to run the associated image or command file.

If the TSAP-ID in the inbound connection request does not match any of the TSAP-IDs in local OpenVMS OSI transport service application entities, OpenVMS OSI transport service rejects the connection request.

Once a task establishes an active association with a TSAP, all the connection requests for that TSAP use a single process. A task with a passive TSAP association needs a new process for each new connection

request it receives. You can have a task that is both actively and passively associated; see *Section 6.5.1.2, "A Passive TSAP Association that Becomes Active"*.

6.5.1.1. Creating an Active TSAP Association

To create an active TSAP association, a task must have SYSNAM privilege. It must also have assigned an VMS OSIT$DEVICE channel with an associated mailbox.

To request an active TSAP association, a task issues a $QIO(W)(IO$_ACPCONTROL) call on a VMS OSIT$DEVICE channel with an associated mailbox. The call must specify:

- A network function block (NFB) containing a function code for attaching to a TSAP. This code must be one of the following:

NFB$C_FC_ATTACH_TSAP	For applications that run only on OpenVMS OSI transport service
NFB$C_DECLNAME or NFB$C_DECLOBJ	For applications that may run on either DECnet or OpenVMS OSI transport service

- A TSAP-ID. The format is either a string of up to 32 ASCII characters, or a string of an even number of hexadecimal digits up to 64 digits long.

When OpenVMS OSI transport service receives this call, it creates an active association between the task (identified by its PID) and the TSAP (identified by its TSAP-ID). When inbound connection requests arrive for this TSAP, OpenVMS OSI transport service sends details to the mailbox for the VMS OSIT $DEVICE channel used to create the TSAP, that is, the one used for the $QIO(IO$_ACPCONTROL) call.

A task can actively associate itself with a number of TSAPs by:

- Creating several TSAP associations on a single channel.
- Assigning several channels to VMS OSIT$DEVICE, and creating one or more TSAP associations on each.

A task cannot share a TSAP with another task.

6.5.1.2. A Passive TSAP Association that Becomes Active

You may want to create an active TSAP association for a task that already has a passive association. This would allow the newly started task to receive further inbound connection requests.

A task will have both an active and a passive TSAP association if:

- The system manager has created an OpenVMS OSI transport service application entity that specifies the name of an image or command file to activate the task.
- The task contains a $QIO(IO$_ACPCONTROL) call requesting to be associated with a TSAP.
- The TSAP-ID in the $QIO(IO$_ACPCONTROL) call matches the TSAP-ID specified in the OpenVMS OSI transport service application entity.

You need to ensure that the TSAP-ID in the $QIO(IO$_ACPCONTROL) call is exactly the same as the one specified in the OpenVMS OSI transport service application entity.

When the system manager creates an OpenVMS OSI transport service application entity, the TSAP-ID is specified as an octet string. The TSAP-ID of an associated task is passed as an ASCII string. You must make sure that the ASCII string matches the octet string.

6.5.1.3. Deleting an Active TSAP Association

A task that is actively associated with a TSAP can delete that association in any of the following ways:

- The task deassigns the channel that was used to associate with the TSAP, by issuing a $DASSGN call on that channel.
- The task issues a $QIO(IO$_CANCEL) call on the channel used to associate with the TSAP.
- The task exits.

When any of these happen, OpenVMS OSI transport service ends the active association between the TSAP and the task.

6.5.1.4. Passive TSAP Association: Supplying a .COM File

The system manager can create a passive TSAP association between a TSAP and either an image or a command file. It is preferable to use a command file, however, because OpenVMS generates a log file for a command file but not for an image file.

The name of the log file is the same as that of the command file, except that it has the extension .LOG instead of .COM. OpenVMS creates and starts writing to the log file as soon as it has created a process to validate and run the command file.

An example of a .COM file is shown below. This file rejects connection requests from hosts that are not on its list of "friendly nodes."

```
$    node_1 = "X25%74309998125556,"
$    node_2 = "X25%74089786756662,"
$    node_3 = "IEEE%DD98789CC012,"
$    node_4 = "IEEE%BEADD8466331"
$    node_list = node_1+node_2+node_3+node_4
$ !
$    address_length = f$locate("::",f$trn("sys$net"))
$    remote_address = f$extract(0,address_length,f$trn("sys$net"))
$    num = 0
$loop:
$       friendly_node = f$element(num,",",node_list)
$       if friendly_node .eqs. "," then goto intruder_alert
$       if remote_address .eqs. friendly_node then goto run_responder
$       num = num + 1
$       goto loop
$ !
$run_responder:
$    write sys$output "Connection request received from friendly node
$    ''remote_address'"
$ !   run sys$sysdevice:[user]tc_responder
$    goto logout
$ !$intruder_alert:
$    write sys$output "Intruder attempting access from ''remote_address'"
$logout:
$ !    logout/full
```

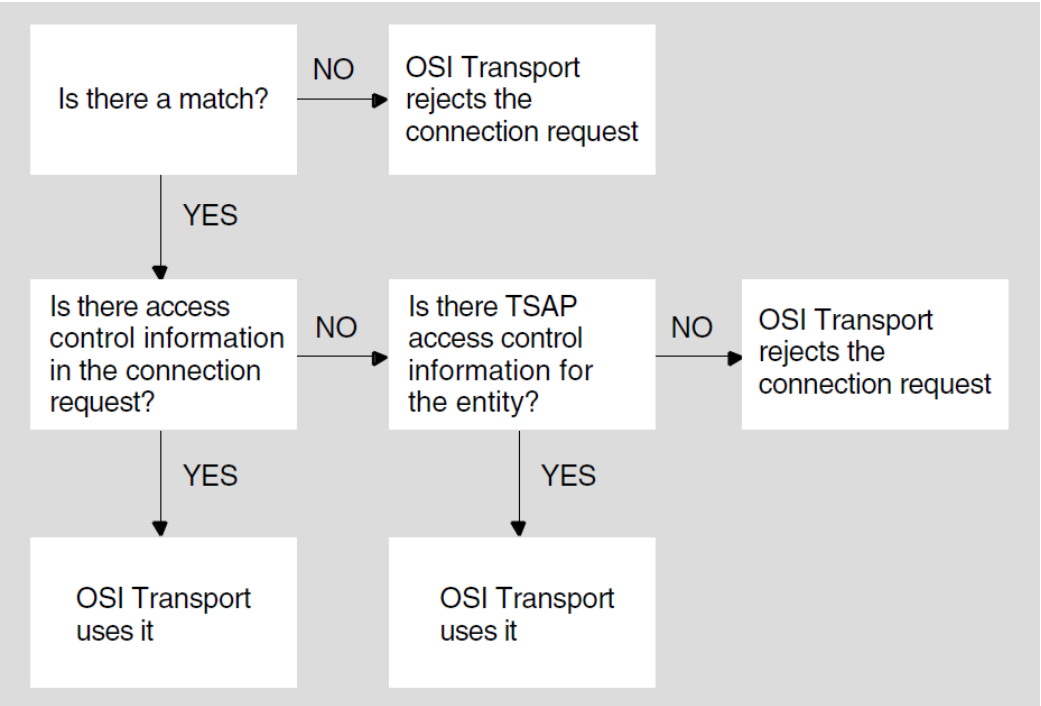
6.5.1.5. Passive TSAP Association: Access Control Information

OpenVMS OSI transport service checks access control information in inbound connection requests for passively associated TSAPs. This information is in the form of an OpenVMS user name and password.

The system manager can specify access control information in an OpenVMS OSI transport service application entity. If access control information is taken from the OpenVMS OSI transport service application entity, there is no password;the user name is used with proxy access.

When a connection request arrives, OpenVMS OSI transport service looks for an OpenVMS OSI transport service application entity with a matching TSAP-ID. Then OpenVMS OSI transport service takes the actions shown in *Figure 6.2, "Handling Inbound Connection Requests"*:

Figure 6.2. Handling Inbound Connection Requests



6.5.2. Reading Inbound Connection Requests

When a connection request arrives and OpenVMS OSI transport service matches a TSAP with the TSAP-ID in the connection request, what happens next depends on whether the TSAP has an active or passive association:

- If the association is active, OpenVMS OSI transport service sends details of the connection request to the mailbox for the channel on which the TSAP was associated.

  For example, if a task issues a $QIO(IO$_ACPCONTROL) call on channel 23 to associate with a TSAP whose TSAP-ID is "FRED", then subsequent inbound connection requests for the TSAP with TSAP-ID "FRED" will be sent to the mailbox associated with channel 23.

  Connection request details are in the form of an NCB.

  A task with an active TSAP association should include a $QIO(W)(IO$_READVBLK) call to read its mailbox, which is activated when a connection request arrives.

- If the association is passive, OpenVMS OSI transport service creates a process to run LOGINOUT. LOGINOUT validates the access control information and invokes DCL to execute the .EXE or .COM file.

  Once the file is an executing task, it can read the connection request details in the NCB. OpenVMS OSI transport service equates the NCB to the logical name SYS$NET. To retrieve the NCB, the task must translate SYS$NET. See *Section E.1.2, "Translation of SYS$NET"* for an example routine to translate SYS$NET.

A task may examine the NCB to decide whether it will accept or reject the connection request. The task may also get additional information about the connection request by issuing a $QIO(W)(IO $_SENSEMODE) call.

6.5.3. Examining the NCB

The format of the NCB constructed by OpenVMS OSI transport service is as follows:

```
host::"TSAP=calling-tsap-id tc-id  user-data CALLED
task-id
```

The fields are:

host	This is the address of the initiating user; that is, the remote host sending the connection request. This is in the form of an OpenVMS OSI transport service address; see *Section 6.4.5, "Addressing the Remote Host"*. The template name in the OpenVMS OSI transport service address is the name of the inbound OpenVMS OSI transport service template that was selected when the connection request was received.
::	This is a delimiter between the details relating to the host and those relating to the task.
TSAP=	This is a string included in the NCB by OpenVMS OSI transport service. It is followed by the TSAP-ID of the initiating user; that is, the calling TSAP.
tc-id	This is a unique identifier assigned by OpenVMS OSI transport service to this transport service connection. If you supply this NCB when you accept or reject the connection request, make sure you do not overwrite the TC-ID, because OpenVMS OSI transport service needs it to route the response.
user-data	This is an optional string. It may contain information to identify the responding user.
task-id	This is the TSAP-ID of the responding user.

A task typically uses this inbound NCB when accepting, rejecting, or examining the inbound connection request. See *Section 6.5.5.1, "Accepting a Connection Request"* for more details.

6.5.4. Examining the Connection Request Using$QIO(IO $_SENSEMODE)

The NCB returns only a limited amount of information about an inbound connection request. To get more detail, you can issue a $QIO(IO$_SENSEMODE) call. Typically, you would issue this call to examine the transport service protocol classes and options in the connection request.

When you issue $QIO(IO$_SENSEMODE), you must specify:

- An input item list

- A buffer to hold an output item list

6.5.4.1. Input Item List for $QIO(IO$_SENSEMODE)

The only mandatory information in the input item list is the transport service connection identifier (TC-ID). This identifies the transport service connection used by the inbound connection request.

The tc-id is in the NCB constructed by OpenVMS OSI transport service for the inbound connection request. The task must analyze the NCB to extract the tc-id. The easiest way to do this is by using the library routine LIB$PARSE_NCB. This routine is provided with OpenVMS OSI transport service; see *Appendix D, "LIB$PARSE_NCB"* for a description.

In the input item list, supply the TC-ID in the item OSIT$K_ITEM_TC_ID.

6.5.4.2. Output Item List for $QIO(IO$_SENSEMODE)

You specify the address of the buffer to hold the output item list in *p3*. The output item list returned gives the transport protocol options and classes being requested by the remote user. Also included are extended format, expedited data, checksums, and send implementation. You need to provide a routine to read the output item list; see *Section E.1.21, "Display Output Item List"* and *Section E.1.22, "Displays a Specified Item"* for example routines.

Once the task has examined the output item list, it can decide whether to accept or reject it. For example, it can make acceptance conditional on receiving specific options and classes from the remote host.

See *Section 6.3.3.2, "Output Item Lists"* and *Chapter 9, "System Service Calls Using Item Lists"* for more details.

6.5.5. Accepting or Rejecting a Connection Request

Once the task has examined the connection request, it must accept or reject it.

If the task is passively associated with a TSAP, it will only have been started when the connection request arrived. Therefore, it needs to issue an $ASSIGN call to assign a channel to VMS OSIT $DEVICE before it can accept or reject the connection request.

6.5.5.1. Accepting a Connection Request

To accept a connection request, a task issues a $QIO(W)(IO$_ACCESS) call. This call establishes the connection. For Class 2 or Class 4 transport connections, the accept call may contain up to 32 bytes of user data.

Note

The Network Architecture (NA) implementation of OSI transport allows more than 32 bytes of connect data. See Section 6.7 for more information on NA implementations.

If the task issues a $QIO call, control will return to the task when the accept request is queued. If it issues a $QIOW call, control will return to the task when the connection is established.

In Phase IV, a program can accept an incoming request to establish a link with a no-wait $QIO (FUNC=IO$_ACCESS) call followed immediately with a read or write operation on the channel. This behavior is not preserved with DECnet-Plus. In DECnet-Plus, your application must wait for the connect accept to complete before attempting to use the connection.

When your accept call is processed and the connection is established, the IOSB will hold the completion status code SS$_NORMAL. If the accept call fails, the IOSB and the mailbox will hold an error status code. See *Chapter 8, "System Service Calls Using Network Control Blocks"* and *Chapter 9, "System Service Calls Using Item Lists"* for lists of status codes.

See *Section E.1.17, "Accept an Inbound Connection"* for an example routine to accept an inbound connection. See *Section E.1.16, "Check Acceptance of Inbound Connection"* for an example AST routine to check the acceptance of an inbound connection.

A task must supply an NCB or an input item list with an accept call.

Supplying an NCB with an Accept Call

If a task supplies an NCB, it is easiest to use the inbound NCB returned by OpenVMS OSI transport service (either in the mailbox, or from the translation of SYS$NET).

If you wish to supply new user data, you need to examine the inbound NCB and then modify the user data field.

Supplying an Item List with an Accept Call

There are two methods for supplying an input item list:

- If you issued a $QIO(IO$SENSEMODE) call when you received the connection request, you can use the output item list for that call as the input item list for your accept call. You may wish to modify the class and options items.
- You can supply a new input item list. This must include the TC-ID that identifies the connection used by the inbound connection request.

  Use the item VMS OSIT$K_ITEM_TC_ID for the TC-ID.

  The TC-ID is in the NCB constructed by OpenVMS OSI transport service for the inbound connection request. The task must analyze the NCB to extract the TC-ID. The easiest way to do this is by using the library routine LIB$PARSE_NCB. See *Appendix D, "LIB$PARSE_NCB"* for a description of this routine.

  You may also supply your preferred protocol classes, extended format, expedited data, checksums, send implementations, and options parameters in the accept call.

If the preferred protocol classes, extended format, expedited data, checksums, send implementation, or options (which includes the previous four items) are not supplied with the accept call, they will be supplied by the inbound OpenVMS OSI transport service template selected when the incoming connection request was received. See your *VSI DECnet-Plus for OpenVMS Network Management Guide* manual for details of how the inbound OpenVMS OSI transport service template is selected.

You can specify a buffer for an output item list with the accept call. The output item list will contain the actual transport protocol class and options negotiated between OpenVMS OSI transport service and the remote transport service.

6.5.5.2. Rejecting a Connection Request

To reject the connection request, a task issues a $QIO(W)(IO$_ACCESS) call with the modifier IO $M_ABORT. For Class 2 or Class 4 transport connections, this call may contain up to 64 bytes of user data.

You must supply an input item list or an NCB with a reject call. You do this in the same way as for an accept call; see *Section 6.5.5.1, "Accepting a Connection Request"*. However, the only items of information OpenVMS OSI transport service uses for reject calls are the TC-ID and user data (if any).

When the reject call is processed, the IOSB will hold the status code SS$_NORMAL.

6.5.5.3. Using Different Channels for Receiving and Accepting

Inbound connection requests for tasks actively associated with a TSAP arrive on the channel used to associate with the TSAP. It is advisable for a task actively associated with a TSAP to accept a connection request on a different channel from the one on which it arrived. If the task is using several connections, this will enable it to keep the inbound requests separate from the established connections.

For example, once a given number of connections are established, you may wish to prevent further inbound connection requests for a TSAP. You can do this by issuing a $CANCEL or $DASSGN call on the channel used to associate with the TSAP. Provided that this channel has not been used for an accept call, this will not destroy existing connections.

6.6. Exchanging Data

When the transport connection is established, the communicating tasks can begin to exchange data. Either task may send or receive data.

To send data on an OpenVMS OSI transport service system, a task issues a $QIO(W) write request. To receive data, a task issues a $QIO(W) read request.

If OSI transport times out during data transfer mode because the remote end is unreachable, the error SS$_CONNECFAIL may also be returned in the IOSB for the $QIO(IO$_READVBLK) or $QIO(IO $_WRITEVBLK) call. As a workaround, user code should be made to handle SS$_TIMEOUT, as well as SS$_CONNECFAIL.

You use write and read requests to exchange:

- Normal or expedited data

  Normal data is used for most data exchange. Expedited data bypasses normal flow control, and is used for small control or emergency messages.

- Fragmented data messages

  When a task makes a read or write request, it must specify a buffer of a given size to hold the data. The size of the buffer is limited by the maximum buffer size allowed on the system.

  OpenVMS OSI transport service allows a task to send and receive messages that are larger than the maximum. To do this, a task indicates that a single read or write request will be fragmented over multiple $QIO(W) calls. The task indicates this by adding a modifier to the read or write request.

6.6.1. Exchanging Normal Data with No Fragmentation

To send unfragmented normal data, a task issues a $QIO(IO$_WRITEVBLK) call with no modifier. To receive such data, a task issues a $QIO(IO$_READVBLK) call with no modifier.

If a task does not specify fragmented messages, it must agree with the other transport task on the buffer size they are going to use. If the buffer size is not agreed on, data may be lost. For example, if an

inbound message is too large for the buffer size specified in the read request, the part of the message that does not fit the buffer is lost. OpenVMS OSI transport service returns the failure status code SS $_DATAOVERUN.

The communicating tasks need to define a method for agreeing on a buffer size.

6.6.2. Exchanging Expedited Data

When requesting expedited data, the OSI transport service user should make sure there is a mailbox associated with the channel to VMS OSIT$DEVICE. OpenVMS OSI transport service does not check to see if there is a mailbox associated with the channel to VMS OSIT$DEVICE. If there is no associated mailbox, the OSI transport service user will only be able to send expedited data; it will not be able to receive the expedited data. If expedited data is received and there is no mailbox associated with the channel to VMSOSIT$DEVICE, OpenVMS OSI transport service will assume implicit success. This could lead to potential problems for the sender of the expedited data, including connections that hang or break.

Expedited data has priority over normal data. It bypasses the flow control mechanisms provided by OpenVMS OSI transport service, such as credit allocation. Expedited data is used for signaling and interrupt purposes, typically when a task needs to communicate some urgent information. Note that expedited data is not necessarily sent faster than normal data.

A task may send up to 16 bytes of expedited data at a time.

A task can only send or receive expedited data if all the following are true:

- A mailbox is associated with the VMS OSIT$DEVICE channel used for the required connection.
- The task is using a Class 2 or Class 4 transport connection.
- The task has negotiated the use of expedited data.

  A task can indicate whether it wants expedited data by specifying a value for VMS OSIT $K_EXPEDITED or VMS OSIT$K_OPTIONS when it issues a connection request. OpenVMS OSI transport service will try to negotiate whatever has been specified.

  If the task does not request use of expedited data and does not specifically request non-use of expedited data, it should use a transport service template that has expedited data characteristic set to false. See *Chapter 10, "Negotiating Protocol Classes and Options"* for more about negotiation.

To send expedited data, issue a $QIO(IO$_WRITEVBLK) call with the modifier IO$M_INTERRUPT.

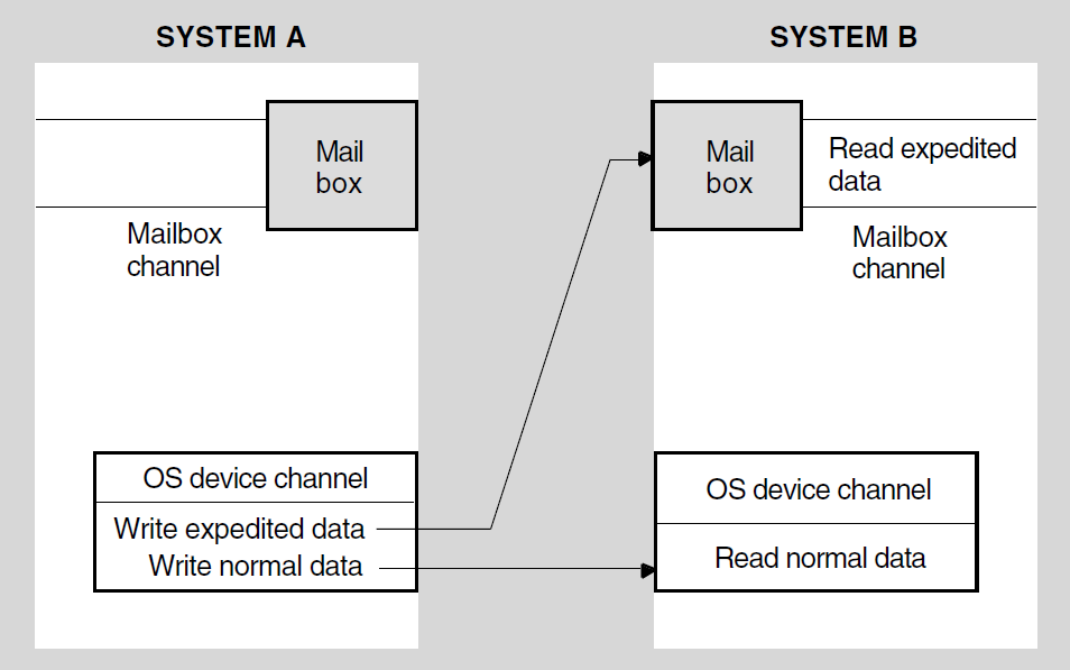
When OpenVMS OSI transport service receives expedited data, it places it in the associated mailbox. The mailbox message type for expedited data is MSG$_INTMSG. The INFO field of the message contains the expedited data.

If there is no mailbox associated with the channel to VMS OSIT$DEVICE and OpenVMS OSI transport service receives expedited data, the expedited data cannot be delivered. OpenVMS OSI transport service will assume implicit success. Transport service users should be careful when requesting use of expedited data with no mailbox.

To receive expedited data, issue a $QIO(IO$_READVBLK) call on the mailbox. If the task is to receive expedited data, it should always have a read request outstanding on the mailbox. Otherwise, the expedited data units may not be received in the correct order.

Figure 6.3, "Expedited and Normal Data" illustrates how OpenVMS OSI transport service handles expedited and normal data.

Figure 6.3. Expedited and Normal Data



6.6.3. Correct Sequence for Expedited and Normal Data

The ISO transport protocol requires that expedited data be delivered before normal data that is sent after it. OpenVMS OSI transport service guarantees to deliver expedited and normal data in the correct sequence, as required by ISO. However, it is possible for a task to receive them out of order. This is because a task receives normal and expedited data in different ways:

- It receives normal data by making QIO(IO$_READVBLK) calls on the VMS OSIT$DEVICE channel.
- It receives expedited data by issuing QIO(IO$_READVBLK) calls on the mailbox associated with the VMS OSIT$DEVICE channel.

If the task does not read all the mailbox messages as soon as OpenVMS OSI transport service delivers them, it may not receive expedited data before normal data sent after it.

To ensure that it receives all expedited data before any normal data sent after it, the task should empty the mailbox each time it is read. To do this, VSI recommends the following:

- The task always has an outstanding asynchronous read request to the mailbox; that is, a $QIO(IO $_READVBLK) call.
- The AST routine named in this $QIO(IO$_READVBLK) request contains synchronous read requests to read the mailbox immediately; that is, $QIOW calls with the function code (IO $_READVBLK!IO$M_NOW).

The $QIO(IO$_READVBLK) call to the mailbox should be activated whenever a message is delivered to the mailbox. The AST routine should then make repeated $QIOW(IO$_READVBLK!IO$M_NOW)

calls until all the outstanding messages in the mailbox have been read. When the mailbox is empty, the IO$M_NOW modifier will cause the read request to complete with a failure status of SS$_ENDOFFILE in the first longword of the IOSB associated with the call.

See *Section E.1.26, "Wait for Mailbox Message and Read Mailbox"* for an example AST routine to empty the mailbox.

6.6.4. Fragmented Data Messages

A task must specify the size of a buffer to hold data when it makes a read or write request. The size of buffer it is allowed to specify is limited by the maximum buffer size allowed on the OpenVMS system. This maximum is determined by the value of the SYSGEN parameter, MAXBUF. It is also restricted by the maximum buffer size allowed by the $QIO system service, which is 64K.

In order for a task to exchange data messages larger than the maximum, it must fragment the relevant read and write requests. It does this by adding the modifier IO$M_MULTIPLE to the read and write requests.

6.6.4.1. Fragmented Read Requests

A task issues the call $QIO(IO$_READVBLK!IO$M_MULTIPLE) to indicate that it will supply another receive buffer if the inbound data message is larger than the buffer specified in the call.

If an inbound data message is larger than the buffer specified, OpenVMS OSI transport service places the status code SS$_BUFFEROVF in the IOSB for the call. This is not an error status. It simply means that there is more data to come. The next read will receive the next fragment of the data message. If the next message fits into the buffer, OpenVMS OSI transport service returns the status SS$_NORMAL. This indicates that you have received the last fragment of the message.

You are advised always to supply IO$M_MULTIPLE with read requests. If you do not, and the receive buffer is too small for the incoming message, the read request will return the status code SS$_DATAOVERUN. This is an error status. It means that some data has been lost and cannot be recovered.

6.6.4.2. Fragmented Write Requests

The call $QIO(WRITEVBLK!IO$M_MULTIPLE) indicates that the message is not complete and the task will supply more data in the next write request. To send the final fragment of data, the task issues the call $QIO(WRITEVBLK) with no modifier.

6.6.5. How OpenVMS OSI Transport Service Handles Write Requests

OpenVMS OSI transport service handles a write request in two phases:

1. **Accepting or rejecting the request**

   OpenVMS OSI transport service checks the request to see if it can be accepted for processing, for example, that it has no invalid parameters.

   If OpenVMS OSI transport service accepts the request, it passes it to OSI transport where it is queued until resources become available to process it. If the write request was a $QIO call, control

is returned to the task, with SS$_NORMAL in R0. If the write request was a $QIOW call, control is not returned until the request is processed (or aborted).

If OpenVMS OSI transport service refuses the request, it is aborted, and R0 returns an error status. Control returns to the task.

2. **Processing the request**

   If the request is accepted, OSI transport service will process it when sufficient resources become available. The data to be transmitted is copied into system buffers in nonpaged pool and the request given to OpenVMS.

   When processing has finished successfully, one of the following happens:

   - If the task used a $QIO call, the AST routine provided called, or the event flag set. The first word of the IOSB contains SS$_NORMAL.
   - If the task used a $QIOW call, control is returned to the task. R0 and the first word of the IOSB contain SS$_NORMAL.

Because OpenVMS OSI transport service has successfully completed a write request does not mean that the data has been received by the responding user. It just means that the transmit buffer may be reused. The task will know that the data was correctly received only when it receives an acknowledgment from the responding user.

If the transport connection is disconnected after the write request is accepted but before it is processed, it will be aborted. The IOSB will contain an error status code.

6.6.6. Example Routines for Exchanging Data

See *Section E.1.15, "Read Data"* for an example routine to read data. See *Section E.1.14, "Disconnect After Read Is Complete"* for an example AST routine called after the read request is complete.

See *Section E.1.13, "Send Data on the Transport Connection"* for an example routine to send normal data. See *Section E.1.12, "Free Write Buffer When Write Request Completes"* for an example AST routine to free the write buffer.

6.7. Canceling Input/Output on a Channel

A task can cancel outstanding I/O requests at any time during the lifetime of a transport connection. To do this, it issues the system service call $CANCEL on the channel being used. The cancel request does the following:

- Immediately cancels outstanding read and write I/O requests on the channel.
- Deletes any active TSAP associations created on the channel.
- Disconnects the connection on that channel.

6.8. Disconnecting a Transport Connection

Either of the transport users exchanging data may disconnect at any time. Usually, the task that receives the final data unit disconnects first. For a Class 0 connection, a transport disconnection call also disconnects the network connection.

6.8.1. Initiating a Disconnection

A task can end a transport connection by issuing a $QIO(W)(IO$_DEACCESS) call with the modifier IO$M_ABORT. It issues this call on the channel used for the connection.

For Class 2 or Class 4 transport connections, the DEACCESS call may contain up to 64 bytes of user data. However, it is not guaranteed that this data will reach the receiving user.

When a task issues a DEACCESS call, it has the following effects:

- If there are any $QIO(W) requests outstanding when the DEACCESS call is issued, they will complete, but with a failure status. SS$_ABORT will be returned in the first word of the IOSB.
- If the task issues any other $QIO(W) call after issuing a DEACCESS call, it will get a failure or error status in R0.
- Data still in transit when the DEACCESS call is issued may be lost. To prevent this happening, use a higher level protocol to disconnect synchronously.
- The task may use the channel for another transport connection, without having to assign the channel to VMS OSIT$DEVICE again.

Issuing a DEACCESS call does not affect any active associations with TSAPs made on the channel.

See *Section E.1.11, "Disconnect Current Transport Connection"* for an example routine to disconnect a current connection. See *Section E.1.10, "Check Status of Disconnection"* for an example AST routine to check the status of the disconnection call.

6.8.2. Receiving a Disconnection Request

If the remote user disconnects the transport connection, OpenVMS OSI transport service places the message MSG$_ABORT in the mailbox. The mailbox message may also contain user data supplied in the disconnection request.

When the task using OpenVMS OSI transport service receives this mailbox message, it should issue a $QIO (IO$_DEACCESS!IO$M_ABORT) call. This is necessary to make the OpenVMS OSI transport service channel ready for reuse.

If the task does not have a mailbox, it will not be directly informed of the disconnection.

When a task receives a disconnection request, it has the following effects:

- If there are any $QIO(W) requests outstanding when the disconnection request is received, they will complete, but with a failure status. The following will be in the IOSB:
  - SS$_code in the first word.
  - The OpenVMS OSI disconnect reason code in the second word.
  - The OpenVMS OSI transport service-specific reason code in the second longword. This always begins with VMS OSIT$.
- If the task issues further $QIO(W) requests to OpenVMS OSI transport service after receiving the disconnection, it will get a failure or error status in R0. This is normally SS$FILNOTACC.

- Data still in transit when the disconnect call is issued may be lost. To prevent this happening, you can use a higher level protocol to disconnect synchronously.

6.9. Deassigning the Channel

If the task has no further use for the OpenVMS OSI transport service channel, it should issue a $DASSGN call. This call:

- Immediately ends all communication
- Releases the channel for reuse
- Dissociates the mailbox from the channel
- Deletes all active TSAP associations created on the channel

A $DASSGN call has the effect of disconnecting a transport connection. For a Class 0 connection, the $DASSGN call also disconnects the network connection.

Section E.1.9, "Deassign a Channel" contains an example of deassigning a channel.

Chapter 7. Calling the System Services

The programming languages that generate VAX native-mode instructions provide mechanisms for coding procedure calls. When coding a system service call, always supply the arguments that the service requires. To call system service procedures, use the VAX calling conventions.

When a system service completes, it returns control to the calling program with a status code. Always analyze this status code to determine the success or failure of the service call, so the program can alter the flow of execution, if necessary.

If you are a VAX MACRO programmer, read *Section 7.1, "MACRO Coding"*. If you program in any other language, read *Section 7.2, "High-Level Language Coding"* for general information on how to call system services. For detailed information and examples, see the user guide for your language. See the directory SYS$EXAMPLES for example programs.

7.1. MACRO Coding

System service macros generate argument lists and CALL instructions to call system services. These macros are located in the system library SYS$LIBRARY:STARLET.MLB. This library is searched automatically for unresolved references when you assemble a source program.

You will need to make the VMS OSI transport service-specific symbols available in your MACRO program. These symbols are contained in the SYS$LIBRARY:OSIT.MAR file. Always include this file in any MACRO application programs that you write.

You will need to know the MACRO rules for assembly-language coding to follow this section. The *VAX MACRO and Instruction Set Reference Manual* contains the necessary information.

7.1.1. Argument Lists

Chapter 8, "System Service Calls Using Network Control Blocks" and *Chapter 9, "System Service Calls Using Item Lists"* show descriptions of the system services and the arguments needed for a system service call. The MACRO format for each system service shows the keyword names for, and the requirements of, each argument.

All arguments are longwords. The first longword in the list must contain the number of arguments in the remainder of the list in its low-order byte. The remaining three bytes must be zeros.

If you omit an optional argument in a system service macro instruction, the macro supplies a default value for the argument.

There are two generic macro forms for coding calls to system services:

```
$ name_G
$ name_S
```

The form of the macro to use depends on how you construct the argument list for the system service:

- The $ **name**_G form requires you to construct an argument list elsewhere in the program, and specify the address of this list as an argument to the system service. (OpenVMS provides a macro to

create an argument list for each system service.) With this form, you can use the same argument list with modifications, if necessary, for repeated calls to the macro.

- The $ **name**_S form requires you to supply the arguments to the system service in the macro instruction. The macro generates code that associates the argument list to the executing program. With this form, you can use registers to contain or point to arguments, so you can write re-entrant programs.

The $ **name**_G macro form generates a CALLG instruction; the$ **name**_S macro form generates a CALLS instruction. The services are called according to the standard conventions. System services save all registers except R0 and R1, and restore the saved registers before returning control to the calling program.

The *VSI OpenVMS System Services Reference Manual* describes how to code system service calls using each of these macro forms.

7.2. High-Level Language Coding

You will need to make the OpenVMS OSI transport service-specific symbols available in your high-level language program. The OpenVMS OSI transport service-specific symbols are contained in library files with the name SYS$LIBRARY:VMS OSIT. *x*, where *x* identifies a programming language. Always include the appropriate SYS$LIBRARY:VMS OSIT.*x* file in any application programs that you write.

Each high-level language supported by OpenVMS provides a mechanism for calling an external procedure and passing arguments to that procedure. This mechanism and the terminology used, however, varies from one language to another.

There are three ways to pass arguments in a system service call:

- By value

  The argument is the actual value (a number or a symbolic representation of a numeric value).

- By reference

  The argument is the address of an area or field that contains the value. A reference is usually expressed as a label associated with an area or field. A common error is to pass a numeric value without indicating that it is an actual value. If the compiler assumes the numeric value is an address, a run-time access violation error may occur when, for example, the image tries to access the address.

- Using a descriptor

  This argument is also an address, but of a special data structure called a character-string descriptor. *Section 7.2.1, "Descriptors"* shows the format of a descriptor.

The high-level language format for each system service is:

SYS$ASSIGN *(devnam,chan,[acmode],[mbxnam])*

SYS$DASSGN *(chan)*

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

See *Chapter 8, "System Service Calls Using Network Control Blocks"* and *Chapter 9, "System Service Calls Using Item Lists"* for a description of each of the arguments. The description of each service indicates how each argument is to be passed.

Terms such as “address” and “address of a character string descriptor,” identify arguments that are references. Words like “indicator”, “number”, “value”, or “mask” indicate arguments that are actual values.

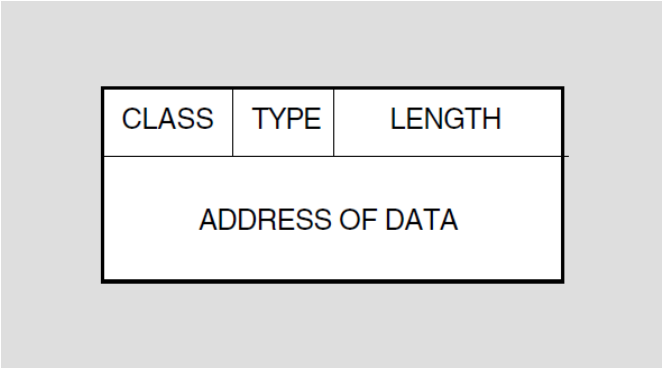
7.2.1. Descriptors

A character-string descriptor is a quadword (8-byte) area that contains the length of the string data and the starting address of the data. In most cases, the compiler automatically generates the descriptor and the data; in some cases, you may need to define all the fields yourself. (See the appropriate user guide for the language you use.)

Descriptors are explained fully in the VAX Procedure Calling and Condition Handling Standard; see the *VAX Architecture Handbook* and the *VSI OpenVMS Utility Routines Manual*.

The format of a descriptor is shown in *Figure 7.1, "Format of a Descriptor"*.

Figure 7.1. Format of a Descriptor



The fields in a descriptor are:

length	Specifies the number of ASCII characters for the data, or the number of bytes in the buffer; this value is placed in the low-order word of the longword. In some cases, you may want to move a value into this field during program execution.
type	Specifies the data type of the argument. This byte is ignored by system services.
class	Specifies the class of descriptor. This byte is ignored by system services; therefore, dynamic string descriptors are treated as fixed-length string descriptors.
address of data	Shows the starting address of the data. You may have to specify the reference name or label associated with the data.

Data is input to or output from the system service. If the descriptor is for output from the service, allocate enough bytes to hold the data returned by the system service. The data is not part of the descriptor.

7.3. Return Status Codes

When a system service returns control to your program, it returns status information in the form of a status code. The $ASSIGN, $DASSGN, $QIO and $QIOW system services place the return status code in the general register R0. In addition, $QIO or $QIOW calls will return a completion status code in the

input/output status block (IOSB), provided that you have specified a descriptor for an IOSB address in the $QIO(W) call.

After each call to a system service, you should check the return and completion status codes to find out whether the call completed successfully. You can also test for specific error conditions.

The operating system does not automatically handle system service failure or warning conditions; you must test for them, and handle them yourself. However, you can interrupt the program or override the default handling, by declaring a condition handler (see the *VSI OpenVMS System Services Reference Manual*).

7.3.1. Format of the Return Status

In R0, the return status is stored as a binary value in a longword. You can test just the low-order bit, the three low-order bits, or the entire value:

- The low-order bit indicates successful (1) or unsuccessful (0) completion of the service.
- The three low-order bits, taken together, represent the severity of the error. Severity code values are:

Value	**Severity Level**
0	Warning
1	Success
2	Error
3	Informational
4	Severe (or fatal) error
5-7	(Reserved)

- The remaining bits (3 through 31) classify the particular return condition and the operating system component that issued the status code. For system service return status values, the high-order word (bits 16 through 31) contains zeros.

Each numeric status code has a symbolic name in the following format where *code* is a mnemonic code describing the return condition:

SS$_*code*

For example, the most common successful return is indicated by SS$_NORMAL. A common error status code is SS$_ACCVIO (access violation), indicating that the service could not read an input argument or write an output argument.

The symbols associated with the different return status values are defined in the default system library.

7.3.2. Information Provided by Status Codes

Status codes usually show if the service completed successfully, although sometimes they simply provide information to the calling program. Moreover, a success code merely indicates that the service has completed all its functions and returned control to the calling program. For example, the status code SS$_BUFFEROVF, indicating that a character-string is longer than a designated buffer, is a success code.

Warning status codes and some error status codes show that the service may not have done everything it should.

For $QIO(W) calls, a status code is returned in the IOSB, as well as in R0. The status code in R0 and the status code in the IOSB provide information about different aspects of the $QIO(W) call. The return status code in R0 gives information about the success or failure of the call, rather than the operation itself. The completion status code in the IOSB gives information about the success or failure of the service operation. To assess accurately the success or failure of the $QIO(W) call, you must check the status codes in both R0 and the IOSB.

For example, a $QIO(IO$_READVBLK) call might place return status SS$_NORMAL in R0, yet fail because the transport service connection breaks, generating the completion status code SS $_LINKABORT in the IOSB.

See *Appendix C, "Structure of an IOSB"* for the structure of the IOSBs that OpenVMS OSI transport service uses for $QIO(W) calls.

The service descriptions in *Chapter 8, "System Service Calls Using Network Control Blocks"* and *Chapter 9, "System Service Calls Using Item Lists"* include the status codes that OpenVMS OSI transport service uses for each call, and where they are placed. *Appendix A, "$QIO(W) Status Codes and OSI Reason Codes"* contains a summary table of status codes used by OpenVMS OSI transport service.

7.3.3. Testing the Status Code

Each language provides some mechanism for testing status codes. Often you only need to check the low-order bit, for example, a test for true (success status) or false (error or warning status).

However, you can check for a specific status. Each language provides a way for your program to determine the values associated with specific, symbolically-defined codes. Always use these symbolic names when your program tests for specific conditions.

For information on how to test for these codes, see the user guide for your programming language.

7.4. Obtaining Values for Other Symbolic Codes

In addition to the symbolic codes for specific return conditions, many individual services also have symbolic codes for offsets, identifiers, or flags associated with these services. For example, the Create Process ($CREPRC) service, which is used to create a subprocess or a detached process, has symbolic codes associated with the various privileges and quotas you can grant to the created process.

If the language you are using has a method of obtaining values for these symbols, refer to the user guide for a description of the method. If the language does not have such a method, do the following:

- Write a short VAX MACRO program containing the desired macros.
- Assemble the program and generate a listing. Use the listing to find the desired symbols and their hexadecimal values.
- Define each symbol and its value within your source program.

7.5. Special Return Conditions

When an error occurs during a system service, two process modes affect the way the calling program regains control. These modes are:

- Resource wait mode
- System service failure exception mode

If you choose to change the default setting for either of these modes, your program must handle the special conditions that result.

7.5.1. Resource Wait Mode

Many system services require certain system resources for execution. These resources include system dynamic memory and process quotas for I/O operations. Usually, when a system service requires a resource that is not available, the program has to wait until the resource becomes available. Only then can the service continue executing. This mode is called resource wait mode.

However, it may not be practical or desirable for a program to wait. In these cases, you can choose to disable resource wait mode, so that when such a condition occurs, control returns immediately to the calling program with an error status code. You can enable or disable resource wait mode with the Set Resource Wait Mode ($SETRWM) system service.

How a program responds to the lack of a resource depends on the application, and the particular service that is being called. In some instances, the program may want to continue execution and retry the service call later. In other instances, it may be necessary just to wait until the resource is free.

7.5.2. System Service Failure Exception Mode

This mode determines whether control is returned to the caller in the normal manner following an error in a system service call, or whether an exception is generated. System service failure exception mode is disabled by default; the calling program regains control following the error. You can enable and disable system service failure exception mode with the Set System Service Failure Exception Mode ($SETSFM) service.

High-level language compilers generate calls to system services for many statements or instructions in source programs. For example, reads and writes to files generate calls to VAX RMS, which uses the $QIO and $QIOW services. If you enable system service failure exception mode, many different types of errors (such as an I/O attempt to a nonexistent device or non-numeric input to a mathematics routine) will generate the message:

```
%SYSTEM-F-SSFAIL, system service failure exception,...
```

Because of this, VSI recommends that you do not use system service failure exception mode in high-level language programs, except perhaps when debugging.

Chapter 8. System Service Calls Using Network Control Blocks

This chapter describes the OpenVMS system service calls you can use to communicate with the OpenVMS OSI transport service.

The calls are in MACRO format. Each high-level language supported by OpenVMS has its own mechanism for calling and passing arguments to the OpenVMS system services. If you are using a high-level language, see the language's user's guide for specific information. See the *VSI OpenVMS System Services Reference Manual* for information about the OpenVMS system services.

Some of the $QIO(W) system service calls described in this chapter are also described in *Chapter 9, "System Service Calls Using Item Lists"*. This chapter discusses the format of these calls when they supply network connect blocks (NCBs). *Section 6.4.4, "Supplying an NCB in a Connection Request"* describes the contents of an NCB. *Chapter 9, "System Service Calls Using Item Lists"* gives the format of the calls when they supply item lists.

8.1. Summary of Call Description

Each system service description contains the following sections. Some services require additional information; this is described in the notes for the service.

Format

Shows the macro name, with all keyword arguments listed in order.

Arguments

Describes the arguments.

Notes

Gives any additional information.

Status Codes

Lists the status codes returned by the service that are significant for OpenVMS OSI transport service, and explains what the codes mean. The $QIO(W) system service calls have two lists of status codes:

- Status codes returned in R0
- Status codes returned in the IOSB

8.1.1. Argument List

In the descriptions of $QIO(W) calls, only those arguments that are unique to the service are described in the Arguments section. This section lists the arguments that are common to all $QIO(W) calls. These arguments always have the values shown here.

For example, for a $QIO call with the format:

$QIO *efn,chan,func,iosb,astadr,astprm,p1,p2,p3,p4,p5,p6*

the relevant section of the system service description describes only arguments *func, p1* and *p2*.

The other arguments have the following values:

efn	The number of the event flag set when a request finishes. If not specified, the default is 0.
chan	Number of the channel assigned to a device: • For calls to OpenVMS OSI transport service, the device is OpenVMS OSIT $DEVICE • For calls to the mailbox, the device is the mailbox Use the channel number obtained through $ASSIGN or through LIB $ASN_WTH_MBX.
iosb	Address of a quadword I/O status block that will receive the completion status.
astadr	Entry point address of an AST routine that executes when the I/O completes. If specified, the AST routine executes at the access mode specified previously in $ASSIGN.
astprm	AST parameter to be passed to the AST completion routine.

8.1.2. Syntax of Calls

The following conventions are used to describe the syntax of the system service calls (the conventions are part of the MACRO language).

- A character is one of the set of alphanumerics, including:
  - A through Z
  - a through z
  - 0 through 9
  - _ (underscore)
  - $ (dollar sign)
- All calls are in uppercase letters, and you must enter these as shown. Arguments are in *italics*, and you must replace the argument in the call format with the precise information required.
- Square brackets [] enclose optional keywords and arguments. Do not include the brackets when entering the call.
- You must enter punctuation such as commas and parentheses () as shown in the call format. Use consecutive commas to indicate omitted arguments; you can omit commas that indicate optional arguments at the end of a call format.

8.2. Assign a Channel

$ASSIGN

You use $ASSIGN to assign a channel to the OpenVMS OSI transport service pseudodevice, OpenVMS OSIT$DEVICE. This enables your task to communicate with OpenVMS OSI transport service. You

must use the channel number of a OpenVMS OSIT$DEVICE channel in any $QIO(W) calls your task issues to the OpenVMS OSI transport service.

If you want to associate a mailbox with the channel to OpenVMS OSIT$DEVICE, use the run-time library routine LIB$ASN_WTH_MBX instead of $ASSIGN. This routine assigns a channel to a device, creates a mailbox, and associates the mailbox with the channel.

Format:

$ASSIGN *devnam,chan,[acmode],[mbxnam]*

Arguments:

devnam	Address of a quadword character string descriptor pointing to the device name string. To assign a channel to the OpenVMS OSI transport service, the character string must contain OpenVMS OSIT$DEVICE or a logical name for VMS OSIT$DEVICE.
chan	Address of a word to receive the channel number assigned.
acmode	Access mode to be associated with the channel. The specified access mode must be an access mode less privileged than, or equal in privilege to, the access mode from which the service was called. The channel allows I/O operations only from equally privileged, or more privileged, access modes. Passed by value.
mbxnam	Address of a quadword character string descriptor pointing to the device name of the mailbox to be associated with the channel. An address of 0 (zero) implies no mailbox; this is the default value. This mailbox remains associated with the channel until you deassign the channel.

Notes:

1. When you assign a channel to OpenVMS OSIT$DEVICE, the OpenVMS OSI transport service creates a new pseudodevice called OS *n*, where *n* is a unique unit number. The channel then belongs to OS *n*. (You may use $GETDVI to discover the actual unit number allocated.)

   Never explicitly assign a channel to an OS *n* device. Always assign the channel to OpenVMS OSIT $DEVICE.

2. Assign only one channel for each transport service connection.

Status Codes in R0:

SS$_ACCVIO	The task issuing the call does not have read access to the device, mailbox name string, string descriptor, buffer, or IOSB; or the task issuing the call does not have write access to the channel number, buffer, or IOSB.
SS_$_ILLSER	An illegal system service was called.
SS_$_INSFARG	Not enough arguments were supplied in the call.

SS$_INSFMEM	There is insufficient system dynamic memory (nonpaged pool) to allow a channel to be assigned.
SS$_IVCHAN	An invalid channel number was supplied.
SS$_NOPRIV	The task issuing the call does not have the privileges required to assign the channel.
SS$_NORMAL	The service has successfully completed. OpenVMS has assigned a channel.
SS$_NOSUCHDEV	The OpenVMS OSI transport service has not been loaded on the local host.

8.3. Canceling Read and Write Requests on a Channel

$CANCEL

You use the $CANCEL system service to cancel pending I/O requests on a specific channel. The OpenVMS OSI transport service cancels read and write I/O requests immediately.

Issuing a $CANCEL call also does the following:

- Deletes any active TSAP associations created on the channel
- Disconnects any connection on that channel

Format:

$CANCEL *chan*

Argument:

chan	Number of the I/O channel on which I/O is to be canceled. Passed by value.

Notes:

1. You can cancel I/O only from an access mode equal to, or more privileged than, the access mode from which you originally assigned the channel.
2. When a request currently in progress is canceled, the OpenVMS OSI transport service is notified immediately. The action taken by the OpenVMS OSI transport service is similar to that taken for queued requests:
   - The specified event flag is set.
   - The OpenVMS OSI transport service places the status code SS$_ABORT in the first word of the IOSB if the request is queued or is in progress.
   - The AST, if specified, is queued.

Outstanding I/O requests are canceled automatically when a task exits.

Status Codes in R0:

SS$_ACCVIO	The task issuing the call does not have write access to the channel number.
SS$_EXQUOTA	The task issuing the call does not have sufficient buffered I/O quota, and it has a disabled resource-wait mode.
SS_$_ILLSER	An illegal system service was called.
SS_$_INSFARG	Not enough arguments were supplied in the call.
SS$_IVCHAN	An invalid channel number was supplied.
SS$_NOPRIV	Either the specified channel has not been assigned, or it was assigned from a more privileged access mode than is being used for the cancel call.
SS$_NORMAL	The service has successfully completed. All I/O has been canceled on the specified channel.

8.4. Deassign the Channel

$DASSGN

The $DASSGN system service is used to free an I/O channel once a transport service connection has concluded. A $DASSGN issued on a channel associated with an active transport service connection:

- Ends pending operations to send or receive data
- Deletes any active TSAP associations created on the channel
- Deletes the channel associated with the connection
- Ends all communication on that channel

Format:

$DASSGN *chan*

Argument:

chan	Number of the channel to be deassigned. Passed by value.

Status Codes in R0:

SS$_IVCHAN	The task issuing the call specified an invalid channel number.
SS$_NOPRIV	Either the channel specified in the call is not assigned, or it was assigned from a more privileged access mode.
SS$_NORMAL	Service successfully completed. Channel has been deassigned.

8.5. Request a Transport Service Connection

$QIO(W)(IO$_ACCESS)

You use the $QIO(W) system service with a function code of IO$_ACCESS to request an outbound transport service connection. For Class 2 and Class 4 transport service connections, you may send up to 32 bytes of user data in this call; the user data forms the final part of the NCB.

Format:

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

Arguments:

func	IO$_ACCESS
p1	Not used
p2	Address of quadword descriptor of the NCB
p3 - p6	Not used

Status Codes in R0:

SS$_ACCVIO	The task issuing the call has no read access to the device, mailbox name, string descriptor, buffer or IOSB; or the task issuing the call has no write access to the channel number, buffer or IOSB.
SS$_BADPARAM	One or more of the parameters *p1* to *p6* is not valid for this call. For example, an invalid NCB, or an invalid combination of parameters.
SS$_DEVOFFLINE	Either the OSI transport service has not yet been loaded or it has been unloaded.
SS$_EXQUOTA	Either the task issuing the call has not enough FILCNT to allow another connection, or not enough BYTCNT for OpenVMS to allocate enough system resources to establish the connection.
SS$_FILALRACC	A connection already exists on the specified channel.
SS$_ILLEFC	The call supplied an illegal event flag number.
SS$_ILLIOFUNC	The call supplied a function code that is unknown to the OpenVMS OSI transport service.
SS_$_ILLSER	An illegal system service was called.
SS_$_INSFARG	Not enough arguments were supplied in the call.
SS$_INSFMEM	There is not enough system dynamic memory (nonpaged pool) for the connection to be established.
SS$_IVCHAN	An invalid channel number was supplied.

SS$_NOPRIV	The transport service user does not have the NETMBX privilege.
SS$_NORMAL	The OSI transport service has accepted and queued the connection request.

Status Codes in the IOSB:

SS$_ABORT	The task issued a $CANCEL or $QIO(IO $_DEACCESS) call before the connection request was processed.
SS$_CONNECFAIL	The connection request failed.
SS$_INSFMEM	There is not enough system dynamic memory (nonpaged pool) for the connection to be established.
SS$_NOLINKS	The maximum number of concurrent transport service connections has been reached, as defined by the MAXIMUM TRANSPORT CONNECTIONS characteristic of the OSI transport service entity.
SS$_NORMAL	The remote host has accepted the connection request. The connection is established.
SS$_NOSUCHNODE	The transport service template specified in the OpenVMS OSI transport service-address cannot be found.
SS$_NOSUCHOBJ	The specified TSAP-identifier is unknown at the remote host.
SS$_PATHLOST	The remote host failed to reply within the required time.
SS$_PROTOCOL	One of these OSI transport protocol errors has occurred: ● The local user failed to request Class 0 when required. ● In response to the connection request, the OSI transport service has received an invalid TPDU; for example, a checksum failure. ● Inbound connection confirm requested an invalid class. ● Inbound connection confirm requested an invalid protocol option. ● Format of the connection confirm is incorrect. For example, the connection confirm specifies Class 0, but also specifies a protocol option that is invalid for Class 0. ● Invalid TPDU size in connection confirm.

	• The remote host has rejected the request with one of the following OSI reason codes (hexadecimal): 83, 84, 85, 88, 90. See *Appendix A, "$QIO(W) Status Codes and OSI Reason Codes"* for the meanings of these codes.
SS$_REJECT	The remote user has rejected the connection request, and supplied one of the following OSI reason codes(hexadecimal): 80, 82. See *Appendix A, "$QIO(W) Status Codes and OSI Reason Codes"* for the meanings of these codes.
SS$_REMSRC	The remote host has not enough system resources to process the connection request.
SS$_SHUT	The system manager has disabled the OpenVMS OSI transport service entity.
SS$_THIRDPARTY	The system manager has disabled the OSI transport service entity.
SS$_TIMEOUT	The connection has failed because it timed out.
SS$_UNREACHABLE	The OSI transport service could not establish a network connection, for one of these reasons: • The remote host has rejected the connection request. • The DTE address was incorrect or unknown. • The maximum number of network connections has been reached, as specified by the MAXIMUM NETWORK CONNECTIONS characteristic of the OSI transport service entity.

8.6. Accept a Request to Set Up a Transport Service Connection

$QIO(W)(IO$_ACCESS)

You use the $QIO system service with a function code of IO$_ACCESS to accept an inbound connection request. For Class 2 and Class 4 transport service connections, you may send up to 32 bytes of user data in this call; the user data forms the final part of the NCB.

Format:

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

Arguments:

func	IO$_ACCESS

p1	Not used
p2	Address of quadword descriptor of the NCB
p3 - p6	Not used

Note:

The OpenVMS OSI transport service sends an NCB to your task to inform it of the inbound connection request. You are advised to use the NCB sent by the OpenVMS OSI transport service as the NCB in argument *p2*.

Make sure that the TC-ID you use is the same as that in the field WORD-ZERO after the slash (/) delimiting character. If the NCB for the connection request has any user data, you may want to remove it, and substitute your own user data.

Status Codes in R0:

SS$_ACCVIO	The task issuing the call has no read access to the device, mailbox name, string descriptor, buffer or IOSB; or the task issuing the call has no write access to the channel number, buffer or IOSB.
SS$_BADPARAM	One or more of the parameters *p1* to *p6* is not valid for this call. For example, an invalid NCB, an invalid combination of parameters.
SS$_DEVOFFLINE	Either the OSI transport service has not yet been loaded or it has been unloaded.
SS$_EXQUOTA	Either the task issuing the call has not enough FILCNT to allow another connection, or not enough BYTCNT for OpenVMS to allocate enough system resources to establish the connection.
SS$_FILALRACC	A connection already exists on the specified channel.
SS$_ILLEFC	The call supplied an illegal event flag number.
SS$_ILLIOFUNC	The call supplied a function code that is unknown to the OpenVMS OSI transport service.
SS_$_ILLSER	An illegal system service was called.
SS_$_INSFARG	Not enough arguments were supplied in the call.
SS$_INSFMEM	There is insufficient system dynamic memory (nonpaged pool) to allow the transport connection to be accepted.
SS$_IVCHAN	An invalid channel number was supplied.
SS$_NOPRIV	The transport service user does not have the NETMBX privilege.
SS$_NORMAL	The OSI transport service has accepted and queued the connection accept.

Status Codes in the IOSB:

SS$_ABORT	The local transport service user issued a $CANCEL or $QIO(IO$_DEACCESS) call before the accept call was processed.
SS$_FILNOTACC	There is no connection associated with the channel in the accept call. Either the connection was disconnected before the OSI transport service could process the call or the task specified a nonexistent TC-ID.
SS$_INSFMEM	There is not enough system dynamic memory (nonpaged pool) available to complete the connection request.
SS$_LINKABORT	The remote transport service has sent a disconnection request (DR) TPDU, with an OSI reason code indicating an error. See *Appendix A, "$QIO(W) Status Codes and OSI Reason Codes"* for a description of OSI reason codes.
SS$_NOPRIV	The TC-ID supplied with the accept call identifies a connection belonging to another user.
SS$_NORMAL	The transport service connection is established.
SS$_PATHLOST	The remote host failed to acknowledge the accept call within the required time.
SS$_PROTOCOL	There has been one of the following transport protocol errors: ● The local user supplied an invalid protocol class in the accept call. Either the class is unsupported by the OSI transport service, or it is not one of those requested by the remote transport service. ● The OSI transport service received an error TPDU from the remote host.
SS$_REJECT	The remote user has rejected the connection confirm, and supplied one of the following OSI disconnect reason codes (hexadecimal): 80,82. See *Appendix A, "$QIO(W) Status Codes and OSI Reason Codes"* for the meanings of these codes.
SS$_REMSRC	Class 4 only. The remote host has insufficient system resources to process the connection confirm.
SS$_SHUT	The system manager has disabled the OSI transport service entity.
SS$_THIRDPARTY	The system manager has disabled the OSI transport service entity.
SS$_TIMEOUT	The connection confirm has failed because it timed out.

8.7. Reject a Request to Set Up a Transport Service Connection

$QIO(W)(IO$_ACCESS!IO$M_ABORT)

You use the $QIO system service with a function code of IO$_ACCESS and modifier IO$M_ABORT to reject an inbound connection request. On Class 2 and Class 4 transport service connections, you may send up to 64 bytes of user data; the user data forms the final part of the NCB.

Format:

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

Arguments:

func	IO$_ACCESS!IO$M_ABORT
p1	Not used
p2	Address of quadword descriptor of the NCB
p3 - p6	Not used

Note:

The OpenVMS OSI transport service sends an NCB to your task to inform it of an inbound connection request. You are advised to use the NCB sent by the OpenVMS OSI transport service as the NCB in argument *p2*.

Make sure that the TC-ID you use is the same as that in WORD-ZERO after the slash (/) delimiting character.

Status Codes in R0:

SS$_ACCVIO	The task issuing the call has no read access to the device, mailbox name, string descriptor, buffer or IOSB; or the task issuing the call has no write access to the channel number, buffer or IOSB.
SS$_BADPARAM	One or more of the parameters *p1* to *p6* is not valid for this call. For example, an invalid NCB, an invalid combination of parameters.
SS$_DEVOFFLINE	Either the OSI transport service has not yet been loaded or it has been unloaded.
SS$_EXQUOTA	The task issuing the reject call does not have enough BYTCNT for OpenVMS to allocate enough system resources to reject the connection.
SS$_FILALRACC	A connection has already been established on the specified channel.
SS$_ILLEFC	The call supplied an illegal event flag number.
SS$_ILLIOFUNC	The call supplied a function code that is unknown to the OpenVMS OSI transport service.

SS_$_ILLSER	An illegal system service was called.
SS_$_INSFARG	Not enough arguments were supplied in the call.
SS$_INSFMEM	There is insufficient system dynamic memory (nonpaged pool) for the OpenVMS OSI transport service to process the reject call.
SS$_IVCHAN	An invalid channel number was supplied.
SS$_NORMAL	OSI transport service has accepted and queued the connection rejection.

Status Codes in the IOSB:

SS$_FILNOTACC	There is no connection associated with the channel in the reject call. Either the connection was disconnected before the OSI transport service could process the call or the task specified a nonexistent TC-ID.
SS$_NOPRIV	The TC-ID supplied with the reject request identifies a connection belonging to another user.
SS$_NORMAL	The inbound connection request has been rejected and the transport service connection is disconnected.
SS$_THIRDPARTY	The system manager has disabled the OSI transport service entity.

8.8. Associate a Task with a TSAP

$QIO(W)(IO$_ACPCONTROL)

You use the $QIO system service with a function code of IO$_ACPCONTROL to actively associate a task with a TSAP. You can associate a task with more than one TSAP.

A task actively associated with a TSAP can handle more than one inbound connection request at a time.

Before you issue this call, you must associate a mailbox with the OpenVMS OSIT$DEVICE channel used. This is because the OpenVMS OSI transport service uses the mailbox to deliver inbound connection requests to a task actively associated with a TSAP. See *Section 6.2, "Assigning a Channel and Setting Up a Mailbox"* for more details.

A task requires SYSNAM privilege to associate itself with a TSAP.

Format:

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

Arguments:

func	IO$_ACPCONTROL
p1	Address of quadword descriptor of an NFB. This is a 5-byte block consisting of a function code (one byte) and a longword parameter. The format of the 5-byte block is:

	.BYTE function code .LONG 0
p2	Address of a quadword descriptor of the TSAP-ID.
p3 - p6	Not used

Note:

For the NFB function code in *p1*, use one of the following:

- NFB$C_FC_ATTACH_TSAP, if the task will only use the OpenVMS OSI transport service
- NFB$C_DECLNAM or NFB$C_DECLOBJ, if the task will use DECnet as well as the OpenVMS OSI transport service

Status Codes in R0:

SS$_ACCVIO	The task issuing the call has no read access to the device, mailbox name, string descriptor, buffer or IOSB; or the task issuing the call has no write access to the channel number, buffer or IOSB.
SS$_BADPARAM	One or more of the parameters *p1* to *p6* is not valid for this $QIO call. For example, an invalid NCB, an extra parameter, an invalid combination of parameters.
SS$_DEVOFFLINE	Either the OSI transport service has not yet been loaded, or it has been unloaded.
SS$_ILLEFC	The call supplied an illegal event flag number.
SS$_ILLIOFUNC	The call supplied a function code that is unknown to the OpenVMS OSI transport service.
SS_$_ILLSER	An illegal system service was called.
SS_$_INSFARG	Not enough arguments were supplied in the call.
SS$_INSFMEM	There is insufficient system dynamic memory (nonpaged pool) to complete the request.
SS$_IVCHAN	An invalid channel number was supplied.
SS$_NOMBX	The task attempting to associate with a TSAP is using a channel without an associated mailbox. This is not allowed.
SS$_NOPRIV	The task does not have SYSNAM privilege. This is required for a task to associate with a TSAP.
SS$_NORMAL	The OSI transport service has accepted and queued the request to associate with a TSAP.

Status Codes in the IOSB:

SS$_NORMAL	The OSI transport service has established an active association between the task issuing the call and the specified TSAP.
SS$_WRITLCK	There is already an active TSAP association with the specified TSAP-ID. This code is returned

	only if a task previously created an active TSAP association with the specified TSAP-ID.

8.9. Receive Data

$QIO(W)(IO$_READVBLK)

You use the $QIO system service with a function code of IO$_READVBLK to receive data from a remote user over the transport service connection. To receive normal data, issue the read call on the channel to OpenVMS OSIT$DEVICE. To receive expedited data, issue it on the mailbox channel.

You can receive data messages larger than the buffer size specified in *p2* by using the modifier IO $M_MULTIPLE. The call $QIO(IO$_READVBLK!IO$M_MULTIPLE) indicates that your task will supply another receive buffer if the inbound data message is larger than the buffer specified. See *Section 6.6.4, "Fragmented Data Messages"* for details.

Format:

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

Arguments:

func	IO$_READVBLK or IO$_READVBLK!IO$M_MULTIPLE
p1	Address of buffer
p2	Buffer length in bytes. The maximum number of bytes is defined by the SYSGEN parameter MAXBUF. Passed by value.
p3 - p6	Not used

Note:

The second word of the IOSB contains a count of the number of bytes of data that have been read.

Status Codes in R0:

SS$_ACCVIO	The task issuing the call has no read access to the device, mailbox name, string descriptor, buffer or IOSB; or the task issuing the call has no write access to the channel number, buffer or IOSB.
SS$_DEVOFFLINE	Either the OSI transport service has not yet been loaded, or it has been unloaded.
SS$_FILNOTACC	There is no connection associated with the specified channel.
SS$_ILLEFC	The call supplied an illegal event flag number.
SS$_ILLIOFUNC	The call supplied a function code that is unknown to the OpenVMS OSI transport service.
SS_$_ILLSER	An illegal system service was called.
SS_$_INSFARG	Not enough arguments were supplied in the call.
SS$_IVCHAN	An invalid channel number was supplied.

SS$_NORMAL	The OSI transport service has accepted and queued the write request.

Status Codes in the IOSB:

SS$_ABORT	The local transport service user issued a $CANCEL or $QIO(IO$_DEACCESS) call before the read request was processed.
SS$_BUFFEROVF	This may be returned if the read call has the modifier IO$M_MULTIPLE. It means that the inbound message is larger than the buffer specified. This is not an error message; it indicates that there is more data to come. The task should continue to issue reads with IO$M_MULTIPLE until the OpenVMS OSI transport service returns the status SS$_NORMAL to indicate the end of the message.
SS$_DATAOVERUN	This may be returned if the read call does not use the modifier IO$M_MULTIPLE. It means that the inbound message is larger than the buffer specified, and some data has been lost. This is an error message.
SS$_FILNOTACC	There is no connection associated with the channel supplied in the read call. Either the connection was disconnected before the OpenVMS OSI transport service could process the call or the task specified a nonexistent TC-ID.
SS$_LINKABORT	The remote transport service has sent a disconnection request (DR) TPDU, with an OpenVMS OSI reason code indicating an error. See *Appendix A, "$QIO(W) Status Codes and OSI Reason Codes"* for a list of OSI reason codes.
SS$_LINKDISCON	The remote transport service has sent a DR TPDU with the OSI reason code 80. This is a normal disconnect.
SS$_NORMAL	The OSI transport service has successfully processed the read request.
SS$_PATHLOST	Applies only to X.25 networks. The network connection has been disconnected or reset.
SS$_PROTOCOL	There has been a transport service protocol error. One of the following has happened: ● The OSI transport service has received an error (ER) TPDU from the remote host. ● The OSI transport service has received an incorrect or invalid TPDU from the remote host.
SS$_SHUT	The system manager has disabled the OSI transport service entity.

SS$_THIRDPARTY	The system manager has disabled the OSI transport service entity.
SS$_TIMEOUT	For Class 4 only. The read request has failed because the retransmission limit has been reached or because the inactivity timer has expired.

8.10. Synchronously Disconnecting a Transport Service Connection

$QIO(W)(IO$_DEACCESS!IO$M_SYNCH)

You use the $QIO system service with a function code of IO$_DEACCESS and modifier IO $M_SYNCH to end a transport connection. It does the following:

- Ends all pending operations to send or receive data
- All pending transmit messages are sent to the remote node before the link is disconnected
- Ends a transport connection
- Frees the channel associated with the connection for further connections.

For Class 2 and Class 4 transport connections, you may send up to 64 bytes of user data with this call.

The transport protocol does not guarantee delivery of the user data in the call itself.

Format:

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

Arguments:

func	IO$_DEACCESS!IO$M_SYNCH
p1	Not used
p2	Address of quadword descriptor of a byte counted string of user data.
p3 - p6	Not used

Status Codes in R0:

SS$_ACCVIO	The task issuing the call has no read access to the device, mailbox name, string descriptor, buffer or IOSB; or the task issuing the call has no write access to the channel number, buffer or IOSB.
SS$_BADPARAM	One or more of the parameters *p1* to *p6* is not valid for this call. For example, an invalid NCB, an invalid combination of parameters.
SS$_DEVOFFLINE	Either OSI transport service has not yet been loaded or it has been unloaded.

SS$_EXQUOTA	The process issuing the request does not have enough BYTCNT to allow OpenVMS to allocate sufficient system resources to end the connection.
SS$_FILNOTACC	There is no transport connection associated with the channel supplied in the disconnect request.
SS$_ILLEFC	The call supplied an illegal event flag number.
SS$_ILLIOFUNC	The call supplied a function code that is unknown to the OpenVMS OSI transport service.
SS_$_ILLSER	An illegal system service was called.
SS_$_INSFARG	Not enough arguments were supplied in the call.
SS$_INSFMEM	There is insufficient system dynamic memory (nonpaged pool) to allow the transport connection to be accepted.
SS$_IVCHAN	An invalid channel number was supplied.
SS$_NORMAL	OSI transport service has accepted and queued the disconnect request.

Status Codes in the IOSB:

SS$_FILNOTACC	There is no connection associated with the channel supplied in the disconnect call. The connection may have been disconnected before the OpenVMS OSI transport service could process the call, for example, by the remote user; alternatively, the task specified a nonexistent TC-ID.
SS$_NOPRIV	The TC-ID supplied with the disconnect request identifies a connection belonging to another user.
SS$_NORMAL	The transport connection is disconnected.

8.11. Send Normal Data

$QIO(W)(IO$_WRITEVBLK)

You use the $QIO system service with a function code of IO$_WRITEVBLK to send a data unit of normal data over the transport service connection.

You can send data messages larger than the buffer size specified in *p2* by using the modifier IO $M_MULTIPLE. The call $QIO(IO$_WRITEVBLK!IO$M_MULTIPLE) indicates that the data supplied by the call is only part of a message, and that there is more to come. See *Section 6.6.4, "Fragmented Data Messages"* for more details.

Format:

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

Arguments:

func	IO$_WRITEVBLK

p1	Address of buffer
p2	Buffer length in bytes. The maximum length is defined by the SYSGEN parameter MAXBUF. The minimum length is 1 byte. Passed by value.
p3 - p6	Not used

Status Codes in R0:

SS$_ACCVIO	The task issuing the call has no read access to the device, mailbox name, string descriptor, buffer or IOSB; or the task issuing the call has no write access to the channel number, buffer or IOSB.
SS$_DEVOFFLINE	Either the OSI transport service has not yet been loaded or it has been unloaded.
SS$_FILNOTACC	There is no transport service connection associated with the channel specified in the write request.
SS$_ILLEFC	The call supplied an illegal event flag number.
SS$_ILLIOFUNC	The call supplied a function code that is unknown to the OpenVMS OSI transport service.
SS_$_ILLSER	An illegal system service was called.
SS_$_INSFARG	Not enough arguments were supplied in the call.
SS$_IVCHAN	An invalid channel number was supplied.
SS$_NORMAL	The OSI transport service has accepted and queued the write request.

Status Codes in the IOSB:

SS$_ABORT	The local transport service user issued a $CANCEL or $QIO(IO$_DEACCESS) call before the write request was processed.
SS$_FILNOTACC	There is no connection associated with the channel supplied in the write request. Either the connection was disconnected before the OSI transport service could process the call, or the task specified a nonexistent TC-ID.
SS$_LINKABORT	The remote transport service has sent a disconnection request (DR) TPDU, with an OSI reason code indicating an error. See *Appendix A, "$QIO(W) Status Codes and OSI Reason Codes"* for a list of OSI reason codes.
SS$_LINKDISCON	The remote transport service has sent a DR TPDU with the OSI reason code 80. This indicates a normal disconnect.
SS$_NORMAL	The OpenVMS OSI transport service has successfully processed the write request, and it will now be sent.
SS$_PATHLOST	Applies only to X.25 networks. The network connection has been disconnected or reset.

SS$_PROTOCOL	There has been a transport service protocol error. One of the following has happened: • The OpenVMS OSI transport service has received an error (ER) TPDU from the remote host. • The OpenVMS OSI transport service has received an incorrect or invalid TPDU from the remote host.
SS$_SHUT	The system manager has disabled the OSI transport service entity.
SS$_THIRDPARTY	The system manager has disabled the OSI transport service entity.
SS$_TIMEOUT	For Class 4 only. The write request has failed because the retransmission limit has been reached, or because the inactivity timer has expired.

8.12. Send Expedited Data

$QIO(W)(IO$_WRITEVBLK!IO$M_INTERRUPT)

You use the $QIO system service with a function code of IO$_WRITEVBLK and modifier IO $M_INTERRUPT to send up to 16 bytes of expedited data to a remote user. You cannot exchange expedited data on Class 0 transport service connections.

Format:

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

Arguments:

func	IO$_WRITEVBLK!IO$M_INTERRUPT
p1	Address of buffer
p2	Buffer length in bytes. Passed by value.
p3 - p6	Not used

Status Codes in R0:

SS$_ACCVIO	The task issuing the call has no read access to the device, mailbox name, string descriptor, buffer or IOSB; or the task issuing the call has no write access to the channel number, buffer or IOSB.
SS$_DEVOFFLINE	Either the OSI transport service has not yet been loaded or it has been unloaded.
SS$_FILNOTACC	There is no transport service connection associated with the channel specified in the write request.

SS$_ILLEFC	The call supplied an illegal event flag number.
SS$_ILLIOFUNC	The call supplied a function code that is unknown to the OSI transport service.
SS_$_ILLSER	An illegal system service was called.
SS_$_INSFARG	Not enough arguments were supplied in the call.
SS$_IVCHAN	An invalid channel number was supplied.
SS$_NORMAL	The OSI transport service has accepted and queued the request to send expedited data.

Status Codes in the IOSB:

SS$_ABORT	The local transport service user issued a $CANCEL or $QIO(IO$_DEACCESS) call before the write request was processed.
SS$_FILNOTACC	There is no connection associated with the channel supplied in the write call. Either the connection was disconnected before the OSI transport service could process the call or the task specified a nonexistent TC-ID.
SS$_ILLIOFUNC	The expedited data option is not available over this transport service connection.
SS$_LINKABORT	The remote transport service has sent a disconnection request (DR) TPDU, with an OpenVMS OSI reason code indicating an error. See *Appendix A, "$QIO(W) Status Codes and OSI Reason Codes"* for a list of OSI reason codes.
SS$_LINKDISCON	The remote transport service has sent a DR TPDU with the OSI reason code 80. This indicates a normal disconnect.
SS$_NORMAL	The OSI transport service has successfully processed the request, and it will now be sent.
SS$_PATHLOST	Applies only to X.25 networks. The network connection has been disconnected or reset.
SS$_PROTOCOL	There has been a transport service protocol error. One of the following has happened: ● The OSI transport service has received an error (ER) TPDU from the remote host. ● The OSI transport service has received an incorrect or invalid TPDU from the remote host.
SS$_THIRDPARTY	The system manager has disabled the OSI transport service entity.
SS$_TIMEOUT	For Class 4 only. The write request has failed because the retransmission limit has been reached, or because the inactivity timer has expired.

SS$_TOOMUCHDATA	The task specified more than 16 bytes of expedited data in the write request.

Chapter 9. System Service Calls Using Item Lists

This chapter describes the way item lists are used with $QIO(W) system service calls.

Item lists are part of an extended user interface to the $QIO(W) system services. Transport service users supply item lists as parameters to $QIO(W) calls in order to:

- Specify addressing and other information in outbound connection requests, including information that would otherwise be included in an NCB.
- Specify protocol classes and options in outbound connection requests.
- Examine the characteristics of inbound connection requests before they accept or reject them.
- Specify protocol classes and options when accepting an inbound connection request.
- Examine the actual characteristics of a transport service connection that has just been established.

If your applications will only run over the OpenVMS OSI transport service, item lists are the preferred way of using $QIO(W) calls. However, if you want your application to be able to run over either DECnet or the OpenVMS OSI transport service, you must use network connect blocks (NCBs).

Section 9.4, "Request a Transport Connection" to *Section 9.7, "Examine Request to Set Up a Transport Connection"* describe the OpenVMS system service calls you can use to communicate with the OpenVMS OSI transport service using item lists. See *Chapter 8, "System Service Calls Using Network Control Blocks"* for the format of system service calls that use NCBs rather than item lists.

The calls are in MACRO format. Each high-level language supported by OpenVMS has its own mechanism for calling and passing arguments to external procedures such as the OpenVMS system services. If you are using a high-level language, see the user's guide for specific information.

To specify an item list, you supply an address for an item list descriptor in the *p1* parameter of the $QIO(W) call. If there is no *p1* parameter, the OpenVMS OSI transport service assumes you are supplying an NCB as a parameter in *p2*. If *p1* is absent, *p2* is treated as an NCB, and *p3* is ignored.

You must supply either *p1* or *p2*, but not both.

9.1. Kinds of Item Lists

You use item lists when you initiate an outbound connection, accept or reject an inbound connection request, or examine an inbound connection request.

There are two kinds of item lists:

- An **input item list** contains data you supply to the OpenVMS OSI transport service about a transport service connection. It includes addressing information and your preferences for protocol class and options.
- An **output item list** is supplied by the OpenVMS OSI transport service. The OpenVMS OSI transport service places data about a connection in an output item list for you to use. For example, for connection requests, the output item list contains the protocol classes the OSI transport service and the remote transport service entity agreed to use.

You need to specify a buffer for an output item list.

You can only specify an output item list if you use an input item list. See *Section 6.3.3.3, "Structure of an Item in an Item List"* for the structure of an item in an item list.

9.1.1. Item Types

Table 9.1, "Item Types and Sizes" lists the item type codes, and shows:

- Whether they are used in input or output item lists, or both.
- For class, options, and protocol type, possible values (indicated by *).
- The permitted size of the item value field.

Table 9.1. Item Types and Sizes

Item Type Code	Input/Output Item lists	Size of Item Value Field
VMS OSIT$K_ITEM_ADDRESS	Input	String <= 64 bytes
VMS OSIT$K_ITEM_DESTINATION_NSAP	Input	String <= 64 bytes
VMS OSIT$K_ITEM_CALLED_TSAP	Input/Output	String <= 32 bytes
VMS OSIT$K_ITEM_CALLING_TSAP	Input/Output	String <= 32 bytes
VMS OSIT$K_ITEM_CHECKSUM	Input	Longword
VMS OSIT$K_ITEM_CLASS *VMS OSIT$M_CLASS_0 *VMS OSIT$M_CLASS_2 *VMS OSIT$M_CLASS_4	Input/Output	Longword bit mask
VMS OSIT$K_ITEM_EXPEDITED	Input	Longword
VMS OSIT$K_ITEM_EXTENDED	Input	Longword
VMS OSIT $K_ITEM_SEND_IMPLEMENTATION	Input	Longword
VMS OSIT$K_ITEM_NETWORK_SERVICE *VMS OSIT$K_NETWORK_SERVICE_CLNS *VMS OSIT$K_NETWORK_SERVICE_CONS *VMS OSIT$K_NETWORK_SERVICE_ANY	Output	Longword
VMS OSIT$K_ITEM_NULL	Input	String
VMS OSIT$K_ITEM_OPTIONS *VMS OSIT$M_CHECKSUM *VMS OSIT$M_EXPEDITED *VMS OSIT$M_EXTENDED	Input	Longword bit mask

Item Type Code	Input/Output Item lists	Size of Item Value Field
*VMS OSIT$M_FLOW_CONTROL *VMS OSIT $M_ITEM_SEND_IMPLEMENTATION		
VMS OSIT$K_ITEM_PROTOCOL_TYPE *VMS OSIT$K_OSI_PROTOCOL	Input/Output	Longword
VMS OSIT$K_ITEM_PROTOCOL_VERSION	Output	Longword
VMS OSIT$K_ITEM_SECURITY	Input	String
VMS OSIT$K_ITEM_TC_ID	Input/Output	Longword
VMS OSIT$K_ITEM_USER_DATA	Input/Output	Word Counted String <= 32 or 64 bytes

9.2. Input Item Lists

Table 9.2, "Use of Input Items in $QIO(W) calls" shows the item types used with the $QIO(W) system services to communicate with the OpenVMS OSI transport service.

Column 1 lists the item types.

Columns 2 to 5 show the system service calls:

Reje CR	Reject an inbound connection request (IO $_ACCESS!IO$M_ABORT)
Requ CR	Request an outbound connection (IO$_ACCESS)
Acce CR	Accept an inbound connection request (IO $_ACCESS)
Exam CR	Examine an inbound connection request (IO $_SENSEMODE!IO$M_ACCESS)

Table 9.2. Use of Input Items in $QIO(W) calls

Item Name	Reje CR	Requ CR	Acce CR	Exam CR
VMS OSIT$K_ITEM_ADDRESS	I	M	I	I
VMS OSIT $K_ITEM_DESTINATION_NSAP	I	M	I	I
Note: The two tags above are mutually exclusive.				
VMS OSIT$K_ITEM_CALLED_TSAP	I	O	O	I
VMS OSIT$K_ITEM_CALLING_TSAP	I	O	O	I
VMS OSIT$K_ITEM_CHECKSUM	N	O	O	N
VMS OSIT$K_ITEM_CLASS	N	O	O	N
VMS OSIT$K_ITEM_EXPEDITED	N	O	O	N

Item Name	Reje CR	Requ CR	Acce CR	Exam CR
VMS OSIT$K_ITEM_EXTENDED	N	O	O	N
VMS OSIT $K_ITEM_NETWORK_SERVICE	O	O	O	O
VMS OSIT$K_ITEM_NULL	O	O	O	O
VMS OSIT$K_ITEM_OPTIONS	N	O	O	N
VMS OSIT $K_ITEM_PROTOCOL_TYPE	I	M	M	M
VMS OSIT $K_ITEM_PROTOCOL_VERSION	N	N	N	N
VMS OSIT$K_ITEM_SECURITY	I	O	I	I
VMS OSIT $K_ITEM_SEND_IMPLEMENTATION	N	O	O	N
VMS OSIT$K_ITEM_TC_ID	M	N	M	M
VMS OSIT$K_ITEM_USER_DATA	O	O	O	N

Key:

M	Mandatory.
O	Optional.
I	The OpenVMS OSI transport service will ignore the item.
N	Not allowed; the OpenVMS OSI transport service will return an error.

You can supply the same item type more than once in an input item list; The OpenVMS OSI transport service will only use the last of the duplicated item types. This makes it easier to modify pre-built or supplied item lists. For example, it means you can alter output item lists to make input item lists, without too much rearranging and searching through the output item list.

9.2.1. Description of Input Items

This section describes the item types you can use in an input item list, and the values they can take.

9.2.1.1. Address (item type: VMS OSIT$K_ITEM_ADDRESS)

Type: string

This is the address of the remote host, in the form of an OpenVMS OSI transport service address. It is 64 bytes or less:

Template %NSAP

where

- Template in ASCII
- Remote address format based on the template network type (See *Section 6.4.5, "Addressing the Remote Host"*, Addressing the Remote Host, for more information.)

Note

If this tag is used, the tag OSIT$K_ITEM_DESTINATION_NSAP cannot be present.

9.2.1.2. Destination NSAP (item type: VMS OSIT $K_ITEM_DESTINATION_NSAP)

Type: String

This is the destination NSAP of the remote host, in the form of an OpenVMS OSI transport service address. It is 64 bytes or less.

Template %NSAP

where

- Template in ASCII
- Remote address NSAP in hexadecimal
- Network type (See *Section 6.4.5, "Addressing the Remote Host"*, Addressing the Remote Host)

Note

If this tag is used, the tag OSIT$K_ITEM_ADDRESS cannot be present.

9.2.1.3. Called TSAP (item type: VMS OSIT $K_ITEM_CALLED_TSAP)

Type: String

This is a TSAP identifier (TSAP-ID) of the responding transport service user. It is either up to 32 ASCII characters or an even number of up to 64 packed hexadecimal digits. You may use any characters in the TSAP-ID.

9.2.1.4. Calling TSAP (item type: VMS OSIT $K_ITEM_CALLING_TSAP)

Type: String

This is the TSAP-ID of the initiating transport service user. It is either up to 32 ASCII characters or an even number of up to 64 packed hexadecimal digits. You may use any characters in the TSAP-ID.

9.2.1.5. Class (item type: VMS OSIT$K_ITEM_CLASS)

Type: Longword

This bit mask indicates the allowed classes of transport service protocol on this transport service connection. The OpenVMS OSI transport service negotiates the class of protocol with the remote transport service entity. You may supply any or all of the classes supported by the OpenVMS OSI transport service; these are 0, 2 and 4.

It is not mandatory to supply this item to obtain Class 2 and Class 4 connections. If you do not supply it, the OpenVMS OSI transport service uses the value(s) specified by the CLASSES characteristic of the OpenVMS OSI transport service Template entity used for the connection. You identify the transport

service template in the OpenVMS OSI transport service address in VMS OSIT$K_ITEM_ADDRESS or in VMS OSIT$K_ITEM_DESTINATION_NSAP.

See *Section 6.4.5, "Addressing the Remote Host"* for more about addresses. See *Chapter 10, "Negotiating Protocol Classes and Options"* for details of how communicating transport service entities agree what class of transport service protocol to use.

9.2.1.6. Expedited Data (item type: VMS OSIT $K_ITEM_EXPEDITED)

Type: Longword

This item indicates whether you wish to exchange expedited data on this transport service connection. The OpenVMS OSI transport service negotiates this with the remote transport service entity. You may not set this flag to TRUE if Class 0 is the only allowed class in the connection request.

If you do not supply this item, the OpenVMS OSI transport service will use default values specified in the Expedited Data characteristic of the OpenVMS OSI transport service template entity used for this connection.

See *Chapter 10, "Negotiating Protocol Classes and Options"* for details of how communicating transport service entities negotiate expedited data.

9.2.1.7. Null (item type: VMS OSIT$K_ITEM_NULL)

Type: String

This item applies only to input item lists that are derived from output item lists. Use this item to replace an item that was in the output item list but is not required in the input item list.

9.2.1.8. Options (item type: VMS OSIT$K_ITEM_OPTIONS)

Type: Longword

This bit mask indicates the allowed transport service protocol options on this transport service connection. The options are:

VMS OSIT$M_EXTENDED	This applies to Class 2 and Class 4 connections only. It indicates whether the OpenVMS OSI transport service should try to negotiate the use of extended format TPDUs on this transport service connection. If you set this flag to true, OSI transport service will try to negotiate extended format. If preferred Class 0, this is ignored and normal format will be forced. This item indicates whether you wish to use extended format on this transport service connection. A larger sequence space will be used when sending OSI transport protocol data units (TPDUs). The format of these TPDUs will be larger than in normal format (extended format set to false). OpenVMS OSI transport service negotiates this with the remote transport service entity. You may not set this flag to true if Class 0 is the only allowed class in the connection request.

	If you do not supply this item, OpenVMS OSI transport service will use the value of the EXTENDED FORMAT characteristic of the OSI transport service template entity used for this connection. You identify a transport template in the OpenVMS OSI transport service-address used for VMS OSIT$K_ITEM_ADDRESS or for VMS OSIT$K_ITEM_DESTINATION.
VMS OSIT$M_FLOW_CONTROL	This is automatically set to true. This applies to Class 2 connections only. It indicates whether the OpenVMS OSI transport service will try to negotiate flow control on the transport service connection. Do not set this flag; the OpenVMS OSI transport service always uses explicit flow control on Class 2 transport connections.
VMS OSIT$M_EXPEDITED	This applies to Class 2 and Class 4 connections only. Set this to true to indicate that the OpenVMS OSI transport service should try to negotiate the use of expedited data on this transport service connection. Set it to false to indicate that OSI transport service should try to negotiate the non-use of expedited data.
VMS OSIT$M_CHECKSUM	This applies to Class 4 connections only. Set this to true to indicate that the OpenVMS OSI transport service should try to negotiate the use of checksums in TPDUs on this transport service connection. Set it to false to indicate that the OpenVMS OSI transport service should try to negotiate the non-use of checksums. If you do not supply this item, OSI transport service will use the value of the CHECKSUMS characteristic of the OSI transport template entity used for this connection. You identify a transport service template in the OpenVMS OSI transport service-address used for VMS OSIT$K_ITEM_ADDRESS or for VMS OSIT $K_ITEM_DESTINATION_NSAP.
VMS OSIT$M_SEND_IMPLEMENTATION	This applies to Class 2 and Class 4 connections only. Set this to true to indicate that OSI transport service should send the Network Architecture (NA)implementation ID on the connect request transport protocol data unit (TPDU). This item indicates whether you wish to have OSI transport service send the NA implementation ID in the OSI connect request TPDU. OSI transport service will use the implementation ID to determine whether both sides of the connection are implemented using the NA OSI transport protocol specification. Some liberties are allowed in this case that would not be allowed under the ISO8073

	OSI transport protocol specification. For instance, NA allows connections to send zero length data and allows more than 32 bytes of connect data in Class 2 and Class 4. In a non-NA implementation, these would be considered protocol errors and the connection rejected. It is important to keep in mind what remote systems you might be connecting to when choosing the send implementation, and choosing to implement the NA-only allowances. If you choose to stray from the ISO 8073:1991 specification, you must always set the send implementation ID to true. Also, be aware that if the remote system is a non-NA implementation, the restrictions of ISO 8073:1991 will be enforced and your connections might not succeed. Simply sending the implementation ID will not cause a problem to a non-NA implementation, but allowing such freedoms as connect data of more than 32 bytes in Class 2 and Class 4 connections, and zero-length data and extended data TPDUs most probably will cause a failure. If you do not supply this item, the OpenVMS OSI transport Service will use the value of the send implementation characteristic of the OSI transport Service Template entity used for this connection. You identify a transport service template in the OpenVMS OSI transport service address used for VMS OSIT$K_ITEM_ADDRESS or for VMS OSIT$K_ITEM_DESTINATION_NSAP. Note that the OSI TRANSPORT template DEFAULT will not allow the send implementation characteristic to be set to false. This template is used by DECnet-Plus for OpenVMS applications that require the NA implementation allowances.

Note that the preferred method for specifying options is to use the separate items VMS OSIT $K_ITEM_EXPEDITED and/or VMS OSIT$K_ITEM_CHECKSUM. The advantage of specifying options separately is that you only need include an item if you want to change its value from the default.

If you do not specify protocol options, the OpenVMS OSI transport service uses these default values:

- VMS OSIT$M_EXPEDITED — The default is the value specified in the Expedited Data characteristic of the OpenVMS OSI transport service template entity used for this connection, that is, true for ON, and false for OFF. You identify a transport service template in the OpenVMS OSI transport service address supplied for VMS OSIT$K_ITEM_ADDRESS or for VMS OSIT $K_ITEM_DESTINATION_NSAP.

Note

OpenVMS OSI transport service does not check to see if there is a mailbox associated with the channel to VMS OSIT$DEVICE. If there is no mailbox associated with the channel to VMS OSIT $DEVICE, the transport service user will only be able to send expedited data. If expedited data is received and there is no mailbox associated with the channel to VMS OSIT$DEVICE, OpenVMS OSI transport service will assume implicit success.

- VMS OSIT$M_CHECKSUM — The default is the value specified in the CHECKSUM characteristic of the OpenVMS OSI transport service template entity used for this connection, that is, true for ON, and false for OFF. You identify a transport service template in the OpenVMS OSI transport service address supplied for VMS OSIT$K_ITEM_ADDRESS or VMS OSIT $K_ITEM_DESTINATION_NSAP.

See *Chapter 10, "Negotiating Protocol Classes and Options"* for details of how the communicating transport service entities agree what option values to use.

9.2.1.9. Protocol Type (item type: VMS OSIT $K_ITEM_PROTOCOL_TYPE)

Type: Longword

Set this item to VMS OSIT$K_VMS_OSI_PROTOCOL.

9.2.1.10. Access Control (item type: VMS OSIT $K_ITEM_SECURITY)

Type: String

The remote transport service entity may require you to send access control information with your connection request. If the remote host is an OpenVMS OSI transport service host, the access control information is an OpenVMS user name and password, and will be used to start up a process to receive your connection request. See *Section 6.4.7, "Access Control Information in Outbound Connection Requests"* for more about access control information.

You cannot use this item if Class 0 is the only allowed class in the connection request.

9.2.1.11. TC Identifier (item type: VMS OSIT$K_ITEM_TC_ID)

Type: Longword

This item contains the transport service connection identifier to identify the transport service connection. You must supply this item when you accept, reject, or examine an inbound connection request.

You find the TC-ID for the connection by analyzing the NCB that the OpenVMS OSI transport service supplies to inform you of the connection request. See Sections *Section 6.5.3, "Examining the NCB"*, *Section 6.5.4, "Examining the Connection Request Using$QIO(IO$_SENSEMODE)"*, and *Section 6.5.5.1, "Accepting a Connection Request"* for details.

9.2.1.12. Optional User Data (item type: VMS OSIT $K_ITEM_USER_DATA)

Type: Word counted string

This item can be up to 32 bytes long.

This item can only be included on Class 2 and Class 4 connections. On Class 0 connections, connect data will cause a protocol error.

Note

If this connection will send the NA OSI transport implementation (see Send Implementation), then this item may contain more than 32 bytes of user data.

9.2.1.13. Network Service (item type: VMS OSIT $K_ITEM_NETWORK_SERVICE)

This longword will contain the network service type on an examine inbound connection request. It will take the following values:

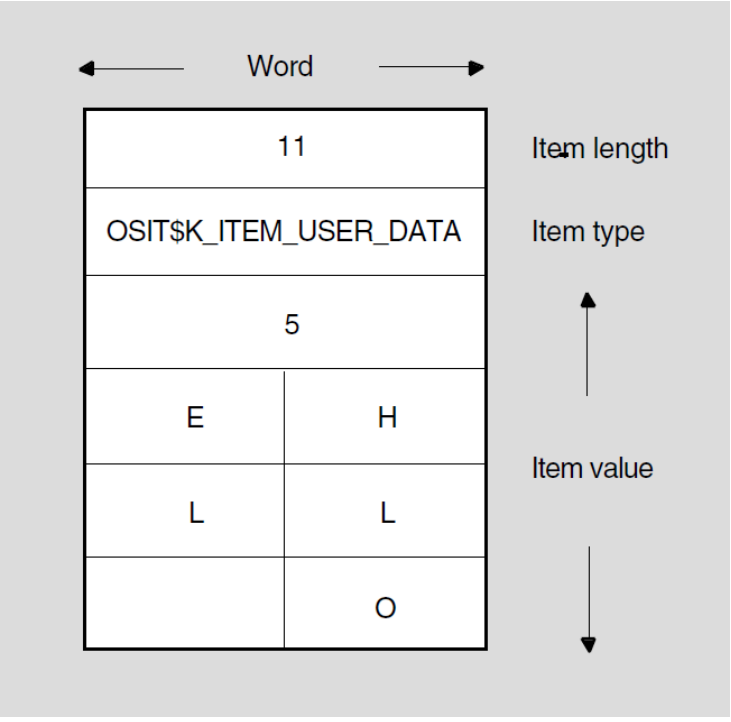
- OSIT$K_NETWORK_SERVICE_CLNS
- OSIT$K_NETWORK_SERVICE_CONS
- OSIT$K_NETWORK_SERVICE_RFC1006
- OSIT$K_NETWORK_SERVICE_ANY (either CLNS or CONS)

This allows a user to determine whether the transport is using a connectionless network service, a connection-oriented network service, or both.

9.2.1.13.1. Example

If you passed "HELLO" as user data, the item would have the structure shown in *Figure 9.1, "Example of an Input Item List"*.

Figure 9.1. Example of an Input Item List



9.3. Output Item Lists

You can specify an output item list in the following $QIO(W) calls:

- $QIO(W)(IO$_ACCESS) — Request an outbound connection.

  The OpenVMS OSI transport service returns the actual transport protocol options and classes negotiated for this transport connection.

- $QIO(W)(IO$_ACCESS) — Accept an inbound connection request.

  The OpenVMS OSI transport service returns the actual transport protocol options and classes negotiated for this transport connection.

- $QIO(IO$_SENSEMODE) — Examine an inbound connection request.

  The OpenVMS OSI transport service returns the transport protocol options and classes being requested by the remote user.

It is mandatory to specify a buffer for an output item list in a $QIO(IO$_SENSEMODE) call. It is optional for the other two calls.

There is no fixed order in the way the OpenVMS OSI transport service returns items in the item list. Always make sure that you analyze the item list properly. If there is more than one item of any type, the OpenVMS OSI transport service only places the last one in the output item list.

9.3.1. Description of Output Items

After the $QIO(W) call completes, the output item list will contain some or all of the following items.

Address (item type: VMS OSIT$K_ITEM_ADDRESS)

The address of the initiator of the transport service connection. This item is returned to $QIO(IO $_SENSEMODE) only.

Note

This tag will not be present if OSIT$K_ITEM_DESTINATION_NSAP is present.

Destination NSAP (item type: VMS OSIT $K_ITEM_DESTINATION_NSAP)

The address of the initiator of the transport service connection. This item is returned to $QIO(IO $SENSEMODE) only.

Note

This tag will not be present if OSIT$K_ITEM_ADDRESS is present.

Called TSAP (item type: VMS OSIT$K_ITEM_CALLED_TSAP)

The TSAP-ID of the responding user, that is, the task receiving the connection request. This item is returned to $QIO(IO$_SENSEMODE) only.

Calling TSAP (item type: VMS OSIT$K_ITEM_CALLING_TSAP)

The TSAP-ID of the initiating user, that is, the user sending the connection request. This item is returned to $QIO(IO$_SENSEMODE) only.

Class (item type: VMS OSIT$K_ITEM_CLASS)

This bit mask indicates which class of transport protocol the transport connection will use, or which classes of transport protocol are being requested in an inbound connection request.

Network Service (item type: VMS OSIT $K_ITEM_NETWORK_SERVICE)

This longword contains the network service type on an examine inbound connection request. It will take the following values:

- OSIT$K_NETWORK_SERVICE_CLNS
- OSIT$K_NETWORK_SERVICE_CONS
- OSIT$K_NETWORK_SERVICE_RFC1006
- OSIT$K_NETWORK_SERVICE_ANY (either CLNS or CONS)

This allows a user to determine whether the transport is using a connectionless network service or a connection-oriented network service or either.

Options (item type: VMS OSIT$K_ITEM_OPTIONS)

This bit mask indicates the transport protocol options that will be used or have been requested for this transport service connection. If the output item list is for a $QIO(W)(IO$_ACCESS) call, it shows the options that will be used; if it is for a $QIO(IO$_SENSEMODE) call, it shows the options requested in an inbound connection request.

The options are:

VMS OSIT$M_EXTENDED	This indicates whether normal or extended format will be used or has been requested. If set to true, extended format will be used or has been requested.
VMS OSIT$M_CHECKSUM	This indicates whether checksums will be used or have been requested. If set to true, checksums will be used or have been requested.
VMS OSIT$M_FLOW_CONTROL	This indicates whether flow control will be used or has been requested. If set to true, flow control will be used or has been requested.
VMS OSIT$M_EXPEDITED	This indicates whether expedited data can be used or has been requested. If set to true, expedited data can be used or has been requested.
VMS OSIT$M_SEND_IMPLEMENTATION	This indicates if the NA implementation ID was included in an outgoing OSI transport connect request TPDU.

Protocol Type (item type: VMS OSIT$K_ITEM_PROTOCOL_TYPE)

This is always set to VMS OSIT$K_VMS_OSI_PROTOCOL.

TC Identifier (item type: VMS OSIT$K_ITEM_TC_ID)

A longword containing the TC-ID that the OpenVMS OSI transport service has assigned to this transport service connection.

Optional User Data (item type: VMS OSIT$K_ITEM_USER_DATA)

This item may contain up to 32 bytes of user data and is only used on Class 2 and Class 4 connections.

Note

Note that if both sides of this connection employ the NA OSI transport protocol specification, and exchanged the NA OSI transport implementation ID (see Send Implementation), then this item may contain more than 32 bytes of user data.

9.4. Request a Transport Connection

$QIO(W)(IO$_ACCESS)

You use the $QIO system service with a function code of IO$_ACCESS to request an outbound connection. For Class 2 and 4 transport service connections, you may send up to 32 bytes of user data in this call.

Format:

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

Arguments:

func	IO$_ACCESS
p1	Address of a descriptor of the input item list.
p2	Not used.
p3	Address of a descriptor of the buffer to hold the output item list.
p4 - p6	Not used

Notes:

1. Use the *p1* item list to give the characteristics for the transport service connection.
2. If you specify a buffer for *p3*, the OpenVMS OSI transport service returns the actual characteristics negotiated for this transport connection. You must provide a routine to read this buffer; see *Section 6.4.10.3, "Reading the Output Item List"*.
3. The OpenVMS OSI transport service does not support multiple $QIO(W) calls to retrieve long item lists. If the *p3* buffer is too small for all the items, the OpenVMS OSI transport service returns an integral number of items and the status code SS$_BUFFEROVF in the IOSB.

   To specify the maximum size buffer, use the literal OSIT$K_MAX_OUTPUT_ITEM_LIST as *p3*.

Status Codes in R0:

SS$_ACCVIO	The task issuing the call has no read access to the device, mailbox name, string descriptor, buffer or IOSB; or the task issuing the call has no write access to the channel number, buffer or IOSB.
SS$_BADPARAM	One or more of the parameters *p1* to *p6* is not valid for this call. For example, an invalid item list, an invalid combination of parameters.
SS$_DEVOFFLINE	Either the OSI transport service has not yet been loaded or it has been unloaded.
SS$_EXQUOTA	Either the task issuing the request has not enough FILCNT to allow another connection or not enough BYTCNT for OpenVMS to allocate enough system resources to establish the connection.
SS$_FILALRACC	A connection already exists on the specified channel.
SS$_ILLEFC	The call supplied an illegal event flag number.
SS$_ILLIOFUNC	The call supplied a function code that is unknown to the OpenVMS OSI transport service.
SS$_ILLSER	An illegal system service was called.
SS$_INSFARG	Not enough arguments were supplied in the call.
SS$_INSFMEM	There is insufficient system dynamic memory (nonpaged pool) to allow the transport service connection to be established.
SS$_IVCHAN	An invalid channel number was supplied.
SS$_NOPRIV	The transport service user does not have the NETMBX privilege.
SS$_NORMAL	The OSI transport service has accepted and queued the connection request.

Status Codes in the IOSB:

SS$_ABORT	The task issued a $CANCEL or $QIO(IO $_DEACCESS) call before the connection request was processed.
SS$_BADPARAM	One or more of the parameters *p1* to *p6* is not valid for this call. For example, an invalid item list, an invalid combination of parameters.
SS$_BUFFEROVF	The output item list received is too long for the buffer specified in *p3*; some items have been lost.
SS$_CONNECFAIL	The connection request failed because of an Internet error.
SS$_INSFMEM	There is not enough system dynamic memory (nonpaged pool) for the connection to be established.

SS$_NOLINKS	The maximum number of concurrent transport connections has been reached, as defined by the MAXIMUM TRANSPORT CONNECTIONS characteristic of the OSI transport service entity.
SS$_NORMAL	The remote host has accepted the connection request. The connection is established.
SS$_NOSUCHNODE	The transport template specified in the OpenVMS OSI transport service address cannot be found.
SS$_NOSUCHOBJ	The specified TSAP-ID is unknown at the remote host.
SS$_PATHLOST	The remote host failed to reply within the required time.
SS$_PROTOCOL	There has been a transport protocol error: ● The local transport user failed to request Class 0 when required. ● Inbound connection confirm requested an invalid class. ● Inbound connection confirm requested an invalid protocol option. ● Format of the connection confirm is incorrect. For example, it specifies Class 0, but also specifies a protocol option that is invalid for Class 0. ● Invalid TPDU size in connection confirm. ● The remote host has rejected the request with one of the following OSI reason codes (hexadecimal): 83, 84, 85, 88, 90. See *Appendix A, "$QIO(W) Status Codes and OSI Reason Codes"* for the meanings of these codes.
SS$_REJECT	The remote user has rejected the connection request, and supplied one of the following OSI disconnect reason codes (hexadecimal): 80, 82. See *Appendix A, "$QIO(W) Status Codes and OSI Reason Codes"* for the meanings of these codes.
SS$_REMSRC	The remote host has not enough system resources to process the connection request.
SS$_SHUT	The system manager has disabled the OSI transport entity.
SS$_THIRDPARTY	The system manager has disabled the OSI transport entity.
SS$_TIMEOUT	The connection has failed because it timed out.

SS$_UNREACHABLE	The OpenVMS OSI transport service could not establish a network connection, for one of these reasons: • The network connection limit specified by the MAXIMUM NETWORK CONNECTIONS characteristic of the OSI transport entity has been exceeded. • The remote host has rejected the connection request. • The DTE address was incorrect or unknown.

9.5. Accept a Request to Set Up a Transport Connection

$QIO(W)(IO$_ACCESS)

You use the $QIO system service with a function code of IO$_ACCESS to accept an inbound connection request. For Class 2 and 4 transport service connections, you may send up to 32 bytes of user data in this call.

Format:

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

Arguments:

func	IO$_ACCESS
p1	Address of a descriptor of the input item list.
p2	Not used.
p3	Address of a descriptor of the buffer that is to hold the output item list.
p4 - p6	Not used

Notes:

1. Use the *p1* parameter to define the characteristics for the transport service connection. If you have examined the inbound connection request using $QIO(IO$_SENSEMODE), you may use the output item list returned from that call as the *p1* parameter to this call. You can modify the output item list to change the protocol classes or options specified.
2. Use the *p3* parameter to look at the actual characteristics of the connection that were negotiated between the OSI transport service and the remote transport.
3. The OpenVMS OSI transport service does not support multiple $QIO(W) calls to retrieve long item lists. If the *p3* buffer is too small for all the items, the OpenVMS OSI transport service returns an integral number of items and status code SS$_BUFFEROVF in the IOSB.

   To specify the maximum size buffer, use the literal VMS OSIT$K_MAX_OUTPUT_ITEM_LIST as *p3*. See *Section 6.3.3.2, "Output Item Lists"*.

Status Codes in R0:

SS$_ACCVIO	The task issuing the call has no read access to the device, mailbox name, string descriptor, buffer or IOSB; or the task issuing the call has no write access to the channel number, buffer or IOSB.
SS$_BADPARAM	One or more of the parameters *p1* to *p6* is not valid for this call. For example, an invalid item list, an invalid combination of parameters.
SS$_DEVOFFLINE	Either OSI transport service has not yet been loaded or it has been unloaded.
SS$_EXQUOTA	Either the task issuing the call has not enough FILCNT to allow another connection, or not enough BYTCNT for OpenVMS to allocate enough system resources to establish the connection.
SS$_FILALRACC	A connection already exists on the specified channel.
SS$_ILLEFC	The call supplied an illegal event flag number.
SS$_ILLIOFUNC	The call supplied a function code that is unknown to the OpenVMS OSI transport service.
SS_$_ILLSER	An illegal system service was called.
SS_$_INSFARG	Not enough arguments were supplied in the call.
SS$_INSFMEM	There is insufficient system dynamic memory (nonpaged pool) to allow the transport connection to be accepted.
SS$_IVCHAN	An invalid channel number was supplied.
SS$_NOPRIV	The transport service user does not have the NETMBX privilege.
SS$_NORMAL	OSI transport service has accepted and queued the connection accept.

Status Codes in the IOSB:

SS$_ABORT	The local transport service user issued a $CANCEL or $QIO(IO$_DEACCESS) call before the accept call was processed.
SS$_BADPARAM	One or more of the parameters *p1* to *p6* is not valid for this call. For example, an invalid item list, an invalid combination of parameters.
SS$_BUFFEROVF	The output item list received is too long for the buffer specified in *p3*; some items have been lost.
SS$_FILNOTACC	There is no connection associated with the channel supplied in the accept call. Either the connection was disconnected before the OSI transport service could process the call or the task specified a nonexistent TC-ID.

SS$_INSFMEM	There is not enough system dynamic memory (nonpaged pool) available to complete the connection request.
SS$_LINKABORT	The remote transport service has sent a disconnection request (DR) TPDU, with an OSI reason code indicating an error. See *Appendix A, "$QIO(W) Status Codes and OSI Reason Codes"* for a list of OSI reason codes.
SS$_NOPRIV	The TC-ID supplied with the accept call identifies a connection belonging to another user.
SS$_NORMAL	The transport connection is established.
SS$_PATHLOST	The remote host failed to acknowledge the accept call within the required time.
SS$_PROTOCOL	There has been one of the following transport protocol errors: • The local user supplied an invalid protocol class in the accept call. Either the class is unsupported by the OpenVMS OSI transport service, or it is not one of those requested by the remote transport service. • OSI transport service received an error TPDU from the remote host.
SS$_REJECT	The remote user has rejected the connection confirm, and supplied one of the following OSI disconnect reason codes (hexadecimal): 80, 82. See *Appendix A, "$QIO(W) Status Codes and OSI Reason Codes"* for the meanings of these OSI reason codes.
SS$_REMSRC	Class 4 only. The remote host has insufficient system resources to process the connection confirm.
SS$_SHUT	The system manager has disabled the OSI transport entity.
SS$_THIRDPARTY	The system manager has disabled the OSI transport entity.
SS$_TIMEOUT	The connection confirm has failed because it timed out.

9.6. Reject a Request to Set Up a Transport Connection

$QIO(W)(IO$_ACCESS!IO$M_ABORT)

You use the $QIO system service with a function code of IO$_ACCESS and modifier IO$M_ABORT to reject an inbound connection request. On Class 2 and Class 4 transport service connections, you can send up to 64 bytes of user data; this is optional.

Format:

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

Arguments:

func	IO$_ACCESS
p1	Address of a descriptor of the input item list.
p2 - p6	Not used

Modifier:

IO$M_ABORT	Use this modifier for a connection reject.

Status Codes in R0:

SS$_ACCVIO	The task issuing the call has no read access to the device, mailbox name, string descriptor, buffer or IOSB; or the task issuing the call has no write access to the channel number, buffer or IOSB.
SS$_BADPARAM	One or more of the parameters *p1* to *p6* is not valid for this call. For example, an invalid item list, an invalid combination of parameters.
SS$_DEVOFFLINE	Either the OSI transport service has not yet been loaded or it has been unloaded.
SS$_EXQUOTA	The task issuing the reject call does not have enough BYTCNT for OpenVMS to allocate enough system resources to reject the connection.
SS$_FILALRACC	A connection has already been established on the specified channel.
SS$_ILLEFC	The call supplied an illegal event flag number.
SS$_ILLIOFUNC	The call supplied a function code that is unknown to OpenVMS OSI transport service.
SS_$_ILLSER	An illegal system service was called.
SS_$_INSFARG	Not enough arguments were supplied in the call.
SS$_INSFMEM	There is insufficient system dynamic memory (nonpaged pool) for the OSI transport service to process the reject call.
SS$_IVCHAN	An invalid channel number was supplied.
SS$_NORMAL	OSI transport service has accepted and queued the connection rejection.

Status Codes in the IOSB:

SS$_BADPARAM	One or more of the parameters *p1* to *p6* is not valid for this call. For example, an invalid item list, an invalid combination of parameters.

SS$_FILNOTACC	There is no transport connection associated with the channel specified in the call. Either the connection was disconnected before the OSI transport could process it or the task specified a nonexistent TC-ID.
SS$_NOPRIV	The TC-ID supplied with the call identifies a connection belonging to another user.
SS$_NORMAL	The inbound connection request has been rejected. The transport connection is disconnected.
SS$_SHUT	The system manager has disabled the OSI transport entity.
SS$_THIRDPARTY	The system manager has disabled the OSI transport entity.

9.7. Examine Request to Set Up a Transport Connection

$QIO(W)(IO$_SENSEMODE!IO$M_ACCESS)

You use the $QIO(W) system service with a function code of IO$_SENSEMODE and modifier IO$M_ACCESS to examine an inbound connection request. Use this call to check whether the characteristics proposed for the connection are acceptable.

If you wish to accept an inbound connection request, use the output item list from this call as the input item list for the accept call. If you wish to negotiate an alternative set of characteristics, modify the output item list before you use it as the input item list in the accept call.

Format:

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

Arguments:

func	IO$_SENSEMODE
p1	Address of a descriptor of the input item list.
p2	Not used.
p3	Address of a descriptor of the buffer that is to hold the output item list.
p4 - p6	Not used

Modifier:

IO$M_ACCESS	Use this modifier to look at the request.

Notes:

1. You must include the item VMS OSIT$K_ITEM_TC_ID in the *p1*item list. Use the TC-ID in the NCB from the inbound connection request as the value for this item. You can use the library routine LIB$PARSE_NCB to analyze the NCB.

2. OpenVMS OSI transport service does not support multiple $QIO(W) calls to retrieve long item lists. If the *p3* buffer is too small for all the items, OpenVMS OSI transport service returns however many items fit into the buffer, and the status code SS$_BUFFEROVF in the IOSB.

   To specify the maximum size buffer, use the literal VMS OSIT$K_MAX_OUTPUT_ITEM_LIST as the descriptor in *p3*. See *Section 6.3.3.2, "Output Item Lists"*.

Status Codes in R0:

SS$_ACCVIO	The task issuing the call has no read access to the device, mailbox name, string descriptor, buffer or IOSB; or the task issuing the call has no write access to the channel number, buffer or IOSB.
SS$_BADPARAM	One or more of the parameters *p1* to *p6* is not valid for this call. For example, an invalid item list, an invalid combination of parameters.
SS$_DEVOFFLINE	Either the OSI transport service has not yet been loaded or it has been unloaded.
SS$_EXQUOTA	Either the task issuing the request has not enough FILCNT to allow another connection, or not enough BYTCNT for OpenVMS to allocate enough system resources to establish the connection.
SS$_FILALRACC	A connection already exists on the specified channel.
SS$_ILLEFC	The call supplied an illegal event flag number.
SS$_ILLIOFUNC	The call supplied a function code that is unknown to the OpenVMS OSI transport service.
SS_$_ILLSER	An illegal system service was called.
SS_$_INSFARG	Not enough arguments were supplied in the call.
SS$_INSFMEM	There is insufficient system dynamic memory (nonpaged pool) to allow the call to be processed.
SS$_IVCHAN	An invalid channel number was supplied.
SS$_NORMAL	OSI transport service has accepted and queued the call.
SS$_TOOMUCHDATA	The output item list received is too long for the buffer specified in *p3*; some items have been lost.

Status Codes in the IOSB:

SS$_NORMAL	OpenVMS OSI transport service has returned the output item list.
SS$_ABORT	The local transport service user issued a $CANCEL or $QIO(IO$_DEACCESS) call before the $QIO(IO$_SENSEMODE) call could be processed.

SS$_BADPARAM	One or more of the parameters *p1* to *p6* is not valid for this call. For example, an invalid item list, an invalid combination of parameters.
SS$_BUFFEROVF	The output item list received is too long for the buffer specified in *p3*; some items have been lost.
SS$_FILNOTACC	There is no connection associated with the channel supplied in the call. Either the connection was disconnected before the OSI transport service could process the call or the task specified a nonexistent TC-ID.
SS$_INSFMEM	There is not enough system dynamic memory (nonpaged pool) available to process the call.
SS$_NOPRIV	The TC-ID supplied with the accept call identifies a connection belonging to another user.
SS$_THIRDPARTY	The system manager has disabled the OSI transport entity.

Chapter 10. Negotiating Protocol Classes and Options

Before different OSI transport service implementations can work together, they must agree on the OSI transport protocol class and options to use. The OSI transport protocol standard provides rules and guidelines for negotiating this process.

This chapter describes the way NA OSI transport negotiates protocol classes and options.

10.1. Options Within the Transport Service Protocol Standard

When a transport service connection is being set up, the two communicating OSI transports need to negotiate:

- The class of transport service protocol to use
- The options (characteristics) of that class

An initiating OpenVMS OSI transport service user provides information about class and options when it issues a connection request. Specifically, it provides information about:

- Preferred class of protocol and suitable alternatives
- Preferred options

The initiating OSI transport sends these details to the responding OSI transport.

If the responding OpenVMS OSI transport service user accepts the connection request, it can issue a connection response which gives:

- The protocol class or classes it wants to use for this transport service connection
- The protocol options it wants to use for this OSI transport service connection

In this way, the two communicating OSI transports state the protocol class and options they will both support on the OSI transport connection. The outcome of negotiation depends partly on which preferences are acceptable to both users, and partly on rules of negotiation defined by ISO. For this reason, OSI transport may not be able to negotiate the preferred options and classes specified by a user.

The rest of this chapter describes how OSI transport negotiates transport protocol class and options. The sections are:

- Transport protocol version (see *Section 10.2, "Transport Service Protocol Version Number"*)
- Transport protocol class (see *Section 10.3, "Transport Protocol Class"*)
- Transport protocol options:
    - Transport protocol data unit (TPDU) format (see *Section 10.4.2, "Negotiating Protocol Options"*)

  - Expedited data (see *Section 10.4.1, "Specifying Checksums, Expedited Data, Extended Format and Send Implementation"*)
  - Checksums (see *Section 10.4.1, "Specifying Checksums, Expedited Data, Extended Format and Send Implementation"*)
  - Send implementation ID (see *Section 10.4, "Checksums, Expedited Data, TPDU Format and Send Implementation"*)
- Maximum TPDU size (see *Section 10.5, "Maximum TPDU Size"*)

10.2. Transport Service Protocol Version Number

OSI transport conforms to Version 1 of the OSI transport protocol standard, which does not require a version-number parameter to be included in connection request TPDUs. OSI transport does not transmit a version number in connection request TPDUs.

10.3. Transport Protocol Class

OSI Transport supports Classes 0, 2 and 4 of the OSI transport protocol. The class of protocol OSI transport chooses during negotiation with the remote transport service agrees with the class-selection algorithm defined the *International Standard ISO 8073:1986*.

OSI transport imposes the following restrictions on class negotiation:

- Connectionless Network Service (CLNS) supports Class 4 only.
- Connection-oriented Network Service (CONS) supports Classes 0, 2, and 4.

There are two $QIO(W) calls in which an OpenVMS OSI transport service user can specify a protocol class:

- Request an outbound connection
- Accept an inbound connection

10.3.1. Class Negotiation in Outbound Connection Requests

To specify protocol class, an OpenVMS OSI transport service user must use an input item list, and give the desired classes in the item VMS OSIT$K_ITEM_CLASS. A user is not required to specify protocol class. *Section 10.3.1.2, "OpenVMS OSI Transport Service User Does Not Specify Protocol Class"* describes how OSI transport negotiates if no class is specified. *Section 10.3.1.1, "OpenVMS OSI Transport Service User Specifies Protocol Class"* describes how OSI transport negotiates when a class is specified.

If a user specifies an output item list buffer in the connection request, OpenVMS OSI transport service will return the actual protocol class and options negotiated. However, note that you can only specify an

output item list if you also specify an input item list. See *Section 6.3.3.2, "Output Item Lists"* and *Section 6.4.10.3, "Reading the Output Item List"* for details.

10.3.1.1. OpenVMS OSI Transport Service User Specifies Protocol Class

If the OpenVMS OSI transport service user specifies a protocol class or classes, OSI transport checks them against the supported classes for the network service being used. Then it does one of the following:

- If all the classes in the connection request are supported, OSI transport sends on the connection request with these classes.
- If some of the classes in the connection request are supported, OSI transport sends on the connection request with the supported classes.
- If none of the classes in the connection request are supported, OSI transport refuses the connection request.

10.3.1.2. OpenVMS OSI Transport Service User Does Not Specify Protocol Class

If an OpenVMS OSI transport service user does not specify a protocol class in an outbound connection request, OSI transport selects the protocol classes from the CLASSES attribute of the OSI transport template being used.

10.3.2. Class Negotiation in Inbound Connection Requests

There are two stages in negotiating protocol class with inbound connection requests:

1. When an inbound connection request arrives, OSI transport checks that the classes specified are supported for the network service being used.

   If any of the classes specified in the connection request are supported, OSI transport passes the call to the OpenVMS OSI transport service user, specifying the preferred class and the deriving alternate classes.

   If none of the classes specified in the connection request are supported, OSI transport rejects the connection.

2. When the OpenVMS OSI transport service user accepts the connection request, it can specify one or more protocol classes in the accept call.

   If the user does not specify any class in the accept call, OSI transport selects the preferred protocol class as retrieved from the incoming connect request. OSI transport checks the user-supplied classes against the set of classes specified in the inbound connection request and the set of supported classes.

   - If the two sets of classes overlap, OpenVMS OSI transport service establishes the connection, using the highest class common to them all
   - If the two sets of classes do not overlap, OpenVMS OSI transport service cannot establish the connection. The accept call fails with a protocol error.

Note

The classes specified in the CLASSES characteristic of the inbound OpenVMS OSI transport service template selected when the connection request was received is not used at all in the inbound case of an OSI transport connection.

Before an OpenVMS OSI transport service user accepts or rejects a connection request, it can issue a $QIO(IO$_SENSEMODE) call to look at the requested classes. OpenVMS OSI transport service will return the requested classes in the output item list supplied with the call. The task can then decide whether the classes are acceptable or not. If they are not, it can reject the call.

If the transport service user specifies an output item list buffer in its accept call, OpenVMS OSI transport service will return the actual protocol class and options negotiated. See *Section 6.3.3.2, "Output Item Lists"* and *Section 6.4.10.3, "Reading the Output Item List"* for details.

Example 1

The classes specified in an inbound connection request over CONS are 0 and 2. The inbound OSI transport template specifies Classes 0, 2, and 4.

All the inbound connection classes are supported, so OpenVMS OSI transport service informs the transport service user of the connection request.

The transport service user specifies Class 4, which is not one of the classes held in common with the connect request. OSI transport cannot establish the connection, and the accept call fails with a protocol error.

Example 2

The classes specified in the inbound connection request over CONS and the inbound OSI transport template are the same as in **Example 1**.

The transport service user issues an accept call that specifies Class 0. OSI transport uses Class 0 because it is the highest class held in common by the transport service user, the connection request, and the supported CONS classes.

Example 3

The classes specified in the inbound connection request over CONS are the same as in **Example 1**.

Before accepting or rejecting the connection, the transport service user issues a $QIO(W)(IO $_SENSEMODE) call to examine the classes in the connection request. On the basis of this, it rejects the call because it only wants to use Class 4 on this connection.

Example 4

The classes specified in the connection request over CONS are the same as in **Example 1**.

The transport service user does not specify any protocol class in its accept call, so OSI transport selects Classes 0 and 2 from the inbound connect request on behalf of the user. OSI transport uses Class 2 because it is the highest class held in common by the connection request and the supported CONS classes.

10.3.3. Special Restrictions Applying to Class 0Connections

If you want to select Class 0 for transport service connections, you should note the following:

- OSI transport implements Class 0 on CONS only.
- Class 0 has only five types of TPDUs: connection request, connection confirm, disconnect request, data, and error response.
- If you select Class 0, you may not specify user data in any of the following calls: connection request, connection confirm, or disconnect requests.
- If you select Class 0, you may not specify security data in a connection request.
- Class 0 does not have any explicit flow control procedures. Therefore, you should have a $QIO(W) (IO$_READVBLK) call always outstanding to read data. To regulate the data flow, OpenVMS OSI transport service uses "back-pressure" on the X.25 network. This stops OpenVMS OSI transport service being flooded with data that the transport service user is unwilling to accept. Any higher-layer protocol that uses a Class 0 transport service connection should have flow control procedures.
- Expedited data is not allowed on Class 0 transport service connections. Regardless of how the item in the user's connector connect accept is set, expedited data will be forced to false.
- Checksums are not used on Class 0 transport service connections. Regardless of how the item in the user's connect requestor connect accept is set, checksums will be forced to false.
- Extended format TPDUs are not allowed on Class 0 transport service connections. Regardless of how the item in the user's connect request or connect accept is set, extended format will be forced to false.
- Implementation IDs are not allowed on Class 0 transport service connections. Regardless of how the item in the user's connect requestor connect accept is set, send implementation will be forced to false.
- A Class 0 transport service connection is not multiplexed with any other transport connection across a network connection.
- There are no acknowledgment TPDUs sent over Class 0 transport service connections.

10.4. Checksums, Expedited Data, TPDU Format and Send Implementation

When transport service users request or accept a connection, they can specify the following protocol options:

- **Checksums**. This indicates whether the user wants TPDUs to include checksums. Checksums detect whether data has been corrupted during transmission over the network.

  Checksums can only be included in TPDUs using Class 4 connections.

- **Expedited data**. This indicates whether the user wants to send or receive expedited data over this connection. Expedited data transfer enables users to send small messages that bypass normal flow control.

Expedited data can only be specified for connections using Class 2 or Class 4.

- **TPDU Format**. There are two types of TPDU format: normal and extended.

  Normal format allows a field of up to 7 bits for the sequence number. Extended format allows a field of up to31 bits for the sequence number. The sequence number shows the order in which the data TPDUs were transmitted.

  Extended format allows a much larger range of sequence numbers than normal format, which in turn allows a much larger credit window for TPDUs.

  Extended format can only be specified for connections using Class 2 or Class 4. Class 0 connections can only use normal format.

- **Send Implementation**. This indicates whether the user wants the NA implementation ID to be sent in an outgoing OSI connect request TPDU. This allows the OSI transport to recognize that the connection is between two NA implementations. In some specific cases, strict adherence to the ISO 8073 OSI transport protocol specification is relaxed. For example, in Class 2 and Class 4 connections, connect data may be more than 32 bytes. Note that if you choose to employ these NA-only allowances, you must be aware of the send implementation ID value and the OSI transport implementation on the remote system. Send implementation must be set to true, and you must be aware that this connection will be rejected by a non-NA implementation of OSI transport.

One option is automatically set by OpenVMS OSI transport service. This is flow control for Class 2.

10.4.1. Specifying Checksums, Expedited Data, Extended Format and Send Implementation

There are two $QIO(W) calls in which the transport service user can specify checksums, expedited data, extended format and send implementation:

- Request an outbound connection
- Accept an inbound connection

The transport service user specifies the use of these options in one of two ways:

- By specifying values for both options together, using the item VMS OSIT$K_ITEM_OPTIONS
- By specifying values for either or all of the options separately, using the items VMS OSIT$K_ITEM_EXPEDITED, VMS OSIT$K_ITEM_CHECKSUM, VMS OSIT $K_ITEM_EXTENDED, and/or VMS OSIT$K_ITEM_SEND_IMPLEMENTATION.

The default for expedited data, checksums, extended format and send implementation options are taken from the OSI transport template used for the connection.

If the transport service user wants to use expedited data, it should associate a mailbox with the channel to VMS OSIT$DEVICE. When expedited data is requested, OpenVMS OSI transport service does not check to see if there is a mailbox associated with the channel to VMS OSIT$DEVICE. If there is no mailbox associated with the channel to VMS OSIT$DEVICE and OpenVMS OSI transport service receives expedited data, the expedited data cannot be delivered. OpenVMS OSI transport service will assume implicit success.

10.4.2. Negotiating Protocol Options

When you request an option, OSI transport attempts to negotiate that option with the remote transport service. However, you may not always receive the option requested. The results of negotiation depend on three factors:

- Whether you are the initiator or responder
- The response of the remote transport service
- The rules applying to that option, as defined by transport service protocol standard ISO 8073:1986

Table 10.1, "Option Negotiation" summarizes the rules applying to option negotiation.

Table 10.1. Option Negotiation

Option	Proposal Made by Initiator	Valid Selection by Responder
Use of extended format (Classes 2, 4)	OFF ON	OFF OFF or ON
Use of expedited data (Classes 2, 4)	OFF ON	OFF OFF or ON
Use of checksums (Class 4)	OFF ON	OFF OFF or ON

ON means:

- **Extended Format**. The transport service has requested extended format TPDUs.
- **Expedited Data**. The transport service has requested expedited data.
- **Checksums**. The transport service has requested checksums in TPDUs.

OFF means:

- **Extended Format**. The transport service has requested no extended format TPDUs.
- **Expedited Data**. The transport service has requested non-use of expedited data.
- **Checksums**. The transport service has requested non-use of checksums.

10.5. Maximum TPDU Size

In the transport protocol standard, the maximum TPDU size can take one of these values: 128, 256, 512, 1024, 2048, 4096 or 8192 octets (an octet is a sequence of 8 bits). For Class 0, the maximum TPDU size is 2048 octets. Preferred maximum TPDU size offers a little better granularity, with values being multiples of 128 (128, 256, 512, 640, 768, and so forth). However, this parameter is not always implemented by the remote transport partner. OSI transport will attempt to use preferred maximum TPDU size, but will also include maximum TPDU size in case the remote does not implement the preferred parameter.

For VAX only, the maximum TPDU size selected is the largest valid maximum TPDU size that will fit into a network service data unit (NSDU). For CLNS, the maximum NSDU size is determined by the

Routing layer, in such a way that no segmentation of NSDUs is required in the Routing layer. For CONS, the maximum NSDU size is determined by the X.25 layer in such a way that no segmentation of NSDUs is required in the X.25 layer, the OSI transport template characteristic MAXIMUM NSDU SIZE is used as an upper boundary. The preferred maximum TPDU size is selected in the same manner.

Thus, for CLNS, the maximum TPDU and preferred maximum TPDU size is determined by the Routing layer, and cannot be altered by network management. For CONS, the maximum TPDU and preferred maximum TPDU size is determined by X.25 layer, and can be altered somewhat by network management (by altering X25 Access Template Packet Size characteristic and X25 Protocol DTE Default Packet Size characteristic and OSI transport template MAXIMUM NSDU SIZE characteristic).

10.5.1. Outbound Connection Requests

In outbound connection requests, OSI transport calculates the maximum and preferred maximum TPDU size to be used in the request. The actual size negotiated depends on the responding user:

- If the response specifies a value that is the same or smaller than the requested preferred maximum TPDU size, this is the value used.
- If the response does not give a value for preferred maximum TPDU size, OSI transport looks for the maximum TPDU size parameter. If there is a value, and the response specifies a value that is the same or smaller than the requested maximum TPDU size, this is the value used.
- If the response does not give a value for either preferred or maximum TPDU size, OSI transport uses 128 bytes for the size, as required by the transport protocol standard ISO 8073:1991.
- If the response specifies a value larger than your preferred or maximum TPDU size, the connection request fails with a protocol error.

10.5.2. Inbound Connection Requests

For inbound connection requests, OSI transport uses either the calculated TPDU size, or the TPDU size in the inbound connection request, or the preferred maximum TPDU size in the inbound connection request (if included), whichever is smaller. OSI transport communicates its choice in the connection confirm.

If neither TPDU size nor preferred maximum TPDU size is specified in the inbound connection request, OSI transport assumes 128 bytes for the TPDU size, as required by the transport protocol standard ISO 8073:1991.

Chapter 11. How OpenVMS OSI Transport Service Differs from DECnet-Plus for OpenVMS

OpenVMS OSI transport service, like DECnet-Plus for OpenVMS, allows you to implement a wide range of task-to-task applications. DECnet-Plus for OpenVMS includes both the Network Services Protocol (NSP) and OSI transport protocol. However, OpenVMS OSI transport service can only interface to the OSI transport.

11.1. Device Name

The suffix _NET is the device name used to assign a channel to DECnet-Plus for OpenVMS. The equivalent name for OpenVMS OSI transport service is VMS OSIT$DEVICE.

11.2. NCB Format

This section provide the format for outbound connection requests and inbound connection requests.

11.2.1. NCB Format for Outbound Connection Requests

In DECnet-Plus for OpenVMS, the NCB (network connect block) in an outbound connection request has the format:

node-name" *acc-info*"::"TASK= *taskname/ word-zero user-data*"

The fields are:

node-name	The name of a DECnet-Plus for OpenVMS node.
acc-info	An OpenVMS user name and password.
TASK=	A literal.
task name	A valid task specification string.
word-zero	A word set to zero for outbound connection requests.
user-data	Up to 16 bytes of optional user data.

In OpenVMS OSI transport service, the NCB in an outbound connection request looks like this:

resp-host" *acc-info*"::"TSAP= *tsap-id/ word-zero user-data*"

The fields are:

resp-host	An OpenVMS OSI transport service address (or a logical name for an OpenVMS OSI transport service address) to identify the responding host.
acc-info	An OpenVMS user name and password if the responding host is an OpenVMS OSI transport service system; otherwise it is access control information defined by the responding host.
TSAP=	A literal. Valid alternatives are TASK= and 0=.

tsap-id	The TSAP identifier for the responding user.
word-zero	A word set to zero for outbound connection requests.
user-data	Up to 32 bytes of optional user data.

11.2.2. NCB Format for Inbound Connection Requests

In DECnet-Plus for OpenVMS, the NCB from an inbound request looks like this:

node-name::"0= *taskname/ link-id user-data dest-desc*"

The fields are:

node-name	The name of the DECnet-Plus for OpenVMS node sending the connection request.
0=	A literal.
task name	A valid task specification string.
link-id	The DECnet-Plus for OpenVMS identifier for the logical link.
user-data	Up to 16 bytes of optional user data.

In OpenVMS OSI transport service, the NCB from an inbound request looks like this:

init-host::"TSAP= *calling-tsap-id/ tc-id user-data called-tsap-id*"

The fields are:

init-host	The OpenVMS OSI transport service address of the host initiating the request.
TSAP=	A literal. If the initiating host is a DECnet-Plus for OpenVMS system, this could also be 0=.
tsap-id	The TSAP-ID for the remote user; also known as the calling transport service access point (TSAP).
tc-id	A reference number assigned by OpenVMS OSI transport service to identify the transport service connection.
user-data	Up to 16 bytes of optional user data.
called-tsap-id	The TSAP-ID of the responding user.

11.3. User Data

The following sections discuss sending user data in outbound connection requests and in connection responses.

11.3.1. User Data in Outbound Connection Requests

In an outbound connection request, a DECnet-Plus for OpenVMS user can send up to 16 bytes of user data in a fixed-length field over NSP transport.

For Class 2 and Class 4, both DECnet-Plus for OpenVMS and OpenVMS OSI transport service let a user send up to 32 bytes of data in a variable-length field. You cannot send any user data in Class 0 connection requests.

See Chapter 9 for a discussion of sending the NA OSI transport implementation ID and allowing more than 32 bytes of connect data in Class 2 and Class 4 connections.

11.3.2. User Data in Connection Response

In response to a connection request, a DECnet-Plus for OpenVMS user can send up to 16 bytes of data in a fixed-length field over NSP transport.

For Class 2 and Class 4, both DECnet-Plus for OpenVMS and OpenVMS OSI transport service let a user send up to 32 bytes of data in a variable-length field. In Class 0, you cannot send any user data in response to a connection request.

See Chapter 9 for a discussion of sending the NA OSI transport implementation ID and allowing more than 32 bytes of connect data in Class 2 and Class 4 connections.

11.3.3. User Data in Disconnection Request

When disconnecting a link, a DECnet-Plus for OpenVMS user can send up to 16 bytes of data in a fixed-length field over NSP transport.

When disconnecting a Class 2 or Class 4 OSI transport service connection, both DECnet-Plus for OpenVMS and OpenVMS OSI transport service let a user send up to 64 bytes of data in a variable-length field. However, you cannot send any user data when concluding a Class 0 connection.

11.4. Access Control Information

For outbound connection requests over Class 0, access control information cannot be sent.

11.5. Identifying Tasks

OpenVMS OSI transport service identifies tasks on its own or remote systems by a TSAP-ID. This can be either:

- A packed hexadecimal string of up to 64 digits, or
- A string of up to 32 ASCII characters.

DECnet-Plus for OpenVMS identifies tasks by a TASK-ID. This is a string of up to 16 ASCII characters.

11.5.1. Identifying a Task in a NCB

In the network control block (NCB) in outbound connection requests, DECnet-Plus for OpenVMS allows the following formats when identifying the called task:

- TASK= *taskname*
- 0= *taskname*
- *taskname*=
- *number*=

where

task name is the task specification string for the called task and *number* is a network object number.

In the NCB in outbound connection requests, OpenVMS OSI transport service allows the following formats when identifying the equivalent of the called task, the responding user:

- TASK= *tsap-id*
- 0= *tsap-id*
- TSAP= *tsap-id*

where *tsap-id* is the identifier for the responding user.

OpenVMS OSI transport service does not let a transport service user use the formats:

taskname=
number=

11.6. Destination Address

DECnet-Plus for OpenVMS allows a user to define the destination of a connection request as a node name. OpenVMS OSI transport service expects an OpenVMS OSI transport service address.

11.7. Zero-Length TSDU

The transport service protocol standard states that all data TPDUs must carry at least one byte of user data. This means that a transport service user on a OpenVMS OSI transport service system cannot send a zero-length TSDU. If a transport service user attempts to send a zero-length TSDU, OpenVMS OSI transport service returns an SS$_BADPARAM status code to the write call.

However, if the connection has been established and the send implementation parameter has been set to true (see Chapter 9 for a description), NA OSI transport implementation IDs have been exchanged. Also, if this is a connection involving two NA implementations of OSI transport, then zero length TSDUs are allowed. DECnet-Plus for OpenVMS allows a user to send zero-length data units.

11.8. Logical Names

OpenVMS OSI transport service searches logical name tables in the following order:

1. First, it searches logical name tables following the conventions defined by OpenVMS.
2. Then, it searches the OpenVMS OSI transport service logical name table, VMSOSIT$NAMES.

See *VSI DECnet-Plus for OpenVMS Network Management Guide* for details of how DECnet-Plus for OpenVMS translates logical names.

11.9. Source Node Identifier

If an incoming call does not contain a source node identifier, DECnet-Plus for OpenVMS assumes that the call is intended for the current node.

OpenVMS OSI transport service expects a source node identifier in an incoming TPDU and rejects any TPDU that does not have a source node identifier.

11.10. Template Support for NA Session

NA session control uses the OSI transport DEFAULT template if none is specified during connection establishment. QIO connections do not provide a way to specify an OSI transport template. Specification of an OSI transport template can be done through the IPC or VOTS$QIO interfaces.

Chapter 12. CMISE Introduction

The DECnet-Plus for OpenVMS CMISE API is an application programming interface that implements the ISO 9595 Common Management Information Service specification on OpenVMS. The Association Control Service Element (ACSE) protocol of the Application layer, the Presentation layer, and Session layer are provided by Digital Equipment Corporation's OSAK software.

Management processes using the CMISE services on nodes in an OSI network are known as CMISE service users. Peer CMISE service users establish an association and exchange management information by means of the CMISE service primitives. The CMIS specification defines the service primitives and their parameters. There are three types of services:

1. Management Association Services consisting of:
   - M-Initialize: Used to establish an association.
   - M-Terminate: Used to terminate an association in a normal manner.
   - M-Abort: Used to terminate an association abruptly.
2. Management Operation Services consisting of:
   - M-Get: Used to retrieve attribute values.
   - M-Set: Used to modify attribute values.
   - M-Action: Used to perform a particular action.
   - M-Create: Used to create a new managed object instance.
   - M-Delete: Used to delete a managed object instance.
   - M-Cancel-Get: Used to cancel a previous M-Get service.
3. Management Notification Services consisting of:
   - M-Event-Report: Used to report an event.

12.1. Data Structures

This section gives a full description of the parameters used in the CMISE routines. The description of each parameter gives the following indications:

- The purpose of the parameter in the routines and the context where it is used.
- The application usage of the parameter and details on the parameter value coding.

In the following, when a parameter is said to be "ASN.1 encoded," it means that this parameter is encoded according to the transfer syntax {joint-iso-ccitt ASN.1(1) basics-encodings}.

Three basic data structures are defined:

```
struct cmise_buffer {                          /* cmise_buffer */
        unsigned short int cmise_w_alloc_len;  /* allocated buffer length*/
        unsigned short int cmise_w_used_len;   /* used length */
        unsigned char *cmise_a_pointer;        /* Ptr to a char buffer */
```

```
            } ;
```

This structure is used to pass/receive an ASN.1 encoded value. While passing the parameter value `(req/rsp) cmise_w_used_len` contains the length of the encoded value. While receiving a parameter value `(ind/cnf) cmise_w_alloc_len` contains the allocated buffer and `cmise_w_used_len` will be set to actual length of the received value.

```
struct cmise_oid {                                  /* cmise_oid */
            unsigned short int cmise_w_max_index; /* allocated max index */
            unsigned short int cmise_w_cur_index; /* current index */
            unsigned long int *cmise_a_array_ptr; /* Ptr to an array */
            } ;
```

This structure is used to pass/receive a local/global form of an identifier. While passing the value `(req/rsp) cmise_w_cur_index` contains the number elements present in the array.

If this is 0, then no id is specified.
If this is 1, then local form of Id is specified
If this is greater than 2, then Global form of ID (Object Identifier) is specified.

While receiving a parameter value `(ind/cnf)`, `cmise_w_max_index` specifies the size of the allocated array and `cmise_w_cur_index` will be set to the number of elements received.

`Cmise_pres_context` and `cmise_abort_context` are defined as synonyms for `cmise_oid`.

```
struct cmise_port {                                 /* cmise_port */
            unsigned char      *port_ptr;       /* osak port */
            unsigned char      *pb_ptr;         /* osak parameter block*/
            unsigned char       event_osak;     /* osak event */
            unsigned char       event_cmise;    /* cmise event */
            unsigned short int  more_on_hand;   /* more data on hand */
            long int            cmise_stat[5]   /* status block */
                      } ;
```

`*port_ptr` is a pointer to an OSAK PORT.

`*pb_ptr` is a pointer to an OSAK PARAMETER BLOCK.

See the *VSI DECnet-Plus OSAK Programming Reference Manual* for information on these OSAK data structures.

Two basic synonyms are defined:

`cmise_scope` – a synonym for unsigned long integer

`cmise_flag` – a synonym for unsigned long integer

12.2. Detailed Parameters

This section gives a full description of the detailed parameters used in CMISE routines.

12.2.1. Access Control

This parameter is used to give optional access control information. As the contents are only used by the user application, the value must be passed in a fully encoded form. This parameter is always optional.

This parameter is passed in a `cmise_buffer` data structure.

The buffer should contain a single ASN.1 encoding of type EXTERNAL.

12.2.2. Action Info

This parameter is used to pass more information about the action to perform and is always optional.

This parameter is passed in a `cmise_buffer` data structure.

The buffer should contain a single ASN.1 SET of ANY DEFINED BY action type.

12.2.3. Action Reply Info

This parameter is used to pass more information about the action reply and is always optional. Note that the presence of this parameter in the Action Response primitive is linked to the presence of the action type parameter.

This parameter is passed in a `cmise_buffer` data structures.

The buffer should contain a single ASN.1 SET of ANY DEFINED BY action type.

12.2.4. Action Type

The action type specifies a particular action that is to be performed. An action type may be passed under two different forms:

1. The local form, which is a unsigned integer.
2. The global form, which is an object identifier (an array of integers).

   If the action type is to be unspecified, `cmise_w_cur_index` must be zero.

This parameter is passed in a `cmise_oid` data structure. Contents of `cmise_w_cur_index` indicates which form (local/global) is being specified.

12.2.5. AE Invocation Identifier

The full name is "Application Entity Invocation Identifier." This longword is used to specify an application entity within a given application process. It is only exchanged during association establishment as following:

- For the initialize request, a CMISE user may specify its own AE invocation identifier (the calling AE invocation identifier), and the called AE invocation identifier.
- For the initialize response, a CMISE user may specify its own AE invocation identifier (the responding AE invocation identifier).

12.2.6. AE Qualifier

The full name is "Application Entity Qualifier." This longword is used to specify an application entity within a given application process. It is only exchanged during association establishment as follows:

- For the initialize request, a CMISE user may specify its own AE qualifier (the calling AE qualifier) and the called AE qualifier.

- For the initialize response, a CMISE user may specify its own AE qualifier (the responding AE qualifier).

12.2.7. AP Invocation Identifier

The full name is "Application Process Invocation Identifier." This longword is used to specify an invocation of an application process. It is only exchanged during association establishment as following:

- For the initialize request, a CMISE user may specify its own AP invocation identifier (the calling AP invocation identifier), and the called AP invocation identifier.
- For the initialize response, a CMISE user may specify its own AP invocation identifier (the responding AP invocation identifier).

12.2.8. Application Context Name

This parameter is an object identifier exchanged during the association establishment and subject to negotiation. The possible values are the results of agreements on association characteristics (permitted actions, application service elements involved in the data exchange, and so forth).

This parameter is passed in a `cmise_oid` data structure. Note that specifying an application context name is always mandatory.

12.2.9. AP Title

The full name is "Application Process Title." This parameter is used to specify an application process on a given system. It is only exchanged during association establishment as following:

- For the initialize request, a CMISE user may specify its own AP title (the calling AP title) and the called AP title.
- For the initialize response, a CMISE user may specify its own AP title (the responding AP title).

AP title is an object identifier. This parameter is passed in a `cmise_oid` data structure. This is an optional parameter.

12.2.10. Association User data

This parameter is used to pass additional user data during association establishment and abort requests.

This parameter is passed in a `cmise_buffer` data structure.

12.2.11. Attribute Identifier List

This parameter is only used in the Get operation request. The attribute identifier list carries the set of attribute identifiers whose values are requested by the get operation invoker. If the list is empty, all attributes values are supposed to be requested.

This parameter is passed in a `cmise_buffer` data structure.

The value is ASN.1 encoded, and consists of a set of attribute identifier. If the list is empty, `cmise_w_used_len` will be zero.

12.2.12. Attribute List

The attribute list carries a set of attributes, which is a sequence of attribute identifier and attribute value. Attribute lists are used in Set, Get and Create services.

This parameter is passed in a `cmise_buffer` data structure.

The value is ASN.1 encoded and consists of a set attributes.

12.2.13. CMISE Error Code

This parameter is only used in the Error service to give the reason of the error. The possible values are listed in the Error service primitives.

This parameter is passed in an integer.

12.2.14. Connection Id

This parameter is used in each service to identify the association. It is filled in by the `M_Initialize_Req` service when an association is established. It is then used by each service request that is servicing the same association.

This parameter is passed in a `cmise_port` data structure.

12.2.15. Context Identifier List

This parameter is only used in the abort service. Its purpose is to avoid any ambiguity on the presentation contexts used to decode the abort user information. This parameter is passed in a `cmise_abort_context` and may be empty if no user abort information is provided.

Each presentation context identifier is composed of two longwords. The first longword contains the context identifier and the second word contains the transfer syntax.

Transfer syntax specifies the transfer syntax effectively used to encode a presentation data value present in the user information parameter. At present, only P_K_BER_ASN1 is supported for transfer syntax.

12.2.16. Event Code

This parameter is used only in the error positive response primitive. It specifies the failed operation. It is only meaningful if bit M_FG_LINKED_ID is set in the flag.

The following are valid values:

- M_E_GET_RESP for Get
- M_E_SET_RESP for Set
- M_E_ACTION_RESP for Action
- M_E_DELETE_RESP for Delete

12.2.17. Event Info

This parameter is used to pass more information about the reported event and is always optional.

This parameter is passed in a `cmise_buffer` data structure.

The buffer should contain a single ASN.1 SET of ANY DEFINED BY event type.

12.2.18. Event Reply Info

This parameter is used to pass more information about the reported event reply.

This parameter is always optional. Note that the presence of this parameter in the Event Response primitive is linked to the presence of the event type parameter.

This parameter is passed in a `cmise_buffer` data structure.

The buffer should contain a single ASN.1 SET of ANY DEFINED BY event type.

12.2.19. Event Type

The event type specifies the type of an occurring event. An event type may be passed under two different forms:

- The local form, which is a signed integer.
- The global form, which is an object identifier (an array of integers).

If the event type is to be unspecified, `cmise_w_cur_index` must be zero.

This parameter is passed in a `cmise_oid` data structure. The contents of `cmise_w_cur_index`indicates which form (local/global) is being specified.

12.2.20. Filter

The filter is a boolean expression involving attribute values to be evaluated for all selected objects. If the result for an object is true, the operation is executed on that object; otherwise, the object is discarded.

This parameter is passed in a `cmise_buffer` data structure.

The value is ASN.1 encoded.

12.2.21. Flags

Most CMISE services contain a FLAGS parameter. These flags are used to specify service options that can take only a small set of values (two for most of them). This parameter is a bit mask and bit constants are provided to ease handling of these bits. A bit can be set performing a boolean OR, and tested by performing a boolean AND with its bit constant.

The CMISE entity makes use of the following flags:

- Linked identifier presence: set if the linked identifier parameter is meaningful.

  (bit constant name: M_FG_LINKED_ID)

- Mode: set if a confirmed operation is requested

  (bit constant name: M_FG_CONFIRMED)

- Synchronization: can be specified in three different ways:

  - bit constant name: M_FG_NOSYNC

    Set if no synchronization is to be encoded in PDU. If set, overrides any meaning of the following flag.

  - bit constant name: M_FG_ATOMIC_SYNC

    Set if the requested synchronization is atomic.

  - bit constant name: M_FG_ATOMIC_SYNC

    If not set, the default is to request best effort synchronization.

- Superior instance: set if the given object instance specifies a superior object.

  (bit constant name: M_FG_SUPER_INST)

- Complex filter: set if the specified filter is too complex, `cmise_error_code` should be M_CD_COMPLX_LIMIT.

  (bit constant name: M_FG_COMPLEX_FILTER)

- Complex scope: set if the specified scope is too complex, `cmise_error_code` should be M_CD_COMPLX_LIMIT.

  (bit constant name: M_FG_COMPLEX_SCOPE)

- Complex synch: set if M_FG_ATOMIC_SYNC is too complex, `cmise_error_code` should be M_CD_COMPLX_LIMIT.

  (bit constant name: M_FG_COMPLEX_SYNCH)

- PDU response format: set to only encode the InvokeId.

  (bit constant name: M_FG_RORS_BASIC)

Next three flags are set in `M_Reject_Cnf` alone.

- Invoke id: set if `M_Reject_Cnf` returned an invoke id.

  (bit constant name: M_FG_INVOKE_ID)

- Problem type: set if `M_Reject_Cnf` returned a problem_type.

  (bit constant name: M_FG_PROB_TYPE)

- Problem number: set if `M_Reject_Cnf` returned a problem_number.

  (bit constant name: M_FG_PROB_NUM)

All other bits are unused.

12.2.22. Functional Units

The set of functional units successfully negotiated on a CMISE association determines the permitted operation requests. The negotiated functional units are always a subset of those proposed by the initiating

user. The only exception is the multiple reply functional unit that may be implicitly negotiated with the multiple object selection. Each functional unit is mapped on one bit, which is set if the functional unit is required.

The following functional units are selectable:

- Multiple object selection: allows the selection of several managed objects in a single request (constant M_FU_MUL_OBJ_SEL). Note that this functional unit implies the multiple reply functional unit.
- Filter: allows the use of filters to select managed objects

  (constant M_FU_FILTER).
- Multiple reply: allows sending several responses for a single operation

  (constant M_FU_MUL_REPLY).
- Extended service: provides the Presentation service at the CMISE service interface (constant M_FU_EXTEND_SVCE). Note that this functional unit is currently not supported, it will therefore never be negotiated successfully.
- Cancel-get: allows the use of the Cancel_Get service

  (constant M_FU_CANCEL_GET).

12.2.23. Invoke Identifier

This is the identifier of the requested operation, given by the operation invoker. Any value may be chosen, provided that there is no pending operation requested by this user with the same invoke identifier. If the operation performer needs to send a response, the same identifier must be specified.

12.2.24. Linked Identifier

This parameter is passed by the performer when several responses are sent to link all the responses to the same operation. When an operation performer sends a partial response, the linked identifier parameter must have the value of the request invoke identifier.

12.2.25. Network Service Access Point (NSAP)

This parameter is used for initialize request/indication. In the request it specifies the called or calling network service access point. In the indication it gives the called or calling network service access point.

This parameter is passed in a `cmise_buffer` data structure.

12.2.26. NSAP Type

This parameter is for future use, and is not supported at present.

12.2.27. Object Class

The object class parameter specifies the class of a managed object. An object class may be passed under two different forms:

- The local form, which is a unsigned integer.

- The global form, which is an object identifier (an array of integers).

If the object class is to be unspecified, `cmise_w_cur_index` must be zero.

Use `cmise_oid` data structure to pass or receive the value. The contents of `cmise_w_cur_index` indicates which form (local/global) is being specified.

12.2.28. Object Instance

The object instance parameter specifies the instance of an object, for example, the unique name of a managed object within a network management containment tree. It can be specified in three different ways:

```
ObjectInstance ::=      CHOICE {
distinguishedName       [2] IMPLICIT DistiguishedName ,
nonSpecificForm         [3] IMPLICIT OCTET STRING,
localDistinguishedName  [4] IMPLICIT RDNSequence }
```

This parameter is passed in a `cmise_buffer` data structure.

The buffer should contain an ASN.1 encoding of object instance.

If no object instance is specified, a null pointer must be given.

12.2.29. Presentation Context Definition List

This parameter is used during the connection establishment to specify the list of the presentation contexts the user wants to be negotiated. For all the association responses, the parameter carries the negotiated result.

This parameter is passed in a `cmise_pres_context`.

The presentation context definition list is essentially an array of unsigned long integers. This array must be encoded as follows:

```
Integer 0 should have the 1st Abstract Syntax.
  Allowed values are: P_K_ACSE and PK_CMISE

Integer 1 should have 1st Presentation context identifier.

Integer 2 has - Status (1st octet) (meaningless but mandatory
  in connection request).
  This octet is divided in two parts of 4 bits each:

        - 4 high order bits are set to:
          0: ok (indication, response and confirmation)
             the 4 low order bits are meaningless,
          1: user rejection (response and confirmation),
          2: provider rejection (indication, response
             and confirmation).

        - 4 low order bits are set to:
          0: ok or no specified reason,
          1: abstract syntax not supported,
          2: transfer syntax not supported,
          3: local limit on DCS exceeded.
```

```
Integer 3 should have number of transfer syntax specified.
  For now this should have a value of 1.

Integer 4 should have the transfer syntax.
  For now only allowed value is P_K_BER_ASN1

Integer 5 should have the 2nd Abstract Syntax.
  Allowed values are: P_K_ACSE and PK_CMISE

Integer 6 should have 2nd Presentation context identifier.

Integer 7 should have status

Integer 8 should have number of transfer syntax specified.
  For now this should have a value of 1.

Integer 9 should have the transfer syntax.
  For now only allowed value is P_K_BER_ASN1
```

Note that CMISE needs at least the successful negotiation of the ACSE and CMISE abstract syntaxes, combined with the ASN.1 BER transfer syntax.

12.2.30. Presentation Selector (PSEL)

This parameter is use for initialize request/indication. In the request, it specifies the called or calling presentation selector. In the indication it gives the called or calling presentation selector. This parameter is passed in a `cmise_buffer` data structure.

12.2.31. Problem Number

This parameter is only used in the Reject services. It is an integer value that gives a specific reason for a reject. The value of this parameter is dependent on the Problem Type parameter. Possible values are:

M_SE_UNRECOGNIZE	M_SE_MISTYPED_PDU
M_SE_BAD_STRUCT	M_SE_DUP_INVOKE
M_SE_UNKNOWN_OP	M_SE_MISTYPED_ARG
M_SE_RES_LIMIT	M_SE_RELEASING
M_SE_UNK_LINK	M_SE_LINK_EXPEC
M_SE_CHILD_OP	M_SE_UNK_INVOKE
M_SE_UNEX_RESULT	M_SE_MISTYPED_RES
M_SE_UNK_INVOKE	M_SE_UNEX_ERR_RES
M_SE_UNK_ERROR	M_SE_UNEX_ERROR
	M_SE_MISTYPED_ERR

12.2.32. Problem Type

This parameter is only used in the Reject services. It is an integer value that gives a general reason for a reject. Possible values are:

- M_PR_GENERAL_PROB

- M_PR_INVOKE_PROB
- M_PR_RESULT_PROB
- M_PR_ERROR_PROB

12.2.33. Protocol Version

The protocol version is a bit mask used to specify each CMIP protocol supported by the CMISE user. The negotiated protocol version is the highest CMIP version supported by both CMISE users. This parameter is a bit mask with the following bits defined:

- CMIP_K_PROT_VER_1
- CMIP_K_PROT_VER_2

All other bits are unused.

12.2.34. Reference Object Instance

This parameter is used only in the Create service. It is used to specify the name of an existing object that may be used as a model to create the new object. At the CMISE interface level, the rules for the object instance parameter apply to this parameter (see the object instance parameter description above). If no reference object instance is specified, a null pointer must be given. Note that it is impossible to specify an empty reference object instance. If the buffer is null, the instance is assumed to be absent.

This parameter is passed in a `cmise_buffer` data structure.

12.2.35. Refuse Reason

This parameter is used to specify the reason of the refuse when an association is refused.

The possible values are outlined as follows:

If the refuse source is the ACSE user:

- M_RU_REASON_UNK: no reason given
- M_RU_ACN_NOT_SUPP: application context name not supported
- M_RU_BAD_AP_TITL_G: calling AP title not recognized
- M_RU_BAD_AP_INV_ID_G: calling AP invocation id not recognized
- M_RU_BAD_AE_QUAL_G: calling AE qualifier not recognized
- M_RU_BAD_AE_INV_ID_G: calling AE invocation id not recognized
- M_RU_BAD_AP_TITL_D: called AP title not recognized
- M_RU_BAD_AP_INV_ID_D: called AP invocation id not recognized
- M_RU_BAD_AE_QUAL_D: called AE qualifier not recognized
- M_RU_BAD_AE_INV_ID_D: called AE invocation id not recognized

If the refuse source is the ACSE provider:

- M_RP_REASON_UNK: no reason given
- M_RP_NO_COMM_VERS: no common ACSE version

If the refuse source is the presentation provider:

- M_RN_USER_REJ: user rejected
- M_RN_LOC_LIM_EXEED: local limit exceeded
- M_RN_USER_DATA_UNREAD: user data not readable

12.2.36. Release Urgency

This parameter is used in the terminate request/indication service to specify the urgency of the release.

The possible values are:

- M_UR_NORMAL: normal
- M_UR_URGENT: urgent
- M_UR_USER_DEF: user defined

12.2.37. Scope

The scope parameter is applicable only if the multiple object selection functional unit has been proposed by the application, and several objects are actually selected. The scope specifies the part of the containment tree involved by a network management service invocation. The scope is applied on the containment subtree whose root is the specified base object. It may have five different values:

- Base object only: the base object is the only object involved in the request.
- First level only: the request must be applied to all objects, first level subordinates, located at one level below the base object.
- Whole subtree: the request must be applied to the base object and all subordinate all objects contained in the subtree.
- Only nth level: the request must be applied to all objects located at an nth level below the base object.
- Up to nth level: the request must be applied to all objects contained in the base object subtree, whose level is not greater than the given value.

This parameter is passed in `cmise_scope` (unsigned long integer) in the following manner:

```
                          4 byte representation:
                                                3 2 1 0
base object alone:                              0 0 0 0
first level subordinates:                       0 0 0 1
base object and all subordinates:               0 0 0 2
the nth level below the base object:            0 0 1 n
base object and all subordinates to level n:    0 0 2 n
```

This means that the object hierarchy is limited to 256 levels.

12.2.38. Service Data

This parameter is only used in the Error service to give additional error data. The value of the parameters and the error codes that require the additional data are listed in the Error service primitives.

This parameter is passed in a `cmise_buffer` data structure.

12.2.39. Session Connection Identifier

This parameter is for future use, and is not supported at present.

12.2.40. Session Selector (SSEL)

This parameter is used for initialize request/indication. In the request, it specifies the called or calling session selector. In the indication, it gives the called or calling session selector.

This parameter is passed in a `cmise_buffer` data structure.

12.2.41. Source Reason

This parameter is used in a user abort indication to specify where the abort has been issued.

The possible values are listed below:

```
M_AS_ACSE_PROV          : ACSE provider (local or remote) ;
M_AS_LOC_CMISE_PROV     : local CMISE provider ;
M_AS_REM_CMISE_PROV     : remote CMISE provider ;
M_AS_REM_CMISE_USER     : remote CMISE user.
```

12.2.42. Template

This parameter is for future use, and is not supported at present.

12.2.43. Time

For some services, this parameter is used to pass a time (for example, the time of an event, of a response). Its size is 24 and contains MCC or DTSS binary absolute time. This parameter is passed in a `cmise_buffer` data structure.

12.2.44. Transport Selector (TSEL)

This parameter is use for initialize request/indication. In the request, it specifies the called or calling transport selector. In the indication, it gives the called or calling transport selector.

This parameter is passed in a `cmise_buffer` data structure.

12.3. Using the CMISE API

Two peer CMISE users establish an association and exchange management information using the CMISE services. The peer CMISE users are known as the invoking CMISE service user and the performing

CMISE service user. The invoking CMISE service user invokes management requests that are performed by the performing CMISE service user.

The CMISE API is implemented as a shareable image that is linked with the user application program. An "include" file is supplied that defines function prototypes and data definitions. The CMISE API software is installed as part of the DECnet-Plus for OpenVMS base components.

To compile C programs that use the CMISE API, add the following line to the C source module:

```
#include <net_cmise.h>
```

To link your program with the API, you must include the following line in the options file which is input to the linker:

```
NET$CMISE/SHAREABLE
```

Chapter 13. Common Management Information Services

13.1. M_INITIALIZE Service

This service is used by a CMISE user to establish an association with a peer CMISE user. It must, therefore, be used before sending any CMISE operation request.

The Initialize service is directly mapped on the ACSE associate service.

The logic of this service is as follows:

- **Initiator Side**: Once the user has invoked the M_Initialize_Req primitive, it may receive either an M_E_INITIALIZE_ACC event if the association is established, or an M_E_INITIALIZE_REJ if the association has been refused.
- **Responder Side**: Once the user has received an M_E_ASSOCIATE_IND event, it may invoke either the M_Accept primitive to accept the association or the M_Reject primitive to refuse it.

13.1.1. M_INITIALIZE Request

M_Initialize_Req

This function sends a M_INITIALIZE request to the CMISE entity to establish an association with a peer CMISE entity. Therefore, it must be used before sending any CMISE operation request.

Format

```
M_Initialize_Req (Connection_Id,
                  Called_PSEL, Called_SSEL, Called_TSEL, Called_NSAP,
                 [Calling_PSEL], [Calling_SSEL], [Calling_TSEL],
                 [Calling_NSAP],
                 [Calling_AP_Inv_Id], [Calling_AE_Inv_Id],
                 [Called_AP_Inv_Id], [Called_AE_Inv_Id],
                  Pres_Cont_Def_List, [Sess_Connect_Id], Appli_Cont_Name,
                 [Calling_AP_Title], [Calling_AE_Qual],
                 [Called_AP_Title], [Called_AE_Qual],
                 [NSAP_Type], [Template], Protocol_Version,
                  CMISE_Fu, [Access_Control], [User_Info])
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	write only
Mechanism:	by reference

This parameter identifies the association created by this function.

Called_PSEL

Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the called presentation selector.

Called_SSEL	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the called session selector.

Called_TSEL	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the called transport selector.

Called_NSAP	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the called network service access point.

Calling_PSEL	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the calling presentation selector.

Calling_SSEL	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the calling session selector.

Calling_TSEL	
Type:	cmise_buffer
Access:	read only

Mechanism:	by reference

This parameter defines the calling transport selector.

Calling_NSAP	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the calling network service access point.

Calling_AP_Inv_Id	
Type:	longword
Access:	read only
Mechanism:	by reference

This parameter identifies the calling application process invocation. Its value is in the range of 0 to 999998.

Calling_AE_Inv_Id	
Type:	longword
Access:	read only
Mechanism:	by reference

This parameter identifies the calling application entity invocation. Its value is in the range of 0 to 999998.

Called_AP_Inv_Id	
Type:	longword
Access:	read only
Mechanism:	by reference

This parameter identifies the called application process invocation. Its value is in the range of 0 to 999998.

Called_AE_Inv_Id	
Type:	longword
Access:	read only
Mechanism:	by reference

This parameter identifies the called application entity invocation. Its value is in the range of 0 to 999998.

Pres_Cont_Def_List	
Type :	cmise_pres_context
Access:	read only
Mechanism:	by reference

The full name is Presentation Context Definition List. This parameter is used to specify the list of the presentation contexts the user wants to be negotiated.

Note

Currently only P_K_BER_ASN1 is allowed for transfer syntax. Currently only P_K_CMISE and P_K_ACSE are allowed for abstract syntax.

Sess_Connect_Id	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the session connection. This argument is for future support.

Appli_Cont_Name	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter is an object identifier exchanged during the association establishment and subject to negotiation. The value is an array of integers (object Id integers).

Calling_AP_Title	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

The full name is Calling Application Process Title. This parameter is used to specify an application process on a given system.

Calling_AE_Qual	
Type:	longword
Access:	read only
Mechanism:	by reference

The full name is calling application entity qualifier. This parameter is used to specify an application entity within a given application process.

Called_AP_Title	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

The full name is Called Application Process Title. This parameter is used to specify an application process on a given system.

Called_AE_Qual	
Type:	longword
Access:	read only
Mechanism:	by reference

The full name is Called Application Entity Qualifier. This parameter is used to specify an application entity within a given application process.

NSAP_Type	
Type:	longword
Access:	read only
Mechanism:	by reference

This parameter defines the network protocol. This argument is for future support.

Template	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the transport template.

Protocol_Version	
Type:	unsigned char (byte) bit mask
Access:	read only
Mechanism:	by reference

This parameter defines the supported CMIP protocol version. Allowed values are:

CMIP_K_PROT_VER_1
CMIP_K_PROT_VER_2

CMISE_Fu	
Type:	unsigned char (byte) bit mask
Access:	read only
Mechanism:	by reference

This parameter defines the CMISE functional units.

Access_Control	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter is used to give optional access control information.

User_Info	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter may carry additional user information. Maximum size of data is 512 bytes for a session version 1, unlimited for session version 2.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

CMISE_S_PRSADD

13.1.2. Positive Response

M_Initialize_Accept

This function sends an M_INITIALIZE Positive response to the CMISE entity and accepts the association with the negotiated results.

Format

```
M_Initialize_Accept (Connection_Id,
                    [Responding_AP_Inv_Id], [Responding_AE_Inv_Id],
                     Pres_Cont_Def_Res_List, [Sess_Connect_Id],
                     Appli_Cont_Name, [Responding_AP_Title],
                     [Responding_AE_Qual],
                     Protocol_Version, CMISE_Fu, [Access_Control],
                     [User_Info])
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association accepted by this function.

Responding_AP_Inv_Id	
Type:	longword

Access:	read only
Mechanism:	by reference

This parameter identifies the responding application process invocation. Its value is in the range of 0 to 999998.

Responding_AE_Inv_Id	
Type:	longword
Access:	read only
Mechanism:	by reference

This parameter identifies the responding application entity invocation. Its value is in the range of 0 to 999998.

Pres_Cont_Def_Res_List	
Type:	cmise_pres_context
Access:	read only
Mechanism:	by reference

This parameter gives the result of the presentation context list negotiation.

Sess_Connect_Id	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the session connection. This argument is for future support.

Appli_Cont_Name	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter defines the application context name. The value is an array of integers (object Id integers).

Responding_AP_Title	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter defines the responding application process title.

Responding_AE_Qual	
Type:	longword

Access:	read only
Mechanism:	by reference

This parameter defines the responding application entity qualifier.

Protocol_Version	
Type:	unsigned char (byte) bit mask
Access:	read only
Mechanism:	by reference

This parameter defines the supported CMIP protocol version. Allowed values are:

CMIP_K_PROT_VER_1
CMIP_K_PROT_VER_2

CMISE_Fu	
Type:	unsigned char (byte) bit mask
Access:	read only
Mechanism:	by reference

This parameter defines the CMISE functional units accepted by the association responder.

Access_Control	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter gives access control information.

User_Info	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter may be used to pass additional user information. Maximum size of data is 512 bytes for a session version 1, unlimited for session version 2.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

13.1.3. M_INITIALIZE Negative Response

M_Initialize_Reject

This function sends a M_INITIALIZE Negative response to the CMISE entity and so, refuses an incoming connection.

Format

```
M_Initialize_Reject (Connection_Id,
                     [Responding_AP_Inv_Id], [Responding_AE_Inv_Id],
                      Pres_Cont_Def_Res_List, [Sess_Connect_Id],
                      Appli_Cont_Name, [Responding_AP_Title],
                     [Responding_AE_Qual],
                      Refuse_Reason, Protocol_Version, CMISE_Fu,
                     [Access_Control], [User_Info] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association rejected by this function.

Responding_AP_Inv_Id	
Type:	longword
Access:	read only
Mechanism:	by reference

This parameter identifies the responding application process invocation. Its value is in the range of 0 to 999998.

Responding_AE_Inv_Id	
Type:	longword
Access:	read only
Mechanism:	by reference

This parameter identifies the responding application entity invocation. Its value is in the range of 0 to 999998.

Pres_Cont_Def_Res_List	
Type:	cmise_pres_context
Access:	read only
Mechanism:	by reference

This parameter gives the result of the presentation context list negotiation.

Sess_Connect_Id	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the session connection. This argument is for future support.

Appli_Cont_Name	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter defines the application context name. The value is an array of integers (object Id integers).

Responding_AP_Title	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter defines the responding application process title.

Responding_AE_Qual	
Type:	longword
Access:	read only
Mechanism:	by reference

This parameter defines the responding application entity qualifier.

Protocol_Version	
Type:	unsigned char (byte) bit mask
Access:	read only
Mechanism:	by reference

This parameter defines the supported CMIP protocol version. Allowed values are:

CMIP_K_PROT_VER_1
CMIP_K_PROT_VER_2

Refuse_Reason	
Type:	longword
Access:	read only
Mechanism:	by reference

This parameter specifies the association refuse reason. Allowed values are:

M_RA_NULL

M_RU_NULL
M_RP_USER_REJ
M_RA_REASON_UNK
M_RU_REASON_UNK
M_RP_LOC_LIMIT_EXEED
M_RA_NO_COMM_VERS
M_RU_ACN_NOT_SUPP
M_RP_NO_DEFAULT
M_RU_BAD_AP_TITL_G
M_RP_USER_DATA_UNREAD
M_RU_BAD_AP_INV_ID_G
M_RS_NO_SUCH_SSAP
M_RU_BAD_AE_QUAL_G
M_RS_NO_USER
M_RU_BAD_AE_INV_ID_G
M_RS_CONGESTED
M_RU_BAD_AP_TITL_D
M_RS_UNSUPPORTED
M_RU_BAD_AP_INV_ID_D
M_RS_REFUSED
M_RU_BAD_AE_QUAL_D
M_RU_BAD_AE_INV_ID_D

CMISE_Fu	
Type:	unsigned char (byte) bit mask
Access:	read only
Mechanism:	by reference

This parameter defines the CMISE functional units accepted by the association responder.

Access_Control	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter gives access control information.

User_Info	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter may be used to pass additional user information. Maximum size of data is 512 bytes for a session version 1, unlimited for session version 2.

Return Values

CMISE_S_NORMAL
CMISE_S_ASNERR
CMISE_S_INVPAR

CMISE_S_IPCERR
CMISE_S_MISPAR
CMISE_S_MEMERR

13.1.4. M_INITIALIZE Indication

M_Initialize_Ind

This function is used to decode a M_INITIALIZE Indication (event M_E_INITIALIZE_IND) sent by the CMISE entity.

Format

```
M_Initialize_Ind (Connection_Id,
                  Called_PSEL, Called_SSEL, Called_TSEL, Called_NSAP,
                 [Calling_PSEL], [Calling_SSEL], [Calling_TSEL],
                 [Calling_NSAP],
                 [Calling_AP_Inv_Id], [Calling_AE_Inv_Id],
                 [Called_AP_Inv_Id], [Called_AE_Inv_Id],
                  Pres_Cont_Def_List, [Sess_Connect_Id], Appli_Cont_Name,
                 [Calling_AP_Title], [Calling_AE_Qual],
                 [Called_AP_Title], [Called_AE_Qual],
                 [NSAP_Type], Protocol_Version,
                  CMISE_Fu, [Access_Control], [User_Info])
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Called_PSEL	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the called presentation selector.

Called_SSEL	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the called session selector.

Called_TSEL	
Type:	cmise_buffer

Access:	read only
Mechanism:	by reference

This parameter defines the called transport selector.

Called_NSAP	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the called network service access point.

Calling_PSEL	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the calling presentation selector.

Calling_SSEL	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the calling session selector.

Calling_TSEL	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the calling transport selector.

Calling_NSAP	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the calling network service access point.

Calling_AP_Inv_Id	
Type :	longword
Access:	write only
Mechanism:	by reference

This parameter identifies the calling application process invocation. Its value is in the range of 0 to 999998.

Calling_AE_Inv_Id	
Type :	longword
Access:	write only
Mechanism:	by reference

This parameter identifies the calling application entity invocation. Its value is in the range of 0 to 999998.

Called_AP_Inv_Id	
Type :	longword
Access:	write only
Mechanism:	by reference

This parameter identifies the called application process invocation. Its value is in the range of 0 to 999998.

Called_AE_Inv_Id	
Type:	longword
Access:	write only
Mechanism:	by reference

This parameter identifies the called application entity invocation. Its value is in the range of 0 to 999998.

Pres_Cont_Def_List	
Type :	cmise_pres_context
Access:	write only
Mechanism:	by reference

The full name is Presentation Context Definition List. This parameter is used to specify the list of the presentation contexts the user wants to be negotiated.

Note

Currently only P_K_BER_ASN1 is allowed for transfer syntax. Currently only P_K_CMISE and P_K_ACSE are allowed for abstract syntax.

Sess_Connect_Id	
Type :	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter defines the session connection. This argument is for future support.

Appli_Cont_Name	
Type:	cmise_oid
Access:	write only

Mechanism:	by reference

This parameter is an object identifier exchanged during the association establishment and subject to negotiation. The value is an array of integers (object Id integers).

Calling_AP_Title	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

The full name is Calling Application Process Title. This parameter is used to specify an application process on a given system.

Calling_AE_Qual	
Type :	longword
Access:	write only
Mechanism:	by reference

The full name is Calling Application Entity Qualifier. This parameter is used to specify an application entity within a given application process.

Called_AP_Title	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

The full name is Called Application Process Title. This parameter is used to specify an application process on a given system.

Called_AE_Qual	
Type:	longword
Access:	write only
Mechanism:	by reference

The full name is Called Application Entity Qualifier. This parameter is used to specify an application entity within a given application process.

NSAP_Type	
Type:	longword
Access:	read only
Mechanism:	by reference

This parameter defines the network protocol. This argument is for future support.

Protocol_Version	
Type:	unsigned char (byte) bit mask
Access:	read only

Mechanism:	by reference

This parameter defines the supported CMIP protocol version. Allowed values are:

CMIP_K_PROT_VER_1
CMIP_K_PROT_VER_2

CMISE_Fu	
Type:	unsigned char (byte) bit mask
Access:	write only
Mechanism:	by reference

This parameter defines the CMISE functional units.

Access_Control	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter is used to give optional access control information.

User_Info	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter may carry additional user information. Maximum size of data is 512 bytes for a session version 1, unlimited for session version 2.

Return Values

CMISE_S_NORMAL
CMISE_S_ASNERR
CMISE_S_INVPAR
CMISE_S_IPCERR
CMISE_S_MEMERR

13.1.5. M_INITIALIZE Positive Confirm

M_Initialize_Acc

This function is used to decode a M_INITIALIZE Positive confirmation (event M_E_INITIALIZE_ACC) sent by the CMISE entity.

Format

```
M_Initialize_Acc (Connection_Id,
                  [Responding_AP_Inv_Id], [Responding_AE_Inv_Id],
                   Pres_Cont_Def_Res_List, [Sess_Connect_Id],
                   Appli_Cont_Name, [Responding_AP_Title],
```

```
[Responding_AE_Qual],
 Protocol_Version, CMISE_Fu, [Access_Control],
[User_Info] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Responding_AP_Inv_Id	
Type:	longword
Access:	write only
Mechanism:	by reference

This parameter identifies the responding application process invocation. Its value is in the range of 0 to 999998.

Responding_AE_Inv_Id	
Type:	longword
Access:	write only
Mechanism:	by reference

This parameter identifies the responding application entity invocation. Its value is in the range of 0 to 999998.

Pres_Cont_Def_Res_List	
Type:	cmise_pres_context
Access:	write only
Mechanism:	by reference

This parameter gives the result of the presentation context list negotiation.

Sess_Connect_Id	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter defines the session connection. This argument is for future support.

Appli_Cont_Name	
Type:	cmise_oid
Access:	write only

Mechanism:	by reference

This parameter defines the application context name. The value is an array of integers (object Id integers).

Responding_AP_Title	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter defines the responding application process title.

Responding_AE_Qual	
Type:	longword
Access:	write only
Mechanism:	by reference

This parameter defines the responding application entity qualifier.

Protocol_Version	
Type:	unsigned char (byte) bit mask
Access:	read only
Mechanism:	by reference

This parameter defines the supported CMIP protocol version. Allowed values are:

CMIP_K_PROT_VER_1
CMIP_K_PROT_VER_2

CMISE_Fu	
Type:	unsigned char (byte) bit mask
Access:	write only
Mechanism:	by reference

This parameter defines the CMISE functional units accepted by the association responder.

Access_Control	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter gives access control information.

User_Info	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter may be used to pass additional user information. Maximum size of data is 512 bytes for a session version 1, unlimited for session version 2.

Return Values

CMISE_S_NORMAL
CMISE_S_ASNERR
CMISE_S_INVPAR
CMISE_S_IPCERR
CMISE_S_MEMERR

13.1.6. M_INITIALIZE Negative Confirm

M_Initialize_Rej

This function is used to decode a M_INITIALIZE Negative confirmation (event M_E_INITIALIZE_REJ) sent by the CMISE entity.

Format

```
M_Initialize_Rej (Connection_Id,
                  [Responding_AP_Inv_Id], [Responding_AE_Inv_Id],
                   Pres_Cont_Def_Res_List, [Sess_Connect_Id],
                   Appli_Cont_Name, [Responding_AP_Title],
                  [Responding_AE_Qual], Refuse_Reason,
                   Protocol_Version, CMISE_Fu, [Access_Control],
                  [User_Info] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Responding_AP_Inv_Id	
Type:	longword
Access:	write only
Mechanism:	by reference

This parameter identifies the responding application process invocation. Its value is in the range of 0 to 999998.

Responding_AE_Inv_Id	
Type:	longword
Access:	write only
Mechanism:	by reference

This parameter identifies the responding application entity invocation. Its value is in the range of 0 to 999998.

Pres_Cont_Def_Res_List	
Type:	cmise_pres_context
Access:	write only
Mechanism:	by reference

This parameter gives the result of the presentation context list negotiation.

Sess_Connect_Id	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter defines the session connection. This argument is for future support.

Appli_Cont_Name	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter defines the application context name. The value is an array of integers (object Id integers).

Responding_AP_Title	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter defines the responding application process title.

Responding_AE_Qual	
Type:	longword
Access:	write only
Mechanism:	by reference

This parameter defines the responding application entity qualifier.

Refuse_Reason	
Type:	longword
Access:	write only
Mechanism:	by reference

This parameter specifies the association refuse reason. Allowed values are:

M_RA_NULL

M_RU_NULL
M_RP_USER_REJ
M_RA_REASON_UNK
M_RU_REASON_UNK
M_RP_LOC_LIMIT_EXEED
M_RA_NO_COMM_VERS
M_RU_ACN_NOT_SUPP
M_RP_NO_DEFAULT
M_RU_BAD_AP_TITL_G
M_RP_USER_DATA_UNREAD
M_RU_BAD_AP_INV_ID_G
M_RS_NO_SUCH_SSAP
M_RU_BAD_AE_QUAL_G
M_RS_NO_USER
M_RU_BAD_AE_INV_ID_G
M_RS_CONGESTED
M_RU_BAD_AP_TITL_D
M_RS_UNSUPPORTED
M_RU_BAD_AP_INV_ID_D
M_RS_REFUSED
M_RU_BAD_AE_QUAL_D
M_RU_BAD_AE_INV_ID_D

Protocol_Version	
Type:	unsigned char (byte) bit mask
Access:	read only
Mechanism:	by reference

This parameter defines the supported CMIP protocol version. Allowed values are:

CMIP_K_PROT_VER_1
CMIP_K_PROT_VER_2

CMISE_Fu	
Type:	unsigned char (byte) bit mask
Access:	write only
Mechanism:	by reference

This parameter defines the CMISE functional units accepted by the association responder.

Access_Control	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter gives access control information.

User_Info	
Type:	cmise_buffer

Access:	write only
Mechanism:	by reference

This parameter may be used to pass additional user information. Maximum size of data is 512 bytes for a session version 1, unlimited for session version 2.

Return Values

CMISE_S_NORMAL
CMISE_S_ASNERR
CMISE_S_INVPAR
CMISE_S_IPCERR
CMISE_S_MEMERR
CMISE_S_MISPAR

13.2. M_TERMINATE Service

The Terminate service is used by a CMISE user to request the orderly termination of an association with a peer CMISE user. This service is always confirmed.

There are two sets of Terminate response/confirmation. One is used to answer to a normal association release, the other is used when a release collision occurs. A release collision may occur in one of the following cases:

- After having received a RLRQ PDU and passed to the user an M_E_TERMINATE_IND event, the CMISE entity received an M_E_TERMINATE_REQ event from the user.
- After having processed an M_E_TERMINATE_REQ event from the user and sent a TERMINATE_REQ PDU, the CMISE entity receives a TERMINATE_REQ PDU.

If a release collision occurs, the CMISE user must act as follows:

- On the association initiator side, the user has to issue a non-terminal terminate response, then wait for the terminal terminate confirmation (event M_E_TERMINATE_ACC).
- On the association responder side, the user has to wait for a non-terminal release confirmation (event M_E_TERMINATE_REJ), then issue a terminal terminate response.

13.2.1. M_TERMINATE Request

M_Terminate_Req

This function sends an M_TERMINATE Request to the CMISE entity to orderly terminate an association with a peer entity.

Format

```
M_Terminate_Req (Connection_Id, Urgency, [User_Info] )
```

Arguments

Connection_Id

Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Urgency	
Type:	unsigned char (byte)
Access:	read only
Mechanism:	by reference

This parameter specifies the urgency of the request. Allowed values are:

M_UR_NORMAL
M_UR_URGENT
M_UR_USER_DEF

User_Info	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter gives additional user information. Maximum size of data is 512 bytes for a session version 1, unlimited for session version 2.

Return Values

CMISE_S_NORMAL
CMISE_S_IPCERR

13.2.2. M_TERMINATE Positive Response

M_Terminate_Accept

This function sends an M_TERMINATE Positive response to the CMISE entity and so accepts the release of an existing association with a peer entity.

Format

```
M_Terminate_Accept (Connection_Id, [Reason], [User_Info] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Reason	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter specifies the user-specified release reason.

User_Info	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter gives additional user information. Maximum size of data is 512 bytes for a session version 1, unlimited for session version 2.

Return Values

CMISE_S_NORMAL
CMISE_S_IPCERR

13.2.3. M_TERMINATE Negative Response

M_Terminate_Reject

This function sends an M_TERMINATE Negative response to the CMISE entity, and so refuses the release of an existing association with a peer entity (because of a release collision).

Format

```
M_Terminate_Reject (Connection_Id, [Reason], [User_Info] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Reason	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter specifies the user-specified reject reason.

User_Info

Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies additional user information. Maximum size of data is 512 bytes for a session version 1, unlimited for session version 2.

Note

Negotiated release is not currently supported.

Return Values

CMISE_S_NORMAL
CMISE_S_IPCERR

13.2.4. M_TERMINATE Indication

M_Terminate_Ind

This function is used to decode an M_TERMINATE Indication (event M_E_TERMINATE_IND) sent by the CMISE entity.

Format

```
M_Terminate_Ind (Connection_Id, Urgency, [User_Info] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Urgency	
Type:	unsigned char (byte)
Access:	write only
Mechanism:	by reference

This parameter specifies the urgency of the request. One of the following values will be returned:

M_UR_NORMAL
M_UR_URGENT
M_UR_USER_DEF

User_Info	
Type:	cmise_buffer

Access:	write only
Mechanism:	by reference

This parameter gives additional user information. Maximum size of data is 512 bytes for a session version 1, unlimited for session version 2.

Return Values

CMISE_S_NORMAL
CMISE_S_MEMERR

13.2.5. M_TERMINATE Positive Confirm

M_Terminate_Acc

This function is used to decode a M_TERMINATE Positive response (event M_E_TERMINATE_ACC) sent by the CMISE entity.

Format

```
M_Terminate_Acc (Connection_Id, Reason, [User_Info] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Reason	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter specifies the user-specified release reason.

User_Info	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter gives additional user information. Maximum size of data is 512 bytes for a session version 1, unlimited for session version 2.

Return Values

CMISE_S_NORMAL

CMISE_S_IPCERR
CMISE_S_MEMERR

13.2.6. M_TERMINATE Negative Confirm

M_Terminate_Rej

This function is used to decode a M_TERMINATE Negative response (event M_E_TERMINATE_REJ) sent by the CMISE entity.

Format

```
M_Terminate_Rej (Connection_Id, Reason, [User_Info] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Reason	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter specifies the user-specified reject reason.

User_Info	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter gives additional user information. Maximum size of data is 512 bytes for a session version 1, unlimited for session version 2.

Return Values

CMISE_S_NORMAL
CMISE_S_MEMERR

13.3. M_U_ABORT Service

This service is used to request the abrupt termination of an association. An abort may be initiated by:

- CMISE user, invoking the M_U_Abort_Req primitive

- Any entity below CMISE, sending up an M_E_U_ABORT_IND (Application layer abort) or M_E_P_ABORT_IND (communication service provider abort) event to the user

After the invocation (user or provider) of the Abort service, the association can no longer be used.

13.3.1. M_U_ABORT Request

M_U_Abort_Req

This function sends a M_ABORT Request to the CMISE entity to request the abrupt termination of an association.

Format

```
M_U_Abort_Req (Connection_Id, [Abort_Context_Id_List], [User_Info])
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Abort_Context_Id_List	
Type:	cmise_abort_context
Access:	read only
Mechanism:	by reference

This parameter specifies the abort contexts identifiers used to decode the `User_Info` argument content. If no user information is provided, this argument is optional. It is not significant for the session version 1 and the maximum size is unlimited for a session version 2.

User_Info	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter gives additional user information. If any User_Info is given, the Abort_Context_Id_List must be present also. It is not significant for the session version 1 and the maximum size is unlimited for a session version 2.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

13.3.2. M_ABORT Indication

M_U_Abort_Ind

This function is used to decode a M_ABORT Indication (event M_E_U_ABORT_IND) sent by the CMISE entity.

Format

```
M_U_Abort_Ind (Connection_Id, Abort_Context_Id_List, User_Info )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Abort_Context_Id_List	
Type:	cmise_abort_context
Access:	write only
Mechanism:	by reference

This parameter specifies the abort contexts identifiers used to decode the User_Info argument content. If no user information is provided, this argument is optional. It is not significant for the session version 1 and the maximum size is unlimited for a session version 2.

User_Info	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter gives additional user information. If any User_Info is given, the Abort_Context_Id_List must be present also. It is not significant for the session version 1 and the maximum size is unlimited for a session version 2.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

13.3.3. M_P_ABORT Indication

M_P_Abort_Ind

This function is used to decode a M_ABORT Indication (event M_E_P_ABORT_IND) sent by an underlying entity.

Format

```
M_E_P_Abort_Ind (Connection_Id, Source_Reason)
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Source_Reason	
Type:	unsigned char (byte)
Access:	write only
Mechanism:	by reference

This parameter is used to specify where the abort has been issued.

Return Values

CMISE_S_NORMAL

CMISE_S_IPCERR

13.4. M_EVENT_REPORT Service

The Event Report service is used by a CMISE user to report an event to a peer network management application. This service may be confirmed or not.

Since no multiple object selection is allowed, the linked reply mechanism does not apply.

13.4.1. M_EVENT_REPORT Request

M_Event_Req

This function sends a M_Event_Report Request to the CMISE entity. It is used when a CMISE user wants to report an event to a peer network management application.

Format

```
M_Event_Req (Connection_Id, Invoke_Id, Flags,
             Object_Class, Object_Instance, [Time_Of_Event], Event_Type,
            [Event_Info] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier.

Flags	
Type:	cmise_flag
Access:	read only
Mechanism:	by reference

The only significant bit flag is M_FG_CONFIRMED. This flag is set if a confirmed event report is requested.

Object_Class	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter specifies the class of the managed object in which the event occurred.

Object_Instance	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the instance of the managed object in which the event occurred.

Time_Of_Event	
Type:	cmise_buffer
Access:	read only

Mechanism:	by reference

This parameter specifies the time the event was generated.

Event_Type	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter specifies the type of event being reported.

Event_Info	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies additional optional information about the event.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

13.4.2. M_EVENT_REPORT Indication

M_Event_Ind

This function is used to decode a M_EVENT_REPORT Indication (event M_E_EVT_REPORT_IND) sent by the CMISE entity.

Format

```
M_Event_Ind (Connection_Id, Invoke_Id, Flags,
             Object_Class, Object_Instance, Time_Of_Event, Event_Type,
             Event_Info )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier.

Flags	
Type:	cmise_flag
Access:	write only
Mechanism:	by reference

The only significant bit flag is M_FG_CONFIRMED. This flag is set if a confirmed event report is requested.

Object_Class	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter specifies the class of the managed object in which the event occurred.

Object_Instance	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the instance of the managed object in which the event occurred.

Time_Of_Event	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the time the event was generated.

Event_Type	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter specifies the type of event being reported.

Event_Info

Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies additional optional information about the event.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_ILLPAR

CMISE_S_INVPAR

CMISE_S_INVREQ

CMISE_S_MEMERR

CMISE_S_MISPAR

13.4.3. M_EVENT_REPORT Response

M_Event_Resp

This function sends an M_Event_Report Response to the CMISE entity. It is used when an application receives a confirm Event_Report Indication and then must acknowledge the report with an Event_Report Response.

Format

```
M_Event_Resp (Connection_Id, Invoke_Id, Flags,
              [Object_Class], [Object_Instance], [Time_Of_Response],
              [Event_Type], [Event_Reply_Info] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier.

Flags	
Type:	cmise_flag
Access:	read only
Mechanism:	by reference

The significant bit flags are M_FG_CONFIRMED and M_FG_RORS_BASIC. M_FG_CONFIRMED is set if a confirmed event report is requested. M_FG_RORS_BASIC is set if the response contains the Invoke_Id only.

Object_Class	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter specifies the class of the managed object in which the event occurred.

Object_Instance	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the instance of the managed object in which the event occurred.

Time_Of_Response	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the time the event was generated.

Event_Type	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter specifies the type of event being reported.

Event_Reply_Info	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies additional optional information about the event.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

13.4.4. M_EVENT_REPORT Confirm

M_Event_Cnf

This function is used to decode a M_EVENT_REPORT Confirm (event M_E_EVT_REPORT_CNF) sent by the CMISE entity.

Format

```
M_Event_Cnf (Connection_Id, Invoke_Id, Flags,
             Object_Class, Object_Instance, Time_Of_Response, Event_Type,
             Event_Reply_Info )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier.

Flags	
Type:	cmise_flag
Access:	write only
Mechanism:	by reference

The only significant bit flag is M_FG_CONFIRMED. This flag is set if a confirmed event report is requested.

Object_Class	
Type:	cmise_oid
Access:	write only

Mechanism:	by reference

This parameter specifies the class of the managed object in which the event occurred.

Object_Instance	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the instance of the managed object in which the event occurred.

Time_Of_Response	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the time the event was generated.

Event_Type	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter specifies the type of event being reported.

Event_Reply_Info	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies additional optional information about the event.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_INVREQ

CMISE_S_MEMERR

13.5. M_GET Service

The Get service is used by a CMISE user to retrieve attribute values from one or several managed object(s) located on a remote system. This service is always confirmed.

Since multiple object selection is allowed, the linked reply mechanism applies.

13.5.1. M_GET Request

M_Get_Req

This function sends an M_GET Request to the CMISE entity. It is used by a CMISE user to retrieve attributes values from one or several managed object(s) located on a remote system.

Format

```
M_Get_Req (Connection_Id, Invoke_Id, Flags, Object_Class,
           Object_Instance, [ Access_Control ], [ Scope ], [ Filter ],
         [ Attr_Id_List ] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier.

Flags	
Type:	cmise_flag
Access:	read only
Mechanism:	by reference

The significant bit flags are M_FG_NOSYNC and M_FG_ATOMIC_SYNC.

M_FG_NOSYNC is set if no synchronization is requested. M_FG_ATOMIC_SYNC is set if atomic execution is required.

Object_Class	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter specifies the class of the managed object to which the filter (when supplied) is to be applied. If no filter is supplied, then this parameter specifies the class of the object to be managed.

Object_Instance	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the instance of the managed object to which the filter (when supplied) is to be applied. If no filter is supplied, then this parameter specifies the instance of the object to be managed.

Access_Control	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies optional access control information.

Scope	
Type:	cmise_scope
Access:	read only
Mechanism:	by reference

This parameter specifies the part of the containment tree involved in a network management service invocation.

Filter	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the filter used to select managed objects.

Attr_Id_List	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the attribute identifiers list whose values are requested. It may be empty, which means all attributes will be returned.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

13.5.2. M_GET Indication

M_Get_Ind

This function is used to decode a M_GET Indication (event M_E_EVT_REPORT_IND) sent by the CMISE entity.

Format

```
M_Get_Ind (Connection_Id, Invoke_Id, Flags, Object_Class,
           Object_Instance, Access_Control, Scope, Filter, Attr_Id_List)
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier.

Flags	
Type:	cmise_flag
Access:	write only
Mechanism:	by reference

The only significant bit flag is M_FG_ATOMIC_SYNC. This flag is set if atomic execution is required.

Object_Class	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter specifies the class of the managed object to which the filter (when supplied) is to be applied. If no filter is supplied, then this parameter specifies the class of the object to be managed.

Object_Instance	
Type:	cmise_buffer

Access:	write only
Mechanism:	by reference

This parameter specifies the instance of the managed object to which the filter (when supplied) is to be applied. If no filter is supplied, then this parameter specifies the instance of the object to be managed.

Access_Control	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies optional access control information.

Scope	
Type:	cmise_scope
Access:	write only
Mechanism:	by reference

This parameter specifies the part of the containment tree involved in a network management service invocation.

Filter	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the filter used to select managed objects.

Attr_Id_List	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the attribute identifiers list whose values are requested. It may be empty, which means all attributes will be returned.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_ILLPAR

CMISE_S_INVPAR

CMISE_S_INVREQ

CMISE_S_MEMERR

CMISE_S_MISPAR

13.5.3. M_GET Response

M_Get_Resp

This function sends an M_GET Response to the CMISE entity. It is used when an application receives a Get indication and then must send a Get response for each selected object on which execution is successful.

Format

```
M_Get_Resp (Connection_Id, Invoke_Id, Flags, Linked_Id,
            [Object_Class ], [ Object_Instance] , [ Response_Time ],
            [Attribute_List] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

If the response is a linked reply, the value is a new invoke identifier local to the response. Otherwise, it is the Get operation invoke identifier.

Flags	
Type:	cmise_flag
Access:	read only
Mechanism:	by reference

The only significant bit flags are M_FG_LINKED_ID and M_FG_RORS_BASIC.M-FG_LINKED_ID is set if the response is a linked reply, and M_FG_RORS_BASIC is set if the response contains the Invoke_Id only.

Linked_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter is meaningful if the linked identifier flag is set. It gives the Get operation invoke identifier if the response is a linked reply.

Object_Class	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter defines the managed object class. This argument is mandatory if the request has selected several classes.

Object_Instance	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the managed object instance. This argument is mandatory if the request has selected several objects.

Response_Time	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the time the response was generated.

Attribute_List	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter contains the set of attribute identifiers and values that are returned.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

13.5.4. M_GET Confirm

M_Get_Cnf

This function is used to decode a M_GET Confirmation (event M_E_GET_CNF) sent by the CMISE entity.

Format

```
M_Get_Cnf (Connection_Id, Invoke_Id, Flags, Linked_Id,
           Object_Class, Object_Instance, Response_Time, Attribute_List )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

If the response is a linked reply, the value is a new invoke identifier local to the response. Otherwise, it is the Get operation invoke identifier.

Flags	
Type:	cmise_flag
Access:	write only
Mechanism:	by reference

The only significant bit flag is M_FG_LINKED_ID. This flag is set if the response is a linked reply.

Linked_Id	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter is meaningful if the linked identifier flag is set. It gives the Get operation invoke identifier if the response is a linked reply.

Object_Class	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter defines the managed object class. This argument is mandatory if the request has selected several classes.

Object_Instance

Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter defines the managed object instance. This argument is mandatory if the request has selected several objects.

Response_Time	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the time the response was generated.

Attribute_List	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter contains the set of attribute identifiers and values that are returned.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_INVREQ

CMISE_S_MEMERR

13.6. M_CANCEL_GET Service

The Cancel_Get service is used by a CMISE user to request a network management peer to cancel a previous Get service invocation. This service is always confirmed. Since there is no multiple object selection, the linked reply mechanism does not apply.

13.6.1. M_CANCEL_GET Request

M_Cancel_Get_Req

This function sends a M_CANCEL_GET Request to the CMISE entity. It is used by a CMISE user to request a network management peer to cancel a previous Get service invocation. This service is always confirmed.

Format

```
M_Cancel_Get_Req (Connection_Id, Invoke_Id, Cancel_Id)
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier of the Cancel_Get operation.

Cancel_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier of the Get operation to be canceled.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

13.6.2. M_CANCEL_GET Indication

M_Cancel_Get_Ind

This function is used to decode a M_CANCEL_GET Indication (event M_E_CANCEL_GET_IND) sent by the CMISE entity.

Format

```
M_Cancel_Get_Ind ( Connection_Id, Invoke_Id, Cancel_Id )
```

Arguments

Connection_Id

Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter defines the invoke identifier of the Cancel_Get operation.

Cancel_Id	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter defines the invoke identifier of the Get operation to be canceled.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_ILLPAR

CMISE_S_INVPAR

CMISE_S_INVREQ

CMISE_S_MEMERR

CMISE_S_MISPAR

13.6.3. M_CANCEL_GET Response

M_Cancel_Get_Resp

This function sends a M_CANCEL_GET Response to the CMISE entity. It is used when an application receives a Cancel_Get indication and must acknowledge the operation with a Cancel_Get response.

Format

```
M_Cancel_Get_Resp (Connection_Id, Invoke_Id )
```

Arguments

Connection_Id

Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier of the Cancel_Get operation.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

13.6.4. M_CANCEL_GET Confirm

M_Cancel_Get_Cnf

This function is used to decode a M_CANCEL_GET Confirmation (event M_E_CANCEL_GET_CNF) sent by the CMISE entity.

Format

```
M_Cancel_Get_Cnf ( Connection_Id, Invoke_Id )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter defines the invoke identifier of the Cancel_Get operation.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_INVREQ

CMISE_S_MEMERR

13.7. M_SET Service

The SET service is used by a CMISE user to set attribute values of one or several managed object(s) located on a remote system. This service may be confirmed or not.

Since multiple object selection is allowed, the linked reply mechanism applies.

13.7.1. M_SET Request

M_Set_Req

This function sends an M_SET Request to the CMISE entity. It is used by a CMISE user to set attribute values of one or several managed object(s) located on a remote system.

Format

```
M_Set_Req (Connection_Id, Invoke_Id, Flags, Object_Class,
           Object_Instance, [ Access_Control ], [ Scope ], [ Filter ],
           Attr_Set_List )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier.

Flags

Type:	cmise_flag
Access:	read only
Mechanism:	by reference

The significant bit flags are M_FG_CONFIRMED and M_FG_ATOMIC_SYNC.

M_FG_CONFIRMED is set if a confirmed action is requested. M_FG_ATOMIC_SYNC is set if atomic execution is required.

Object_Class	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter specifies the class of the managed object to which the filter (when supplied) is to be applied. If no filter is supplied, then this parameter specifies the class of the object to be managed.

Object_Instance	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the instance of the managed object to which the filter (when supplied) is to be applied. If no filter is supplied, then this parameter specifies the instance of the object to be managed.

Access_Control	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies optional access control information.

Scope	
Type:	cmise_scope
Access:	read only
Mechanism:	by reference

This parameter specifies the part of the containment tree involved in a network management service invocation.

Filter	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the filter used to select managed objects.

Attr_Set_List	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the attribute identifiers list whose values are to be modified.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

13.7.2. M_SET Indication

M_Set_Ind

This function is used to decode a M_SET Indication (event M_E_SET_IND) sent by the CMISE entity.

Format

```
M_Set_Ind (Connection_Id, Invoke_Id, Flags, Object_Class,
           Object_Instance, Access_Control, Scope, Filter, Attr_Set_List )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier.

Flags	
Type:	cmise_flag
Access:	write only

Mechanism:	by reference

The significant bit flags are M_FG_CONFIRMED and M_FG_ATOMIC_SYNC. M_FG_CONFIRMED is set if a confirmed set is requested. M_FG_ATOMIC_SYNC is set if atomic execution is required.

Object_Class	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter specifies the class of the managed object to which the filter (when supplied) is to be applied. If no filter is supplied, then this parameter specifies the class of the object to be managed.

Object_Instance	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the instance of the managed object to which the filter (when supplied) is to be applied. If no filter is supplied, then this parameter specifies the instance of the object to be managed.

Access_Control	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies optional access control information.

Scope	
Type:	cmise_scope
Access:	write only
Mechanism:	by reference

This parameter specifies the part of the containment tree involved in a network management service invocation.

Filter	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the filter used to select managed objects.

Attr_Set_List	
Type:	cmise_buffer

Access:	write only
Mechanism:	by reference

This parameter specifies the attribute identifiers list whose values are to be modified.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_ILLPAR

CMISE_S_INVPAR

CMISE_S_INVREQ

CMISE_S_MEMERR

CMISE_S_MISPAR

13.7.3. M_SET Response

M_Set_Resp

This function sends an M_SET Response to the CMISE entity. It is used when an application receives a set indication and then must send a set response for each selected object on which execution is successful.

Format

```
M_Set_Resp (Connection_Id, Invoke_Id, Flags, Linked_Id,
            [Object_Class ], [ Object_Instance] , [ Response_Time ],
            [Attribute_List] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier. If the response is a linked reply the value is a new invoke identifier local to this response. Otherwise, it is the Set operation invoke identifier.

Flags	
Type:	cmise_flag
Access:	read only
Mechanism:	by reference

The significant bit flags are M_FG_LINKED_ID and M_FG_RORS_BASIC. M_FG_LINKED_ID is set if the response is a linked reply. M_FG_RORS_BASIC is set if the response contains the Invoke_Id only.

Linked_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the linked identifier. This parameter is meaningful if the linked identifier flag is set. It gives the Set operation identifier if the response is a linked reply.

Object_Class	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter defines the managed object class. This argument is mandatory if the request has selected several classes.

Object_Instance	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the managed object instance. It must be provided if the Set request has selected several objects.

Response_Time	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the time the response was generated.

Attribute_List	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter contains the set of attribute identifiers and values that are modified.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

13.7.4. M_SET Confirm

M_Set_Cnf

This function is used to decode a M_SET Confirmation (event M_E_SET_CNF) sent by the CMISE entity.

Format

```
M_Set_Cnf (Connection_Id, Invoke_Id, Flags, Linked_Id,
           Object_Class, Object_Instance, Response_Time, Attribute_List )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	write only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter defines the invoke identifier. If the response is a linked reply the value is a new invoke identifier local to this response. Otherwise, it is the Set operation invoke identifier.

Flags	
Type:	cmise_flag
Access:	read only
Mechanism:	by reference

The only significant bit flag is M_FG_LINKED_ID. This flag is set if the response is a linked reply.

Linked_Id

Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the linked identifier. This parameter is meaningful if the linked identifier flag is set. It gives the Set operation identifier if the response is a linked reply.

Object_Class	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter defines the managed object class. This argument is mandatory if the request has selected several classes.

Object_Instance	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the managed object instance. It must be provided if the Set request has selected several objects.

Response_Time	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the time the response was generated.

Attribute_List	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter contains the set of attribute identifiers and values that are modified.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_INVREQ

CMISE_S_MEMERR

13.8. M_ACTION Service

The ACTION service is used by a CMISE user to request a peer system to perform an action on one or several managed object(s). This service may be confirmed or not. Since multiple object selection is allowed, the linked reply mechanism applies.

13.8.1. M_ACTION Request

M_Action_Req

This function sends an M_ACTION Request to the CMISE entity. It is used by a CMISE user to request a peer system to perform an action on one or several managed object(s).

Format

```
M_Action_Req (Connection_Id, Invoke_Id, Flags, Object_Class,
              Object_Instance, [ Access_Control ], Scope, Filter,
              Action_Type, [Action_Info] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier.

Flags	
Type:	cmise_flag
Access:	read only
Mechanism:	by reference

The significant bit flags are M_FG_CONFIRMED and M_FG_ATOMIC_SYNC. M_FG_CONFIRMED is set if a confirmed action is requested. M_FG_ATOMIC_SYNC is set if an atomic execution is required.

Object_Class	
Type:	cmise_oid
Access:	read only

Mechanism:	by reference

This parameter specifies the class of the managed object to which the filter (when supplied) is to be applied. If no filter is supplied, then this parameter specifies the class of the object to be managed.

Object_Instance	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the instance of the managed object to which the filter (when supplied) is to be applied. If no filter is supplied, then this parameter specifies the instance of the object to be managed.

Access_Control	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies optional access control information.

Scope	
Type:	cmise_scope
Access:	read only
Mechanism:	by reference

This parameter specifies the part of the containment tree involved in a network management service invocation.

Filter	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the filter used to select managed objects.

Action_Type	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter specifies the specific action to be performed.

Action_Info	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies extra information, if necessary, to further define the action to be performed.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

13.8.2. M_ACTION Indication

M_Action_Ind

This function is used to decode a M_ACTION Indication (event M_E_ACTION_IND) sent by the CMISE entity.

Format

```
M_Action_Ind (Connection_Id, Invoke_Id, Flags, Object_Class,
              Object_Instance, Access_Control, [ Scope ], [ Filter ],
              Action_Type, Action_Info )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter specifies the class of the managed object to which the filter (when supplied) is to be applied. If no filter is supplied, then this parameter specifies the class of the object to be managed.

Flags	
Type:	cmise_flag
Access:	write only
Mechanism:	by reference

The significant bit flags are M_FG_CONFIRMED and M_FG_ATOMIC_SYNC.

M_FG_CONFIRMED is set if a confirmed action is requested. M_FG_ATOMIC_SYNC is set if atomic execution is required.

Object_Class	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter specifies the class of the managed object to which the filter (when supplied) is to be applied. If no filter is supplied, then this parameter specifies the class of the object to be managed.

Object_Instance	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the instance of the managed object to which the filter (when supplied) is to be applied. If no filter is supplied, then this parameter specifies the instance of the object to be managed.

Access_Control	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies optional access control information.

Scope	
Type:	cmise_scope
Access:	write only
Mechanism:	by reference

This parameter specifies the part of the containment tree involved in a network management service invocation.

Filter	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the filter used to select managed objects.

Action_Type	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter specifies the specific action to be performed.

Action_Info	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies extra information, if necessary, to further define the action to be performed.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_ILLPAR

CMISE_S_INVPAR

CMISE_S_INVREQ

CMISE_S_MEMERR

CMISE_S_MISPAR

13.8.3. M_ACTION Response

M_Action_Resp

This function sends a M_ACTION Response to the CMISE entity. When an application receives an action indication and then must send an action response for each selected object on which execution is successful.

Format

```
M_Action_Resp (Connection_Id, Invoke_Id, Flags, Linked_Id,
               [Object_Class ], [ Object_Instance] , [ Response_Time ],
               [Action_Type ], [ Action_Reply_Info ] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier. If the response is a linked reply, the value is new invoke identifier local to this response. Otherwise, it is the action operation invoke identifier.

Flags	
Type:	cmise_flag
Access:	read only
Mechanism:	by reference

The significant bit flags are M_FG_LINKED_ID and M_FG_RORS_BASIC. M_FG_LINKED_ID is set if the response is a linked reply. M_FG_RORS_BASIC is set if the response contains the Invoke_Id only.

Linked_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the linked identifier of the action operation if the response is a linked reply. It is meaningful only if the linked identifier flag is set.

Object_Class	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter defines the managed object class. This argument is mandatory if the request has selected several classes.

Object_Instance	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the managed object instance. This argument is mandatory if the request has selected several instances.

Response_Time	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the time the response was generated.

Action_Type	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter specifies the specific action to be performed. It must be included if the Action_Reply_Info parameter is included.

Action_Reply_Info	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter contains the reply to the action.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

13.8.4. M_ACTION Confirm

M_Action_Cnf

This function is used to decode a M_ACTION Confirmation (event M_E_ACTION_CNF) sent by the CMISE entity.

Format

```
M_Action_Cnf (Connection_Id, Invoke_Id, Flags, Linked_Id,
              Object_Class, Object_Instance, Response_Time,
              Action_Type, Action_Reply_Info )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter defines the invoke identifier. If the response is a linked reply, the value is new invoke identifier local to this response. Otherwise, it is the action operation invoke identifier.

Flags	
Type:	cmise_flag
Access:	write only
Mechanism:	by reference

The only significant bit flag is M_FG_LINKED_ID. This flag is set if the response is a linked reply.

Linked_Id	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter defines the linked identifier of the action operation if the response is a linked reply. It is meaningful only if the linked identifier flag is set.

Object_Class	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter defines the managed object class. This argument is mandatory if the request has selected several classes.

Object_Instance	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter defines the managed object instance. This argument is mandatory if the request has selected several instances.

Response_Time	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the time the response was generated.

Action_Type	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter specifies the specific action to be performed. It must be included if the Action_Reply_Info parameter is included.

Action_Reply_Info

Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter gives additional reply information about the requested action. This cannot be empty if the action type is specified.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_INVREQ

CMISE_S_MEMERR

13.9. M_CREATE Service

The CREATE service is used by a CMISE user to request a peer system to create a new managed object instance, complete with its identification and the values of its associated management information. This service is always confirmed. Since there is no multiple object selection, the linked reply mechanism does not apply.

13.9.1. M_CREATE Request

M_Create_Req

This function sends a M_Create Request to the CMISE entity. It is used by a CMISE user to request a peer system to create a new managed object instance, complete with its identification and the values of its associated management information.

Format

```
M_Create_Req (Connection_Id, Invoke_Id, Flags, Object_Class,
              [Object_Instance ], [ Access_Control ],
              [Ref_Obj_Instance ], [ Attribute_List ] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id

Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier.

Flags	
Type:	cmise_flag
Access:	read only
Mechanism:	by reference

The only significant bit flag is M_FG_SUPER_INST. This flag is set if the Object_Instance parameter specifies the instance of the superior object instead of the instance of the new object.

Object_Class	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter specifies the class of the new managed object instance which is to be created.

Object_Instance	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter may specify the new object instance or its superior object instance.

Access_Control	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies optional access control information.

Ref_Obj_Instance	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies an existing instance of a managed object, called the reference object, of the same class as the managed object to be created. Attribute values associated with the reference object instance become the default values for those not specified in the Attribute_List.

Attribute_List	
Type:	cmise_buffer

Access:	read only
Mechanism:	by reference

This parameter specifies the set of attribute identifiers and values that are to be assigned to the managed object instance being created.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

13.9.2. M_CREATE Indication

M_Create_Ind

This function is used to decode a M_Create Indication (event M_E_CREATE_IND) sent by the CMISE entity.

Format

```
M_Create_Ind (Connection_Id, Invoke_Id, Flags,
              Object_Class, Object_Instance, Access_Control,
              Ref_Obj_Instance, Attribute_List )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter defines the invoke identifier.

Flags	
Type:	cmise_flag
Access:	write only

Mechanism:	by reference

The only significant bit flag is M_FG_SUPER_INST. This flag is set if the Object_Instance parameter specifies the instance of the superior object instead of the instance of the new object.

Object_Class	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter specifies the class of the new managed object instance that is to be created.

Object_Instance	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the instance of the managed object to which is to be registered.

Access_Control	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies optional access control information.

Ref_Obj_Instance	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies an existing instance of a managed object, called the reference object, of the same class as the managed object to be created. Attribute values associated with the reference object instance become the default values for those not specified in the Attribute_List.

Attribute_List	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the set of attribute identifiers and values that are to be assigned to the managed object instance being created.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_ILLPAR

CMISE_S_INVPAR

CMISE_S_INVREQ

CMISE_S_MEMERR

CMISE_S_MISPAR

13.9.3. M_CREATE Response

M_Create_Resp

This function sends a M_Create Response to the CMISE entity. It is used when an application receives a Create indication and the creation succeeds, it must send a Create response.

Format

```
M_Create_Resp (Connection_Id, Invoke_Id, Flags ,
               [Object_Class ], [ Object_Instance] ,
               [Response_Time ], [ Attribute_List] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier.

Flags	
Type:	cmise_flag
Access:	read only
Mechanism:	by reference

The only significant bit flag is M_FG_CONFIRMED. This flag is set if a confirmed event report is requested.

Object_Class	
Type:	cmise_oid

Access:	read only
Mechanism:	by reference

This parameter specifies the new object class.

Object_Instance	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the new object instance. The created object instance must be provided if it was not specified in the Create indication.

Response_Time	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the time the response was generated.

Attribute_List	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter contains the set of attribute identifiers and values that are assigned to the created object.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

13.9.4. M_CREATE Confirm

M_Create_Cnf

This function is used to decode a M_Create Confirmation (event M_E_CREATE_CNF) sent by the CMISE entity.

Format

```
M_Create_Cnf (Connection_Id, Invoke_Id, Flags,
              Object_Class, Object_Instance, Response_Time,
```

```
                Attribute_List )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	write only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter defines the invoke identifier.

Flags	
Type:	cmise_flag
Access:	write only
Mechanism:	by reference

The only significant bit flag is M_FG_CONFIRMED. This flag is set if a confirmed event report is requested.

Object_Class	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter specifies the new object class.

Object_Instance	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the new object instance. The created object instance must be provided if it was not specified in the Create indication.

Response_Time	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the time the response was generated.

Attribute_List	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter contains the set of attribute identifiers and values that are assigned to the created object.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_INVREQ

CMISE_S_MEMERR

13.10. M_DELETE Service

The DELETE service is used by a CMISE user to request a network management peer to delete one or several managed object instance(s). This service is always confirmed. Since multiple object selection is allowed, the linked reply mechanism applies.

13.10.1. M_DELETE Request

M_Delete_Req

This function sends an M_Delete Request to the CMISE entity. It is used when a CMISE user wants to request a network management peer to delete one or several managed object instance(s).

Format

```
M_Delete_Req (Connection_Id, Invoke_Id, Flags, Object_Class,
              Object_Instance, [ Access_Control ], [ Scope ], [ Filter ] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier.

Flags	
Type:	cmise_flag
Access:	read only
Mechanism:	by reference

The significant bit flags are M_FG_NOSYNC and M_FG_ATOMIC_SYNC.

M_FG_NOSYNC is set if no synchronization is requested. M_FG_ATOMIC_SYNC is set if atomic execution is required.

Object_Class	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter specifies the class of the managed object to which the filter (when supplied) is to be applied. If no filter is supplied, then this parameter specifies the class of the object to be managed.

Object_Instance	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the instance of the managed object to which the filter (when supplied) is to be applied. If no filter is supplied, then this parameter specifies the instance of the object to be managed.

Access_Control	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies optional access control information.

Scope	
Type:	cmise_scope
Access:	read only
Mechanism:	by reference

This parameter specifies the part of the containment tree involved in a network management service invocation.

Filter	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the filter used to select managed objects.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

13.10.2. M_DELETE Indication

M_Delete_Ind

This function is used to decode a M_Delete Indication (event M_E_DELETE_IND) sent by the CMISE entity.

Format

```
M_Delete_Ind (Connection_Id, Invoke_Id, Flags, Object_Class,
              Object_Instance, Access_Control, Scope, Filter )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter defines the invoke identifier.

Flags	
Type:	cmise_flag
Access:	write only
Mechanism:	by reference

The only significant bit flag is M_FG_ATOMIC_SYNC. This flag is set if atomic execution is required.

Object_Class

Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter specifies the class of the managed object to which the filter (when supplied) is to be applied. If no filter is supplied, then this parameter specifies the class of the object to be managed.

Object_Instance	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the instance of the managed object to which the filter (when supplied) is to be applied. If no filter is supplied, then this parameter specifies the instance of the object to be managed.

Access_Control	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies optional access control information.

Scope	
Type:	cmise_scope
Access:	write only
Mechanism:	by reference

This parameter specifies the part of the containment tree involved in a network management service invocation.

Filter	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the filter used to select managed objects.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_ILLPAR

CMISE_S_INVPAR

CMISE_S_INVREQ

CMISE_S_MEMERR

CMISE_S_MISPAR

13.10.3. M_DELETE Response

M_Delete_Resp

This function sends a M_Delete Response to the CMISE entity. It is used when an application receives a delete indication. Each successful object deletion must be reported with a delete response.

Format

```
M_Delete_Resp (Connection_Id, Invoke_Id, Flags, Linked_Id,
               [Object_Class ], [ Object_Instance] , [ Response_Time ] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier. If this response is a linked reply the value is a new invoke identifier local to this response. Otherwise, it is the Delete operation invoke identifier.

Flags	
Type:	cmise_flag
Access:	read only
Mechanism:	by reference

The significant bit flags are M_FG_LINKED_ID and M_FG_RORS_BASIC. The M_FG_LINKED_ID flag is set if the response is a linked reply. M_FG_RORS_BASIC flag is set if the response contains the Invoke_Id only.

Linked_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the linked identifier. Its only meaningful if the linked identifier flag is set. It gives the Delete operation invoke identifier if the response is a linked reply.

Object_Class	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

This parameter defines the managed object class. This argument is mandatory if the request has selected several classes.

Object_Instance	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the deleted object instance. This parameter is mandatory if the Delete request has selected several objects.

Response_Time	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter specifies the time the response was generated.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_IPCERR

CMISE_S_MEMERR

13.10.4. M_DELETE Confirm

M_Delete_Cnf

This function is used to decode a M_Delete Confirmation (event M_E_DELETE_CNF) sent by the CMISE entity.

Format

```
M_Delete_Cnf (Connection_Id, Invoke_Id, Flags, Linked_Id,
              Object_Class, Object_Instance, Response_Time )
```

Arguments

Connection_Id

Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter defines the invoke identifier. If this response is a linked reply the value is a new invoke identifier local to this response. Otherwise, it is the delete operation invoke identifier.

Flags	
Type:	cmise_flag
Access:	write only
Mechanism:	by reference

The only significant bit flag is M_FG_LINKED_ID. This flag is set if the response is a linked reply.

Linked_Id	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter defines the linked identifier. Its only meaningful if the linked identifier flag is set. It gives the Delete operation invoke identifier if the response is a linked reply.

Object_Class	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter defines the managed object class. This argument is mandatory if the request has selected several classes.

Object_Instance	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter specifies the deleted object instance. This parameter is mandatory if the Delete request has selected several objects.

Response_Time	
Type:	cmise_buffer

Access:	write only
Mechanism:	by reference

This parameter specifies the time the response was generated.

Return Values

CMISE_S_NORMAL

CMISE_S_ASNERR

CMISE_S_INVPAR

CMISE_S_INVREQ

CMISE_S_MEMERR

13.11. M_ERROR Service

The ERROR service is used by a CMISE user to report the failure of any requested operation. When a CMISE application acting as an operation performer meets an error condition for a confirmed operation, it must use the Error Service to report the problem to the operation invoker with information about the error and its context.

13.11.1. M_ERROR Response

M_Error_Resp

This function sends an M_ERROR Response to the CMISE entity. It is used to report the failure of any requested operation.

Format

```
M_Error_Resp (Connection_Id, Invoke_Id, Flags, Linked_Id,
              [Event_Code], CMISE_Error_Code, [Scope_Or_CancelGetId],
               Object_Class, Object_Instance, [ActionEvent_Type],
              [Response_Time], [Service_Data] )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only

Mechanism:	by reference

This parameter defines the invoke identifier. If the response is a linked reply its value is a new invoke identifier local to this response. Otherwise, it is the operation invoke identifier.

Flags	
Type:	cmise_flag
Access:	read only
Mechanism:	by reference

The significant bit flags are:

M_FG_LINKED_ID

This flag is set if the response is a linked reply.

M_FG_COMPLEX_SCOPE

This flag is set if the Specified scope is not supported.

(CMISE_Error_Code = M_CD_COMPLX_LIMIT)

M_FG_COMPLEX_FILTER

This flag is set if the Specified Filter is not supported.

(CMISE_Error_Code = M_CD_COMPLX_LIMIT)

M_FG_COMPLEX_SYNCH

This flag is set if the Specified synch is not supported.

(CMISE_Error_Code = M_CD_COMPLX_LIMIT)

M_FG_ATOMIC_SYNC

This flag is set if atomic execution is not supported, and has meaning only if M_FG_COMPLEX_SYNCH is set.

Linked_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the operation invoke identifier if the response is a linked reply. It is only meaningful if the linked identifier flag is set.

Event_Code	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter specifies the erroneous operation.

The possible values are:

M_E_GET_RESP for Get
M_E_SET_RESP for Set
M_E_ACTION_RESP for Action
M_E_DELETE_RESP for Delete

CMISE_Error_Code	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter identifies the error. The possible values are:

M_CD_NO_SUCH_CLASS
M_CD_NO_SUCH_REF_OBJ
M_CD_NO_SUCH_INST
M_CD_NO_SUCH_EVT_TYP
M_CD_ACCESS_DENIED
M_CD_NO_SUCH_ARG
M_CD_SYNC_NOT_SUPP
M_CD_INV_ARG_VAL
M_CD_INV_FILTER
M_CD_INV_SCOPE
M_CD_NO_SUCH_ATT
M_CD_INV_OBJ_INST
M_CD_INV_ATT_VAL
M_CD_MISS_ATT_VAL
M_CD_GET_LIST_ERR
M_CD_CLASS_INST_CONFL
M_CD_SET_LIST_ERR
M_CD_COMPLX_LIMIT
M_CD_NO_SUCH_ACT
M_CD_MISTYP_OPER
M_CD_PROC_FAIL
M_CD_NO_SUCH_INVOK_ID
M_CD_DUPLIC_OBJ_ID
M_CD_OPER_CANCELD

Scope_Or_CancelGetId	
Type:	cmise_scope (unsigned long int)
Access:	read only
Mechanism:	by reference

Depending on the error code, this parameter may give either the Scope or the Cancel_Get invoke identifier.

Object_Class	
Type:	cmise_oid

Access:	read only
Mechanism:	by reference

This parameter specifies the managed object class. It is mandatory if the error response is a linked reply.

Object_Instance	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the object instance. It is mandatory if the error response is a linked reply.

ActionEvent_Type	
Type:	cmise_oid
Access:	read only
Mechanism:	by reference

Depending on the error code, this parameter may give an event type or an action type.

Response_Time	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

This parameter defines the response time.

Service_Data	
Type:	cmise_buffer
Access:	read only
Mechanism:	by reference

Contents of this parameter depends on the CMISE_Error_Code.

The following errors do not need Service_Data:

M_CD_NO_SUCH_CLASS
M_CD_NO_SUCH_ARG
M_CD_NO_SUCH_INST
M_CD_INV_SCOPE
M_CD_ACCESS_DENIED
M_CD_INV_OBJ_INST
M_CD_SYNC_NOT_SUPP
M_CD_CLASS_INST_CONFL
M_CD_NO_SUCH_ACT
M_CD_MISTYP_OPER
M_CD_DUPLIC_OBJ_ID
M_CD_NO_SUCH_INVOK_ID
M_CD_NO_SUCH_REF_OBJ

M_CD_OPER_CANCELD
M_CD_NO_SUCH_EVT_TYP

The following errors need Service_Data:

M_CD_INV_FILTER	Service_Data contains CMISFilter.
M_CD_NO_SUCH_ATT	Service_Data contains a set of AttributeId. The set should contain a single AttributeId.
M_CD_INV_ATT_VAL	Service_Data contains a set of Attribute. The set should contain a single Attribute.
M_CD_GET_LIST_ERR	Service_Data contains a set of GetInfoStatus.
M_CD_SET_LIST_ERR	Service_Data contains a set of SetInfoStatus.
M_CD_PROC_FAIL	Service_Data contains a set of SpecificErrorInfo. The set should contain a single SpecificErrorInfo.
M_CD_INV_ARG_VAL	Service_Data contains a set of ANY DEFINED BY ActionEvent Type.
M_CD_MISS_ATT_VAL	Service_Data contains a set of AttributeId.
M_CD_COMPLX_LIMIT	Service_Data contains CMISFlter in case M_FG_COMPLEX_FILTER bit is set in the flag.

Return Values

CMISE_S_NORMAL
CMISE_S_ASNERR
CMISE_S_IPCERR
CMISE_S_INVPAR
CMISE_S_MEMERR

13.11.2. M_ERROR Confirm

M_Error_Cnf

This function is used to decode an M_ERROR Confirmation (event M_E_ERROR_CNF) sent by the CMISE entity.

Format

```
M_Error_Cnf (Connection_Id, Invoke_Id, Flags, Linked_Id,
             Event_Code, CMISE_Error_Code, Scope_Or_CancelGetId,
             Object_Class, Object_Instance, ActionEvent_Type,
             Response_Time, Service_Data )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter defines the invoke identifier. If the response is a linked reply, its value is a new invoke identifier local to this response. Otherwise, it is the operation invoke identifier.

Flags	
Type:	cmise_flag
Access:	write only
Mechanism:	by reference

The significant bit flags are:

M_FG_LINKED_ID

This flag is set if the response is a linked reply.

M_FG_COMPLEX_SCOPE

This flag is set if the Specified scope is not supported.

(CMISE_Error_Code = M_CD_COMPLX_LIMIT)

M_FG_COMPLEX_FILTER

This flag is set if the Specified Filter is not supported.

(CMISE_Error_Code = M_CD_COMPLX_LIMIT)

M_FG_COMPLEX_SYNCH

This flag is set if the Specified synch is not supported.

(CMISE_Error_Code = M_CD_COMPLX_LIMIT)

M_FG_ATOMIC_SYNC

This flag is set if atomic execution is not supported, and has meaning only if M_FG_COMPLEX_SYNCH is set.

Linked_Id	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter defines the operation invoke identifier if the response is a linked reply. It is only meaningful if the linked identifier flag is set.

Event_Code	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter specifies the erroneous operation.

The possible values are:

M_E_GET_RESP for Get
M_E_SET_RESP for Set
M_E_ACTION_RESP for Action
M_E_DELETE_RESP for Delete

CMISE_Error_Code	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter identifies the error. The possible values are:

M_CD_NO_SUCH_CLASS
M_CD_NO_SUCH_REF_OBJ
M_CD_NO_SUCH_INST
M_CD_NO_SUCH_EVT_TYP
M_CD_ACCESS_DENIED
M_CD_NO_SUCH_ARG
M_CD_SYNC_NOT_SUPP
M_CD_INV_ARG_VAL
M_CD_INV_FILTER
M_CD_INV_SCOPE
M_CD_NO_SUCH_ATT
M_CD_INV_OBJ_INST
M_CD_INV_ATT_VAL
M_CD_MISS_ATT_VAL
M_CD_GET_LIST_ERR
M_CD_CLASS_INST_CONFL
M_CD_SET_LIST_ERR
M_CD_COMPLX_LIMIT
M_CD_NO_SUCH_ACT
M_CD_MISTYP_OPER
M_CD_PROC_FAIL
M_CD_NO_SUCH_INVOK_ID
M_CD_DUPLIC_OBJ_ID
M_CD_OPER_CANCELD

Scope_Or_CancelGetId	
Type:	cmise_scope (unsigned long int)
Access:	write only
Mechanism:	by reference

Depending on the error code, this parameter may give either the Scope or the Cancel_Get invoke identifier.

Object_Class	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

This parameter specifies the managed object class. It is mandatory if the error response is a linked reply.

Object_Instance	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter defines the object instance. It is mandatory if the error response is a linked reply.

ActionEvent_Type	
Type:	cmise_oid
Access:	write only
Mechanism:	by reference

Depending on the error code, this parameter may give an event type or an action type.

Response_Time	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

This parameter defines the response time.

Service_Data	
Type:	cmise_buffer
Access:	write only
Mechanism:	by reference

Contents of this parameter depends on the CMISE_Error_Code.

Following errors do not have Service_Data:

M_CD_NO_SUCH_CLASS
M_CD_NO_SUCH_ARG
M_CD_NO_SUCH_INST
M_CD_INV_SCOPE
M_CD_ACCESS_DENIED
M_CD_INV_OBJ_INST
M_CD_SYNC_NOT_SUPP
M_CD_CLASS_INST_CONFL

M_CD_NO_SUCH_ACT
M_CD_MISTYP_OPER
M_CD_DUPLIC_OBJ_ID
M_CD_NO_SUCH_INVOK_ID
M_CD_NO_SUCH_REF_OBJ
M_CD_OPER_CANCELD
M_CD_NO_SUCH_EVT_TYP

Following errors have Service_Data:

M_CD_INV_FILTER	Service_Data contains CMISFilter.
M_CD_NO_SUCH_ATT	Service_Data contains a set of AttributeId. The set should contain a single AttributeId.
M_CD_INV_ATT_VAL	Service_Data contains a set of Attribute. The set should contain a single Attribute.
M_CD_GET_LIST_ERR	Service_Data contains a set of GetInfoStatus.
M_CD_SET_LIST_ERR	Service_Data contains a set of SetInfoStatus.
M_CD_PROC_FAIL	Service_Data contains a set of SpecificErrorInfo. The set should contain a single SpecificErrorInfo.
M_CD_INV_ARG_VAL	Service_Data contains a set of ANY DEFINED BY ActionEvent Type.
M_CD_MISS_ATT_VAL	Service_Data contains a set of AttributeId.
M_CD_COMPLX_LIMIT	Service_Data contains CMISFilter in case M_FG_COMPLEX_FILTER bit is set in the flag.

Return Values

CMISE_S_NORMAL
CMISE_S_ASNERR
CMISE_S_IPCERR
CMISE_S_INVPAR
CMISE_S_MEMERR

13.11.3. CMISE_Error_Code Parameter Usage

For each error, the following information is specified:

- Its common nickname
- A short explanation of what it means
- Its constant name
- The operations for which it can be sent
- The list of the meaningful information with the parameter(s) used to pass them

No such object class (M_CD_NO_SUCH_CLASS)

Meaning:	The class of the specified managed object was not recognized.

Primitives:	Event_Report, Get, Set, Action, Create, Delete.
Mandatory parameter:	Object_class.

No such object instance (M_CD_NO_SUCH_OBJ_INST)

Meaning:	The instance of the specified managed object was not recognized.
Primitives:	Event_Report, Get, Set, Action, Create, Delete.
Mandatory parameter:	Object_instance.

Access denied (M_CD_ACCESS_DENIED)

May be used in a linked reply.

Meaning:	The requested operation was not performed for reasons pertinent to the security of the open system.
Primitives:	Get, Set, Action, Create, Delete.
No parameter.	

Synchronization not supported (M_CD_SYNC_NOT_SUPP)

Meaning:	The synchronization type specified is not supported.
Primitives:	Get, Set, Action, Delete.
Mandatory parameter:	Flags.

Invalid filter (M_CD_INV_FILTER)

Meaning:	The filter parameter contains an invalid assertion or an unrecognized logical operator.
Primitives:	Get, Set, Action, Delete.
Mandatory parameter:	Service_Data should contain CMISFilter.

No such attribute (M_CD_NO_SUCH_ATT)

Meaning:	The identifier for the specified attribute was not recognized.
Primitives:	Create.
Mandatory parameter:	Service_Data should contain a set of AttributeId. The set should contain a single AttributeId.

Invalid attribute value (M_CD_INV_ATT_VAL)

May be used in a linked reply.

Meaning:	The attribute value specified was out of range or otherwise inappropriate.

Primitives:	Create.
Mandatory parameter:	Service_Data should contain a set of Attribute. The set should contain a single Attribute.

Get list error (M_CD_GET_LIST_ERR)

May be used in a linked reply.

Meaning:	One or more attribute values were not read for the following reasons: • Access denied: The requested Get operation was not performed for reasons pertinent to the security of the open system • No such attribute: The identifier for the specified attribute or attribute group was not recognized.
Primitives:	Get.
Mandatory parameter:	Service_Data should contain a set of GetInfoStatus.

Set list error (M_CD_SET_LIST_ERR)

May be used in a linked reply.

Meaning:	One or more attribute values were not modified for the following reasons: • Access denied: The requested Set operation was not performed for reasons pertinent to the security of the open system • Invalid attribute value: The attribute value specified was out of range or otherwise inappropriate • No such attribute: The identifier for the specified attribute or attribute group was not recognized.
Primitives:	Set.
Mandatory parameter:	Service_Data should contain a set of SetInfoStatus.

No such action (M_CD_NO_SUCH_ACTION)

May be used in a linked reply.

Meaning:	The action type specified was not supported.
Primitives:	Action.
Mandatory parameter:	Object_class ActionEvent_Type

Processing failure (M_CD_PROC_FAIL)

May be used in a linked reply.

Meaning:	A general failure in processing the operation was encountered.
Primitives:	Event_Report, Get, Set, Create, Delete, Action, Cancel_Get.
Parameter (Mandatory only if linked reply):	Object_class Service_Data should contain a set of SpecificErrorInfo. The set should contain a single SpecificErrorInfo.
Optional parameter:	Object_instance

Duplicate object instance (M_CD_DUPLIC_OBJ_ID)

Meaning:	The new managed object instance value supplied by the Create operation invoker was already registered for a managed object of the specified class.
Primitives:	Create.
Mandatory parameter:	Object_instance.

No such reference object (M_CD_NO_SUCH_REF_OBJ)

Meaning:	The referenced object instance parameter was not recognized.
Primitives:	Create.
Mandatory parameter:	Object_instance.

No such event (M_CD_NO_SUCH_EVENT)

Meaning:	The event type specified was not recognized.
Primitives:	Event_report.
Mandatory parameter:	Object_class ActionEvent_Type

No such argument (M_CD_NO_SUCH_ARGUMENT)

Meaning:	The event or action information specified was not recognized.
Primitives:	Event_Report, Action.
Optional parameter:	ActionEvent_Type.

Invalid argument value (M_CD_INV_ARG_VAL)

Meaning:	The event or action information value specified was out of range or otherwise inappropriated.

Primitives:	Event_Report, Action.
Optional parameter:	Service_Data should contain a set of any defined by ActionEvent Type.

Invalid scope (M_CD_INV_SCOPE)

Meaning:	The scope parameter value is invalid.
Primitives:	Get, set, Action, Delete.
Mandatory parameter:	Scope_Or_CancelGetId.

Invalid object instance (M_CD_INV_OBJ_INST)

Meaning:	The object instance name specified implies a violation of the naming rules.
Primitives:	Create.
Mandatory parameter:	Object_Instance.

Missing attribute value (M_CD_MISS_ATT_VAL)

Meaning:	A required attribute value was not supplied, and a default value was not available.
Primitives:	Create.
Mandatory parameter:	Service_Data should contain a set of AttributeId.

Class instance conflict (M_CD_CLASS_INST_CONFL)

Meaning:	The specified managed object instance may not be created as a member of the specified class.
Primitives:	Get, Set, Action, Create, Delete.
Mandatory parameter:	Object_class.

Complexity limitation (M_CD_COMPLX_LIMIT)

Meaning:	The requested operation was not performed because a parameter was too complex.
Primitives:	Get, Set, Action, Delete.
Optional parameter:	Scope_Or_CancelGetId, Service_Data may contain CMISEFilter

Mistyped operation (M_CD_MISTYP_OPER)

Meaning:	The Get invoke identifier does not refer to a Get operation.
Primitives:	Cancel_Get.
No parameter.	

No such invocation identifier (M_CD_NO_SUCH_INVOK_ID)

Meaning:	The Get invoke identifier does not refer to a Get operation.
Primitives:	Cancel_Get.
Mandatory parameter:	Scope_Or_CancelGetId should contain Erroneous invoke identifier

Operation cancelled (M_CD_OPER_CANCELD)

Meaning:	The Get operation was cancelled by a Cancel_Get operation, and no further attribute values will be returned by the invocation of the get service.
No parameter.	

13.12. M_REJECT Service

The REJECT service is used by a CMISE user to reject a received request or response. If a CMISE user gets an error while decoding either an indication or confirmation, it must use the Reject Service to report the problem to the operation invoker. The CMISE user can use this service to abort a request due to unforeseen error without aborting an association.

13.12.1. M_REJECT Response

M_Reject_Resp

This function sends a M_REJECT Response to the CMISE entity. It is used to report the failure to decode a received indication or confirmation.

Format

```
M_Reject_Resp (Connection_Id, [Invoke_Id], Problem_type,
               problem_number )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This parameter defines the invoke identifier if it can be decoded. Otherwise it should be NULL.

Problem_Type

Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This should be one of the constants outlined as follows.

M_PR_GENERAL_PROB	Indicating that an unforeseen problem arose while calling OSAK services.
M_PR_INVOKE_PROB	Indicating that an unforeseen problem arose while handling an Invoke PDU.
M_PR_RESULT_PROB	Indicating that an unforeseen problem arose while handling an Response PDU.
M_PR_ERROR_PROB	Indicating that an unforeseen problem arose while handling an Error PDU.

Problem_Number	
Type:	unsigned long int
Access:	read only
Mechanism:	by reference

This should be one of the constants outlined as follows.

- Allowed Problem_numbers when problem type is M_PR_GENERAL_PROB:

M_SE_UNRECOGNIZE	Unrecognized APDU problem
M_SE_MISTYPED_PDU	Mistyped APDU problem
M_SE_BAD_STRUCT	Badly structured problem

- Allowed Problem_numbers when Problem type is M_PR_INVOKE_PROB:

M_SE_DUP_INVOKE	Duplicate invocation problem. Performer must send this code if Invoke_Id corresponds to outstanding operation.
M_SE_UNKNOWN_OP	Unrecognized operation problem. Performer must send this code if operation value is not in range (0 – 10).
M_SE_MISTYPED_ARG	Mistyped argument problem
M_SE_RES_LIMIT	Limited resources problem
M_SE_RELEASING	Initiator releasing problem
M_SE_UNK_LINK	Unknown link ID problem
M_SE_LINK_EXPEC	Linked response expected problem
M_SE_CHILD_OP	Unexpected child operation problem

- Allowed Problem_numbers when problem type is M_PR_RESULT_PROB:

M_SE_UNK_INVOKE	Unrecognized invocation problem

M_SE_UNEX_RESULT	Result response unexpected problem
M_SE_MISTYPED_RES	Mistyped result problem

- Allowed Problem_numbers when Problem type is M_PR_ERROR_PROB:

M_SE_UNK_INVOKE	Unrecognized invocation problem
M_SE_UNEX_ERR_RES	Error response unexpected problem
M_SE_UNK_ERROR	Unrecognized error problem
M_SE_UNEX_ERROR	Unexpected error problem
M_SE_MISTYPED_ERR	Mistyped error problem

Return Values

CMISE_S_NORMAL
CMISE_S_ASNERR
CMISE_S_ILLPAR
CMISE_S_INVPAR
CMISE_S_IPCERR
CMISE_S_MEMERR

13.12.2. M_REJECT Confirm

M_Reject_Cnf

This function is used to decode a M_REJECT Confirmation (event M_E_REJECT_CNF) sent by the CMISE entity.

Format

```
M_Reject_Cnf (Connection_Id, [Invoke_Id], Flags, Problem_type,
              problem_number )
```

Arguments

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter identifies the association referred to by this function.

Invoke_Id	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter defines the invoke identifier if it can be decoded.

Flags

Type:	cmise_flag
Access:	write only
Mechanism:	by reference

The only significant bit flags are:

M_FG_INVOKE_ID: set if the Invoke_Id is present.
M_FG_PROB_TYPE: set if the is Problem_Type present.
M_FG_PROB_NUM: set if the Problem_Number is present.

Problem_Type	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This should be one of the constants.

M_PR_GENERAL_PROB	Indicating that an unforeseen problem arose while calling OSAK services.
M_PR_INVOKE_PROB	Indicating that an unforeseen problem arose while handling an Invoke PDU.
M_PR_RESULT_PROB	Indicating that an unforeseen problem arose while handling an Response PDU.
M_PR_ERROR_PROB	Indicating that an unforeseen problem arose while handling an Error PDU.

Problem_Number	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This should be one of the constants:

Allowed Problem_numbers when problem type is M_PR_GENERAL_PROB:

M_SE_UNRECOGNIZE	Unrecognized APDU problem
M_SE_MISTYPED_PDU	Mistyped APDU problem
M_SE_BAD_STRUCT	Badly structured problem

Allowed Problem_numbers when problem type is M_PR_INVOKE_PROB:

M_SE_DUP_INVOKE	Duplicate invocation problem. Performer must send this code if Invoke_Id corresponds to outstanding operation.
M_SE_UNKNOWN_OP	Unrecognized operation problem. Performer must send this code if operation value is not in range (0 – 10)
M_SE_MISTYPED_ARG	Mistyped argument problem

M_SE_RES_LIMIT	Limited resources problem
M_SE_RELEASING	Initiator releasing problem
M_SE_UNK_LINK	Unknown link ID problem
M_SE_LINK_EXPEC	Linked response expected problem
M_SE_CHILD_OP	Unexpected child operation problem

Allowed Problem_numbers when problem type is M_PR_RESULT_PROB:

M_SE_UNK_INVOKE	Unrecognized Invocation problem
M_SE_UNEX_RESULT	Result response unexpected problem
M_SE_MISTYPED_RES	Mistyped result problem

Allowed Problem_numbers when problem type is M_PR_ERROR_PROB:

M_SE_UNK_INVOKE	Unrecognized invocation problem
M_SE_UNEX_ERR_RES	Error Response unexpected problem
M_SE_UNK_ERROR	Unrecognized error problem
M_SE_UNEX_ERROR	Unexpected error problem
M_SE_MISTYPED_ERR	Mistyped error problem

Return Values

CMISE_S_NORMAL
CMISE_S_ASNERR
CMISE_S_INVPAR
CMISE_S_MEMERR

13.13. CMISE Support Services

A number of support services that are not part of the CMIS specification are provided as functions in the CMISE API. These services are used by the CMISE service user applications for synchronization and program control.

13.13.1. cmise_wait_for_event

This function provides a mechanism for a CMISE service user application to wait for an event to arrive from a peer CMISE service user. A timeout value can be specified to limit the amount of time to wait for an event.

Format

```
cmise_wait_for_event (count, port_list, time_out)
```

Arguments

count	
Type:	longword
Access:	read only

Mechanism:	by value

This parameter defines the number of ports in the port list.

port_list	
Type:	cmise_port_list
Access:	read only
Mechanism:	by reference

This parameter contains the list connection identifiers that are waiting for events. The event_waiting field of this structure contains a bit mask of the events that have arrived. Possible values are:

time_out	
Type:	longword
Access:	read only
Mechanism:	by reference

This parameter defines the maximum time, in seconds, to wait for an event. A value of zero indicates no waiting. A null pointer indicates an indefinite wait. The maximum value is 86400 seconds.

Return Values

CMISE_S_NORMAL

CMISE_S_NOEVENT

13.13.2. cmise_what_event

This function returns the CMISE event that has arrived from the peer CMISE user. This function should be called after a successful return from cmise_wait_for_event.

Format

```
cmise_what_event (Connection_Id, cmise_event)
```

Parameters

Connection_Id	
Type:	cmise_port
Access:	read only
Mechanism:	by reference

This parameter defines the connection identifier that has received a notification of an event.

cmise_event	
Type:	unsigned long int
Access:	write only
Mechanism:	by reference

This parameter identifies the CMISE event that has arrived. Possible values are:

M_E_INITIALIZE_ACC
M_E_INITIALIZE_IND
M_E_INITIALIZE_REJ
M_E_TERMINATE_ACC
M_E_TERMINATE_IND
M_E_TERMINATE_REJ
M_E_P_ABORT_IND
M_E_U_ABORT_IND
M_E_ACTION_CNF
M_E_ACTION_IND
M_E_CREATE_CNF
M_E_CREATE_IND
M_E_DELETE_CNF
M_E_DELETE_IND
M_E_ERROR_CNF
M_E_EVT_REPORT_CNF
M_E_EVT_REPORT_IND
M_E_GET_IND
M_E_GET_CNF
M_E_SET_CNF
M_E_SET_IND
M_E_CANCEL_GET_IND
M_E_CANCEL_GET_CNF
M_E_REJECT_CNF

Return Values

CMISE_S_NORMAL

CMISE_S_IPCERR

CMISE_S_INVPAR

Chapter 14. Checking CMISE Status Codes

This chapter lists the status codes returned by the ISO CMISE API to your application. All the status codes are returned as function return values. In some cases when an error status is returned, you can find additional status information in the second longword of the `cmise_stat` field of the Connection_Id parameter. See the descriptions of the CMISE errors to determine which ones provide additional status information.

14.1. Status Codes

- CMISE_S_NORMAL

  Normal successful completion.

- CMISE_S_ASNERR

  The CMISE API encountered an error while encoding or decoding ASN.1 data.

- CMISE_S_ILLPAR

  Your application entered an illegal parameter for the function.

- CMISE_S_INVPAR

  Your application supplied an invalid value for a parameter. Check for a secondary status in the `cmise_stat` field of the Connection_Id parameter. See *Section 14.2, "CMIP Status Codes"* for possible secondary status values.

- CMISE_S_INVREQ

  Your application made an invalid request. This is an internal error that is not passed to the application program.

- CMISE_S_IPCERR

  The CMISE API encountered an error during inter-process communication. Check for a secondary status in the `cmise_stat` field of the Connection_Id parameter. See *Section 14.1.1, "OSAK Status Codes"* for possible secondary status values.

- CMISE_S_MEMERR

  The CMISE API encountered an error when trying to allocate memory.

- CMISE_S_MISPAR

  Your application is missing a required parameter. Check for a secondary status in the `cmise_stat` field of the Connection_Id parameter. See *Section 14.2, "CMIP Status Codes"* for possible secondary status values.

- CMISE_S_NOEVENT

  No event is waiting.

- CMISE_S_PRSADD

  The presentation address specified is invalid.

14.1.1. OSAK Status Codes

- OSAK_S_BADPARAM

  Your application specified a bad parameter.

- OSAK_S_DISRUPTED

  A disruptive event has occurred.

- OSAK_S_INSFMEM

  The process running the application does not have enough dynamic memory to service the request.

- OSAK_S_INSFWS

  There is not enough workspace in the parameter block.

- OSAK_S_INVACTION

  The action_result parameter is invalid.

- OSAK_S_INVFUNC

  The call is invalid. You made an incorrect sequence of calls.

- OSAK_S_INVFUS

  The functional units are invalid.

- OSAK_S_INVPCTXT

  The presentation context is invalid.

- OSAK_S_INVPORT

  The port identifier is invalid.

- OSAK_S_INVREASON

  The reason code specified is invalid.

- OSAK_S_NOCTXTNAME

  The required application context name is missing.

- OSAK_S_TRANSERR

  There is an error in the transport layer.

14.2. CMIP Status Codes

- CMIP_S_CMIPBADOBJID

The object class is invalid.

- CMIP_S_CMIPINSTANCE

  The object instance is invalid.

- CMIP_S_INV_OBJ_CLASS

  The object class parameter is invalid.

- CMIP_S_MISSING_FILTER

  The filter parameter is missing.

- CMIP_S_MISSING_SCOPE

  The scope parameter is missing.

- CMIP_S_NO_ACTION_EVENT

  The action event parameter is missing.

- CMIP_S_NO_FILTER

  The filter parameter is missing.

- CMIP_S_NO_INSTANCE

  The object instance parameter is missing.

- CMIP_S_NO_OBJ_CLASS

  The object class parameter is missing.

Appendix A. $QIO(W) Status Codes and OSI Reason Codes

This appendix lists:

- The status codes returned by $QIO or $QIOW calls (*Section A.1, "Status Codes Returned by $QIO(W) Calls"*)
- The OSI reason codes that can be returned by a remote transport service (*Section A.2, "OSI Reason Codes"*)
- The OSI transport-specific status codes that can be returned in an IOSB by unsuccessful $QIO or $QIOW calls (*Section A.3, "OSI Transport-Specific Reason Codes"*)

A.1. Status Codes Returned by $QIO(W) Calls

Table A.1, "Status Codes for $QIO System Services" is a summary of the status codes returned from $QIO or $QIOW ($QIO(W)) calls.

For each status code listed on the left, the table shows the $QIO(W) call returning that code, and whether the code is returned in R0, the IOSB, or both.

For each $QIO(W) call shown at the top, the table shows which status codes are returned, and whether they are returned in R0, the IOSB, or both.

Table A.1. Status Codes for $QIO System Services

SS Call Status Code	$QIO Req	$QIO Acc	$QIO Rej	$QIO TSAP	$QIO Send	$QIO Send Exped	$QIO Read	$QIO SENSE
SS$_ABORT	IOSB	IOSB			IOSB	IOSB	IOSB	IOSB
SS$_ACCVIO	R0	R0	R0	R0	R0	R0	R0	R0 R0
SS$_BADPARAM	Both	Both	Both	R0				R0 Both
SS$_BUFFEROVF	R0	R0					IOSB	
SS$_CONNECFAIL	IOSB							
SS$_DATAOVERUN							IOSB	
SS$_DEVOFFLINE	R0	R0	R0	R0	R0	R0	R0	R0 R0
SS$_EXQUOTA	R0	R0	R0	R0				R0 R0
SS$_FILALRACC	R0	R0	R0					R0
SS$_FILNOTACC		IOSB	IOSB		Both	Both	Both	Both

SS Call Status Code	$QIO Req	$QIO Acc	$QIO Rej	$QIO TSAP	$QIO Send	$QIO Send Exped	$QIO Read	$QIO SENSE
								IOSB
*SS$_ILLEFC	R0	R0	R0	R0	R0	R0	R0	R0 R0
SS$_ILLIOFUNC	R0	R0	R0	R0	R0	Both	R0	R0 R0
*SS$_ILLSER	R0	R0	R0	R0	R0	R0	R0	R0 R0
SS$_INSFARG	R0	R0	R0	R0	R0	R0	R0	R0 R0
SS$_INSFMEM	Both	Both	R0	R0				R0 Both
*SS$_IVCHAN	R0	R0	R0	R0	R0	R0	R0	R0 R0
SS$_LINKABORT		IOSB			IOSB	IOSB	IOSB	
SS$_LINKDISCON					IOSB	IOSB	IOSB	
SS$_NOLINKS	IOSB							
SS$_NOMBX				R0				
SS$_NOPRIV	R0	Both	IOSB	R0				IOSB IOSB
SS$_NORMAL	Both	Both	Both	Both	Both	Both	Both	Both Both
SS$_ NOSUCHNODE	IOSB							
SS$_NOSUCHOBJ	IOSB							
SS$_PATHLOST	IOSB	IOSB			IOSB	IOSB	IOSB	
SS$_PROTOCOL	IOSB	IOSB			IOSB	IOSB	IOSB	
SS$_REJECT	IOSB	IOSB						
SS$_REMSRC	IOSB	IOSB						
SS$_SHUT	IOSB	IOSB						
SS$_THIRDPARTY	IOSB	IOSB			IOSB	IOSB	IOSB	IOSB
SS$_TIMEOUT	IOSB	IOSB			IOSB	IOSB	IOSB	
SS$_TOOMUCHDATA						IOSB		R0
SS$_ UNREACHABLE	IOSB							
SS$_WRITLCK				IOSB				

Key:	
$QIO Req	Request a connection: $QIO(W)(IO$_ACCESS)
$QIO Acc	Accept a connection: $QIO(W)(IO$_ACCESS)
$QIO Rej	Reject a connection: $QIO(W)(IO$_ACCESS!ABORT)
$QIO TSAP	Attach a task to a TSAP: $QIO(W)(IO$_ACPCONTROL)
$QIO Send	Send data: $QIO(W)(IO$_WRITEVBLK)
$QIO Send Exped	Send expedited data:$QIO(IO$_WRITEVBLK!IO$M_INTERRUPT)
$QIO Read	Receive data: $QIO(W)(IO$_READVBLK)
$QIO Disc	Conclude a connection: $QIO(W)(IO$_DEACCESS)
$QIO Sens	Examine an inbound connection: $QIO(IO$_SENSEMODE!IO$M_ACCESS)

A.2. OSI Reason Codes

Table A.2, "OSI Reason Codes" lists the OSI reason codes that may be returned by a remote transport service.

Table A.2. OSI Reason Codes

Code	**Explanation**
Codes Returned by Class 2/Class 4Implementations	
80	Normal disconnect by session entity.
81	Remote transport entity congestion at connect request time.
82	Connection negotiation failed.
83	Duplicate source references detected for same pair of NSAPs.
84	Mismatched references.
85	Protocol error.
87	Reference overflowed.
88	Transport connection refused on this network connection.
8A	Header or parameter length invalid.
Codes Returned by All Classes	
00	Reason not specified.
01	Congestion at TSAP.
02	Session entity not attached to TSAP.
03	Address unknown.
Codes Returned by Error TPDUs	

Code	Explanation
00	Reason not specified.
01	Invalid parameter code.
02	Invalid TPDU type.
03	Invalid parameter value.

A.3. OSI Transport-Specific Reason Codes

Table A.3, "OSI Transport-Specific Reason Codes Returned in the IOSB" lists the OSI transport-specific reason codes that may be returned in the IOSB of an unsuccessful $QIO(W) call.

Table A.3. OSI Transport-Specific Reason Codes Returned in the IOSB

Reason Code	Associated Primary Status Code
OSIT$_IERADDR	SS$_CONNECFAIL The Internet network has rejected the transmitted IPDU. The destination NSAP is unknown or unreachable.
OSIT$_IERDISCARD	SS$_CONNECFAIL The IPDU has been discarded because it used an option that was not supported by the network.
OSIT$_IERGENERAL	SS$_CONNECFAIL The Internet network has rejected the transmitted IPDU and has not specified a reason.
OSIT$_IERLIFE	SS$_CONNECFAIL The IPDU lifetime expired.
OSIT$_IERREASS	SS$_CONNECFAIL The Internet network has discovered inconsistencies during a reassembly operation. This problem can be caused by old packets being present in the network. The problem may disappear by itself.
OSIT$_IERSRCROUT	SS$_CONNECFAIL There is an error in the source routing option in the IPDU. One of the intermediate systems on the route between the hosts unable to forward the IPDU.
OSIT$_TCLIMEXC	SS$_NOLINKS The maximum number of concurrent transport connections has been reached, as defined by the MAXIMUM TRANSPORT CONNECTIONS characteristic of the OSI TRANSPORT entity.
OSIT$_IVDATREQ	SS$_PROTOCOL

Reason Code	Associated Primary Status Code
	The local task has made an invalid request for user data. User data is only available with Classes 2 and 4.
OSIT$_IVEXPREQ	SS$_PROTOCOL The local task has made an invalid request for expedited data. Expedited data is only available with Classes 2 and 4.
OSIT$_IVEXTREQ	SS$_PROTOCOL The local task has made an invalid request for extended format. Extended format is only available with Classes 2 and 4.
OSIT$_IVCHKREQ	SS$_PROTOCOL The local task has made an invalid request for checksums. Checksums are only available with Class 4.
OSIT$_IVCLSREQ	SS$_PROTOCOL The local task has requested an invalid protocol class. For example, Class 0 over CLNS, or Class 3 (which is not supported by OSI transport).
OSIT$_RRJDUPREF	SS$_PROTOCOL The remote system has rejected the connection and has returned OSI reason code 83; see *Section A.2, "OSI Reason Codes"* for more information.
OSIT$_RRJMISREF	SS$_PROTOCOL The remote system has rejected the connection and has returned OSI reason code 84; see *Section A.2, "OSI Reason Codes"* for more information.
OSIT$_RRJPROT	SS$_PROTOCOL The remote system has rejected the connection and has returned OSI reason code 85; see *Section A.2, "OSI Reason Codes"* for more information.
OSIT$_RRJREFUSED	SS$_PROTOCOL The remote system has rejected the connection and has returned OSI reason code 88; see *Section A.2, "OSI Reason Codes"* for more information.
OSIT$_RRJTPDU	SS$_PROTOCOL The remote system has rejected the connection and has returned OSI reason code 8A; see *Section A.2, "OSI Reason Codes"* for more information.
OSIT$_RRJUNSP	SS$_REJECT

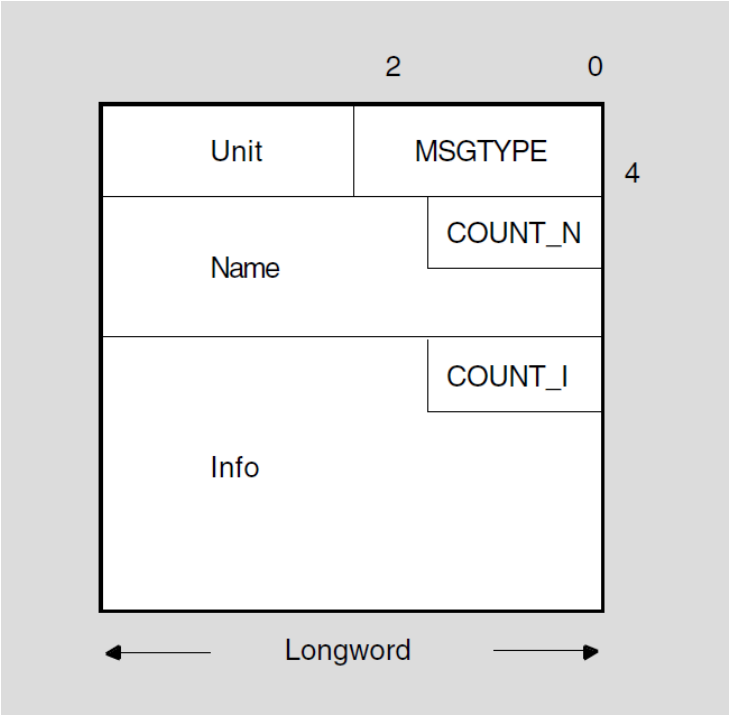
Reason Code	Associated Primary Status Code
	The remote user has rejected the Connection Request or the Connection Confirm. No reason for the rejection was given.
OSIT$_RRJNOR	SS$_REJECT The remote user has rejected the Connection Request or the Connection Confirm. This is a normal disconnection, probably at the request of the remote application. This could happen if incorrect access control information was used.
OSIT$_RRJNEG	SS$_REJECT The remote user has rejected the Connection Request or the Connection Confirm. A protocol class, option or TPDU size was proposed that was unacceptable to the remote user.
OSIT$_RRJCON	SS$_REMSRC The remote system does not have enough resources to process the call, due to congestion.
OSIT$_RRJREF	SS$_REMSRC The remote system does not have enough resources to process the call, due to reference overflow.
OSIT$_RRJTSAP	SS$_REMSRC The remote system does not have enough resources to process the call, due to congestion at the TSAP.
OSIT$_TRANDIS	SS$_SHUT TRANSPORT is disabled at the local system.
OSIT$_OUTCONDIS	SS$_SHUT Outbound transport connections at the local system are disabled.
OSIT$_NCLIMIT	SS$_UNREACHABLE The network connection limit has been exceeded.
OSIT$_NOROUREC	SS$_UNREACHABLE There is no routing information for the remote NSAP specified.
OSIT$_NOSUCHSNAP	SS$_UNREACHABLE The adjacent system specified for the remote NSAP does not exist.
OSIT$_SNAPDIS	SS$_UNREACHABLE

Reason Code	Associated Primary Status Code
	The adjacent system specified for the remote NSAP specifies a SNAP that is disabled.

Appendix B. Mailbox Message Types

A mailbox message has the general format shown in *Figure B.1, "Format of a Mailbox Message"*.

Figure B.1. Format of a Mailbox Message



MSGTYPE

A code that identifies the message type. See *Table B.1, "Mailbox Message Summary"* for a summary of the mailbox message types.

UNIT

The binary unit number of the device to which the message applies.

NAME and COUNT_N

A byte-counted string giving the name of the device to which the message applies. This field, together with the UNIT field, forms a specification for the device associated with the mailbox.

INFO and COUNT_I

A byte-counted string of information; the length of the string depends on the message type. The first byte in the string is the byte count. *Table B.2, "Contents of the INFO Field in a Mailbox Message"* describes the contents of this field as it relates to the message types.

Table B.1. Mailbox Message Summary

Message Type	Category and Meaning of Mailbox Message
Connection Setup and Conclusion Messages	

Message Type	Category and Meaning of Mailbox Message
MSG$_ABORT	The transport connection was disconnected normally.
MSG$_CONNECT	The task has received a connection request.
MSG$_CONFIRM	The transport connection was accepted and confirmed.
MSG$_PATHLOST	The network connection was lost.
MSG$_PROTOCOL	The transport connection was disconnected due to a protocol error.
MSG$_REJECT	The transport connection was rejected.
MSG$_TIMEOUT	The transport connection was disconnected because the remote host did not respond within the permitted time.
Expedited Data Message	
MSG$_INTMSG	Expedited data was received.
Connection Status Messages	
MSG$_NETSHUT	The system manager is closing down the network.
MSG$_PATHLOST	The remote task became inaccessible before the input/output operation finished.
MSG$_PROTOCOL	There is a protocol or software problem.
MSG$_THIRDPARTY	A third party ended the transport connection.
MSG$_TIMEOUT	The transport connection request timed out.

Table B.2. Contents of the INFO Field in a Mailbox Message

Message Type	Contents of INFO Field
MSG$_ABORT	This is a byte-counted string with up to 64 bytes of data. This string is the user data sent with the disconnection request. The first byte is the byte count.
MSG$_CONNECT	The whole network connect block (NCB).
MSG$_CONFIRM	A byte-counted string with up to 32 bytes of data. This string is the user data that came in with the connection indication. The first byte is the byte count.
MSG$_INTMSG	A byte-counted string with up to 16 bytes of data. This string is the expedited data sent with the $QIO call, and which OSI transport delivers as part of a mailbox message. The first byte is the byte count.
MSG$_PATHLOST	None.
MSG$_PROTOCOL	None.
MSG$_TIMEOUT	None.

Appendix C. Structure of an IOSB

The structure of an input/output status block (IOSB) returned by OSI transport depends on several factors, including the $QIO(W) call used, the completion code and whether an item list was supplied with the call.

OSI transport uses six different IOSB structures with $QIO(W) calls. Figure C–1 to Figure C–6 show the structure of these IOSBs. *Table C.1, "Guide to IOSB Structure for $QIO Calls"* is a guide to which figures you should consult for a particular $QIO(W) call.

Table C.1. Guide to IOSB Structure for $QIO Calls

$QIO(W) Calls	**SS$_ NORMAL**	**SS$_ BADPARAM**	**SS$_ LINKABORT**	**All other codes**
IO$_ACCESS (ItemList)	*Figure C.1, "IOSB for Successful $QIO(W) Call with Item List"*	*Figure C.4, "IOSB for $QIO(W) Call with Input Item List Error"*	*Figure C.6, "IOSB for Unsuccessful $QIO(W) Call"*	*Figure C.6, "IOSB for Unsuccessful $QIO(W) Call"*
IO$_ACCESS (No Item List)	*Figure C.2, "IOSB for Successful $QIO(W) Call with No Item List"*	*Figure C.6, "IOSB for Unsuccessful $QIO(W) Call"*	*Figure C.6, "IOSB for Unsuccessful $QIO(W) Call"*	*Figure C.6, "IOSB for Unsuccessful $QIO(W) Call"*
IO$_SENSEMODE	*Figure C.1, "IOSB for Successful $QIO(W) Call with Item List"*	*Figure C.4, "IOSB for $QIO(W) Call with Input Item List Error"*	*Figure C.6, "IOSB for Unsuccessful $QIO(W) Call"*	*Figure C.6, "IOSB for Unsuccessful $QIO(W) Call"*
IO $_ACPCONTROL	*Figure C.2, "IOSB for Successful $QIO(W) Call with No Item List"*	*Figure C.6, "IOSB for Unsuccessful $QIO(W) Call"*	*Figure C.6, "IOSB for Unsuccessful $QIO(W) Call"*	*Figure C.6, "IOSB for Unsuccessful $QIO(W) Call"*
IO$_DEACCESS	*Figure C.2, "IOSB for Successful $QIO(W) Call with No Item List"*	*Figure C.6, "IOSB for Unsuccessful $QIO(W) Call"*	*Figure C.6, "IOSB for Unsuccessful $QIO(W) Call"*	*Figure C.6, "IOSB for Unsuccessful $QIO(W) Call"*
IO$_READVBLK IO$_WRITEVBLK	*Figure C.3, "IOSB for Successful Read and Write $QIO(W) Calls"*	*Figure C.6, "IOSB for Unsuccessful $QIO(W) Call"*	*Figure C.5, "IOSB with SS $LINKABORT for Unsuccessful Read/Write $QIO(W)Calls"*	*Figure C.6, "IOSB for Unsuccessful $QIO(W) Call"*

C.1. IOSB for Successful $QIO(W) Calls

A successful call is one that has the status code SS$_NORMAL in the first word of the IOSB.

OSI transport uses a different IOSB structure for each of these types of successful calls:

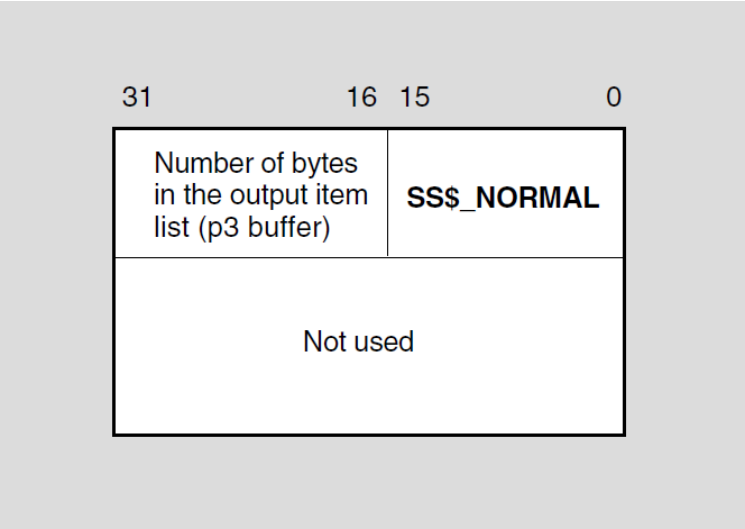
- $QIO(W) calls using item lists
- $QIO(W)(IO$_READVBLK) and $QIO(W)(IO$WRITEVBLK) calls

- Other $QIO(W) calls not using item lists

C.1.1. Successful $QIO(W) Call with Item List

Figure C.1, "IOSB for Successful $QIO(W) Call with Item List" shows the IOSB if a $QIO(W) call with an item list completes successfully.

Figure C.1. IOSB for Successful $QIO(W) Call with Item List



The first word contains the status code SS$_NORMAL.

The second word contains the number of bytes in the *p3* buffer, that is, the output item list.

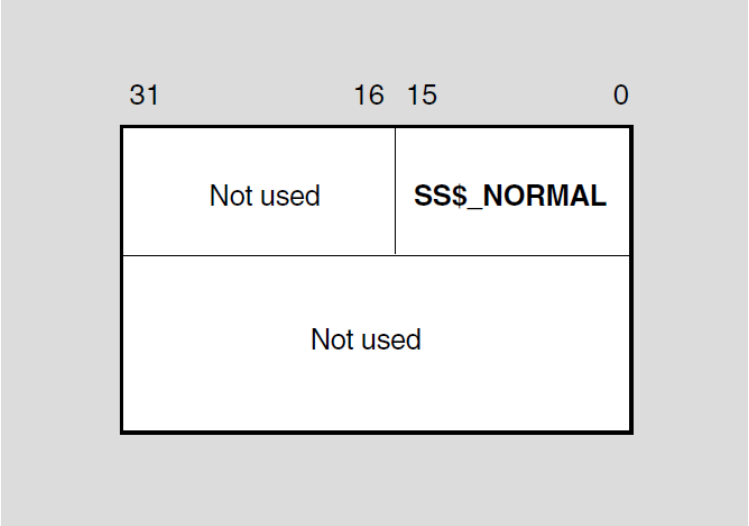
The third and fourth words are not used.

C.1.2. Successful $QIO(W) Call with No Item List

Figure C.2, "IOSB for Successful $QIO(W) Call with No Item List" shows the IOSB when any of these calls complete successfully:

- $QIO(W)(IO_ACCESS) and $QIO(W)(IO_ACCESS!ABORT) with NCB
- $QIO(W)(IO_ACPCONTROL)
- $QIO(W)(IO_DEACCESS)

Figure C.2. IOSB for Successful $QIO(W) Call with No Item List



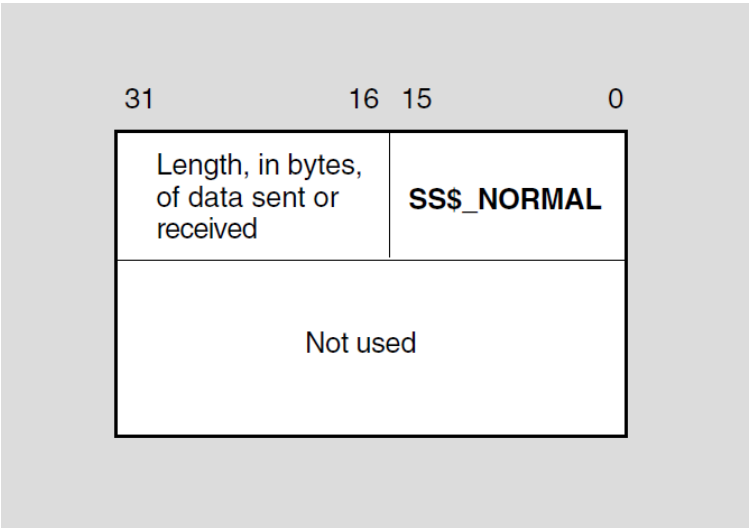
The first word contains the status code SS$_NORMAL.

The second, third and fourth words are not used.

C.1.3. Successful $QIO(W) Read and Write Calls

Figure C.3, "IOSB for Successful Read and Write $QIO(W) Calls" shows the IOSB when a $QIO(W)(IO $_READVBLK) or $QIO(W)(IO$_WRITEVBLK) call completes successfully.

Figure C.3. IOSB for Successful Read and Write $QIO(W) Calls



The first word contains the status code SS$_NORMAL.

The second word contains the length in bytes of the data sent or received.

The third and fourth words are not used.

C.2. IOSB for Unsuccessful $QIO(W) Calls

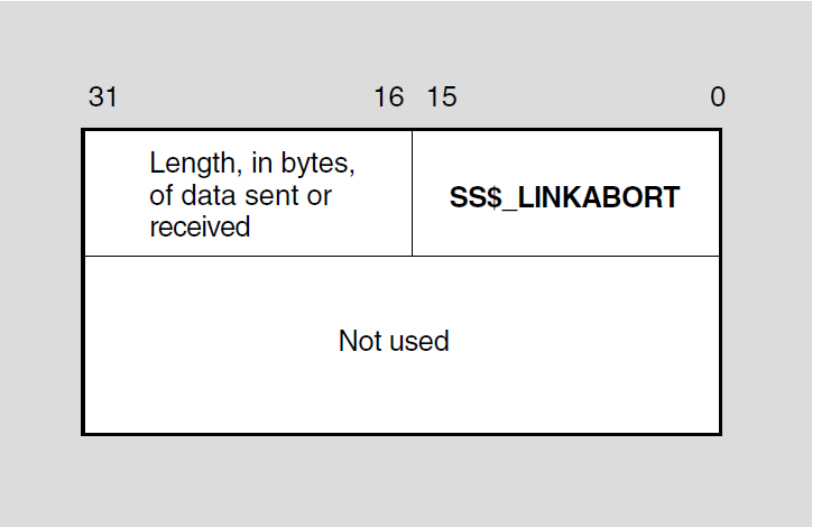
An unsuccessful call is one that fails with an error or failure status code in the first word of the IOSB. OSI transport uses different IOSB structures for:

- $QIO(W) calls with an input item list error
- QIO(W)(IO$_READVBLK) and QIO(W)(IO$_WRITEVBLK) calls that fail with SS $_LINKABORT in the first word of the IOSB
- All other $QIO(W) call failures

C.2.1. Unsuccessful $QIO(W) Call with Input Item List Error

Figure C.4, "IOSB for $QIO(W) Call with Input Item List Error" shows the IOSB if a $QIO(W) call fails because of an error in the input item list.

Figure C.4. IOSB for $QIO(W) Call with Input Item List Error



The first word contains the status code SS$_BADPARAM.

The second word contains the offset to the item with the error in the input item list.

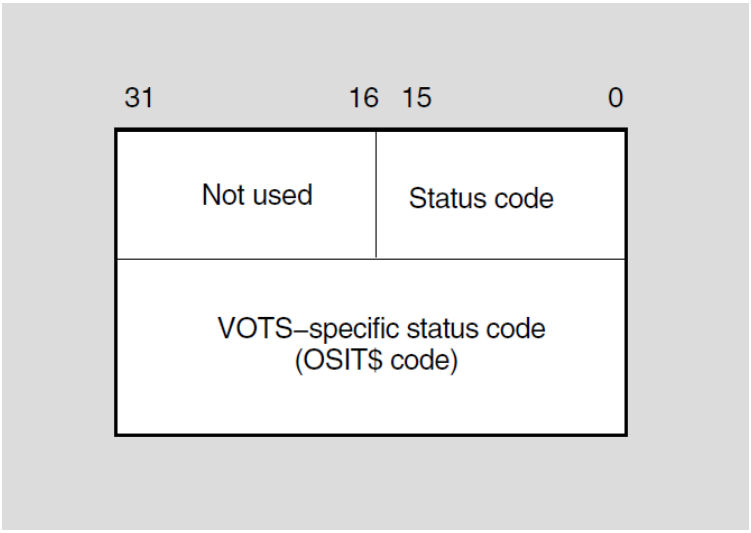
The third word contains the item type of the incorrect item.

The fourth word is not used.

C.2.2. Unsuccessful Read or Write $QIO(W) Call

Figure C.5, "IOSB with SS$LINKABORT for Unsuccessful Read/Write $QIO(W)Calls" shows the IOSB when a $QIO(W)(IO$_READVBLK) or $QIO(W)(IO$_WRITEVBLK) call completes unsuccessfully, with SS$_LINKABORT in the first word.

Figure C.5. IOSB with SS$LINKABORT for Unsuccessful Read/Write $QIO(W)Calls



The first word contains SS$_LINKABORT.

The second word contains the length in bytes of the data sent or received.

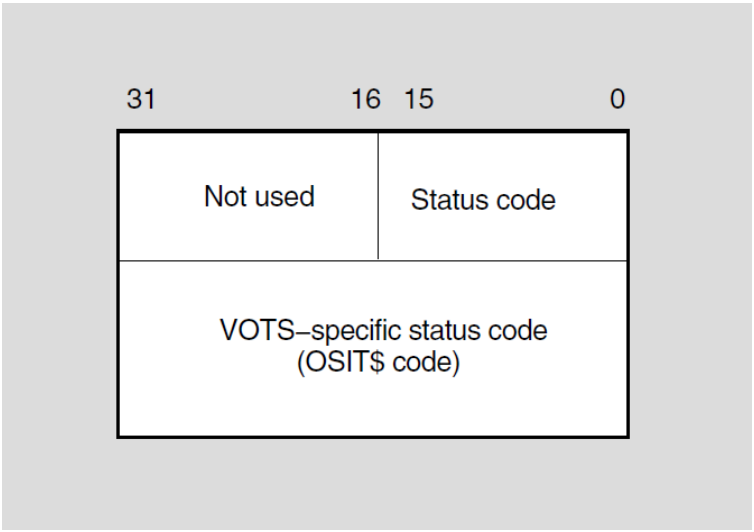
The third and fourth words are not used.

C.2.3. All Other Unsuccessful $QIO(W) Calls

Figure C.6, "IOSB for Unsuccessful $QIO(W) Call" shows the IOSB for these unsuccessful $QIO(W) calls:

- Unsuccessful $QIO(W) call with no item list (except read and write calls)
- Unsuccessful $QIO(W) call using an item list, where the status code in the first word is not SS $_BADPARAM
- Unsuccessful $QIO(W)(IO$_READVBLK) and $QIO(W) (IO$_READVBLK) calls where the status code in the first work is not SS$_LINKABORT

Figure C.6. IOSB for Unsuccessful $QIO(W) Call



The first word contains the status code.

The second word is not used.

The third and fourth words contain a OSI transport-specific status code with the prefix OSIT$.This gives a precise reason why OSI transport has returned an error status. See *Appendix A, "$QIO(W) Status Codes and OSI Reason Codes"* for a list of these codes.

Appendix D. LIB$PARSE_NCB

LIB$PARSE_NCB parses a network connect block (NCB) and produces a network item list. NCBs are used by DECnet and OSI transport to inform users of an inbound connection request.

FORMAT	**LIB$PARSE_NCB** *ncb,itemlist,len*		
RETURNS	OpenVMS usage:	**cond_value**	
	type:	**longword (unsigned)**	
	access:	**write only**	
	mechanism:	**by value**	
ARGUMENTS	*ncb*		
	OpenVMS usage:	**char_string**	
	type:	**character string**	
	access:	**read only**	
	mechanism:	**by descriptor**	
	Network Connect Block string that is to be parsed. The *ncb* argument contains the address of a descriptor pointing to this NCB.		
	itemlist		
	OpenVMS usage:	**char_string**	
	type:	**character string**	
	access:	**write only**	
	mechanism:	**by descriptor**	
	Destination string into which LIB$PARSE_NCB writes the item list it has produced. The *itemlist* argument contains the address of a descriptor pointing to this string.		
	len		
	OpenVMS usage:	**word_unsigned**	
	type:	**word integer (unsigned)**	
	access:	**write only**	
	mechanism:	**by reference**	
	Length of the item list built by LIB$PARSE_NCB. The *len* argument contains the address of an unsigned word to receive the length of the item list.		
CONDITION VALUES RETURNED	SS$_NORMAL		The NCB has been successfully parsed.
	LIB$_SYNTAXERR		There is a syntax error in the NCB.

	SS$_TOOMUCHDATA	The NCB has been successfully parsed but there is not enough room in the item list for all of the items generated.

Appendix E. Programming Examples

The following example programs are contained in the directory SYS$EXAMPLES:

File	Function
OSIT$RANDOM.C	Example program in the C language
OSIT$TRANSMITTER.PAS OSIT$RECEIVER.PAS	Example programs in the Pascal language
OSIT$CMD_EXECUTOR.MAR OSIT$CMD_SOURCE.MAR OSIT$CMD_SOURCE.CLD OSIT$CMD_EXECUTOR.COM	Example programs in the MACRO language
OSIT$ECHO.FOR OSIT$STORAGE.FOR OSIT$RECORD_STRUCTURES.FOR	Example programs in the FORTRAN language

E.1. Example Program in the C Language

This section contains an example of a program using OSI transport. The program is written in the C language.

Each subsection contains an individual program module, as shown in *Table E.1, "Example Programs"*:

Table E.1. Example Programs

Section Number	Content of Routine	Routine Name
Section E.1.1, "Introduction and Data Structures"	Introduction and data structures	
Section E.1.2, "Translation of SYS$NET"	Translation of SYS$NET	main
Section E.1.3, "Routine Called for Initiator"	Routine called for initiator	initiator
Section E.1.4, "Routine Called for Responder"	Routine called for responder	responder
Section E.1.5, "AST Routine to Check Status of Outbound Connection Request"	AST routine to check status of outbound connection request	request_ast
Section E.1.6, "Initiate Outbound Connection Request"	Initiate outbound connection request	request
Section E.1.7, "Assign a Channel to OSI Transport"	Assign a channel to OSI Transport	assign

Section Number	Content of Routine	Routine Name
Section E.1.8, "Create Mailbox and Post a Read"	Create mailbox and post a read	create_mailbox
Section E.1.9, "Deassign a Channel"	Deassign a channel	deassign
Section E.1.10, "Check Status of Disconnection"	Check status of disconnection	disconnect_ast
Section E.1.11, "Disconnect Current Transport Connection"	Disconnect current transport connection	disconnect
Section E.1.12, "Free Write Buffer When Write Request Completes"	Free write buffer when write request completes	write_ast
Section E.1.13, "Send Data on the Transport Connection"	Send data on the transport connection	writedata
Section E.1.14, "Disconnect After Read Is Complete"	Disconnect after read is complete	read_ast
Section E.1.15, "Read Data"	Read data	readdata
Section E.1.16, "Check Acceptance of Inbound Connection"	Check acceptance of inbound connection	accept_ast
Section E.1.17, "Accept an Inbound Connection"	Accept an inbound connection	accept
Section E.1.18, "Build Input Item List"	Build input item list	add_string_item
Section E.1.19, "Analyze NCB and Build Input Item List"	Analyze NCB and build input item list	bldlst_NCB
Section E.1.20, "Build Input Item List for a Connection Request"	Build input item list for a connection request	bldlst_connect
Section E.1.21, "Display Output Item List"	Display output item list	display_list
Section E.1.22, "Displays a Specified Item"	Display a specified item	display_item
Section E.1.23, "Report $QIO Error"	Report $QIO error	report_error
Section E.1.24, "Read Mailbox"	Read mailbox	post_mailbox_read
Section E.1.25, "Report Mailbox Message Type"	Report mailbox message type	process_mailbox_message(mail)
Section E.1.26, "Wait for Mailbox Message and Read Mailbox"	Wait for mailbox message and read mailbox	mailbox_ast(mail)

E.1.1. Introduction and Data Structures

This part of the program briefly describes its purpose, and shows the data structures. The list of routines called is in the TABLE OF CONTENTS in the program.

```
/*
 * random
```

```
 *
 * ABSTRACT:
 *
 * This program is an example program using vots.
 * It runs in one of two modes, either as the initiator
 * or, as the responder.
 *
 * The initiator requests a connection to with the test
 * node TEST, to the tsap RANDOM. Once the connection
 * has been established, it expects 1,000,000 bytes of
 * data to be transmitted by the test responder in
 * random sized buffers (not greater than 2000 bytes).
 * When all of the data has been received, the test
 * initiator disconnects.
 *
 * The test responder acts as a passive TSAP and only ever
 * has one transport connection. The test initiator has
 * many transport connections and is multi-threaded.
 *
 * ENVIRONMENT:
 * VAX/VMS V5.0
 *
 * MODIFICATION HISTORY:
 *
 * X-2 PEY0000        Paul Yager         17-Oct-1994
 * Remove display of unused IOSB 3rd word.
 */

/*
 * INCLUDE FILES:
 */
#include <iodef.h>
#include <descrip.h>
#include <stsdef.h>
#include <ssdef.h>
#include <dvidef.h>
#include <msgdef.h>
#include <psldef.h>
#include <stdio.h>
#include "sys$library:osit"
#include "sys$library:lnmdef"
#include ctype

/*
 * EQUATED SYMBOLS:
 */

#define TRUE 1
#define FALSE 0

/*
** AST Enable/Disable flags
*/

#define BLOCK_AST 0
#define ENABLE_AST 1

/*
```

```
 * Event Flag allocation
 */

#define SYNC_EFN 1   /* EFN for synchronous requests */
#define ASYNC_EFN 0   /* EFN for asynchronous requests */
#define BREAK_EFN 2   /* EFN for disconnection requests */

/*
 * MACROS:
 */

#define $error(test) (!((test) & STS$K_SUCCESS))

/*
 * OWN STORAGE:
 */
/*
 * the number of test streams active (initiator only)
 */

unsigned int random$gl_tests = 0;

/*
 * information about the mailbox, there is only one mailbox used,
 * it may be shared by several transport connections
 */

#define RANDOM$K_MBX_NAMLEN 64
#define RANDOM$K_MBX_MSG_SIZE OSIT$K_MAX_NCB+5
char mbx_name_text [RANDOM$K_MBX_NAMLEN];
$DESCRIPTOR (mbx_name, mbx_name_text);

struct mailblock
 {
 unsigned int channel;
 short int iosb [4];
 struct
  {
  short int type; /* Message code */
  short int unit; /* Device unit number */
  unsigned char nct; /* Number of bytes in device name */
  unsigned char nam [OSIT$K_MAX_NCB]; /* Address of device name string */
  } msg;
 };
/*
 * Structure of test block, this contains all of the information relevent
 * to a particular transport connection or test
 */
#define RANDOM$K_TEST_COUNT 3
struct testblock
 {
 struct testblock *next;
 struct
  {
  unsigned wfdis :1;
  unsigned initiator :1;
  unsigned unused :14;
  } flags;
```

```
 short int channel;
 short int iosb [4];
 short int unit;
 unsigned int tx_total;
 unsigned int tx;
 unsigned int rx_total;
 unsigned int rx;
 struct dsc$descriptor_s in_desc;
 struct dsc$descriptor_s out_desc;

 unsigned char input_list [OSIT$K_MAX_OUTPUT_ITEM_LIST];
 unsigned char output_list [OSIT$K_MAX_OUTPUT_ITEM_LIST];
 };

/*
 * structure of a data block, every read or write has one of
 * these
 */

#define RANDOM$K_MAX_DATA_SIZE 3000
#define RANDOM$K_TOTAL_DATA 15000
struct datablock
 {
 short int iosb [4];
 struct testblock *test;
 unsigned char data [RANDOM$K_MAX_DATA_SIZE];
 };
/*
 * Mailbox messge format.
 */

struct msg
 {
 short int type;    /* Message code */
 short int unit;    /* Device unit number */
 unsigned char nct;    /* Number of bytes in device name */
 unsigned char nam [64];    /* Address of device name string */
 };

/*
 * global variables
 */
int zero = 0;
int status;
struct testblock *random$ga_tests;

$DESCRIPTOR (osi, "OSIT$DEVICE");

/*
 * TABLE OF CONTENTS:
 */
unsigned int main (),
  initiator (),
  responder (),
  request(),
  create_mailbox(),
  accept(),
  process_mailbox_message();
```

```
void request_ast(),
  bldlst_connect(),
  post_mailbox_read(),
  display_list(),
  report_error(),
  display_item(),
  disconnect(),
  bldlst_ncb(),
  readdata(),
  writedata(),
  accept_ast(),
  write_ast (),
  read_ast (),
  disconnect_ast(),
  mailbox_ast(),
  deassign();

struct testblock *assign();
unsigned int *add_string_item();

/*
 * EXTERNAL REFERENCES:
 */

int SYS$QIO (),
 SYS$QIOW (),
 SYS$ASSIGN (),
 SYS$CREMBX(),
 SYS$SETSFM (),
 SYS$GETDVIW (),
 SYS$DASSGN (),
 SYS$HIBER (),
 SYS$SETAST (),
 SYS$WAKE (),
 SYS$TRNLNM ();

int time(),
 free(),
 rand(),
 srand();

void    *malloc();

int LIB$SIGNAL (),
 LIB$STOP (),
 LIB$ASN_WTH_MBX (),
 LIB$MOVC3 (),
 LIB$WAIT (),
 LIB$PARSE_NCB ();
```

E.1.2. Translation of SYS$NET

```
/* ------------------------------------------------------------------------ */

unsigned int main ()

/* ------------------------------------------------------------------------ */
```

```
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine tries to translate sys$net - if it can, it assumes
 * that it's an the test responder, otherwise it assumes that
 * it's the test initiator
 *
 * FORMAL PARAMETERS:
 *
 * None.
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * SS$_NORMAL or whatever is returned by initiator,
 * responder or TNRLOG.
 *
 * SIDE EFFECTS:
 *
 * None.
*/
 {
 unsigned int status;
 static char ncbbuf [OSIT$K_MAX_NCB];
 $DESCRIPTOR (ncb, ncbbuf);
 unsigned int attr;
 $DESCRIPTOR (lognam, "SYS$NET");
 $DESCRIPTOR (tabnam, "LNM$PROCESS_TABLE");
 struct
  {
  unsigned short int bufsiz;
  unsigned short int itmcod;
  char *bufadr;
  unsigned int retlen;
  unsigned int endoflist;
  } itmlst;
/*
 * initialise
 */
 status = SS$_NORMAL;
 random$ga_tests = 0;
/*
 * announce who we are
 */
 printf("RANDOM [1.0]\n");
/*
 * assign a mailbox
 */
 if (!($error(status = create_mailbox())))
```

```
  {
/*
 * Obtain the NCB from a translation of SYS$NET
 */

  attr = LNM$M_CASE_BLIND;
  itmlst.bufsiz = OSIT$K_MAX_NCB;
  itmlst.itmcod = LNM$_STRING;
  itmlst.bufadr = ncb.dsc$a_pointer;
  itmlst.retlen = (unsigned int)&(ncb.dsc$w_length);
  status = SYS$TRNLNM (&attr, &tabnam, &lognam, 0, &itmlst);

/*
 * call either the test initiator or the test responder, depending
 * on whether there was a translation or not
 */
  if $error (status)
   {
   if (status == SS$_NOLOGNAM)
    {
    printf("role - test initiator\n");
     status = initiator ();
    }
   else
    printf("\nerror: translation of logical name sys$net failed\n");
   }
  else
   {
   printf("role - test responder\n");
   status = responder (&ncb);
   }  /* end else passive tsap */

/*
 * if we get to this point, and everything went well, then just
 * sleep, this test program works asynchronously and is multi-threaded,
 * ast routines do all of the work
 */
  if (!($error (status)))
   SYS$HIBER ();
  }  /* end else mailbox created */
/*
 * return the status to the caller
 */
 return status;
 }     /* end routine main */
```

E.1.3. Routine Called for Initiator

```
/* ------------------------------------------------------------------------ */

unsigned int initiator ()

/* ------------------------------------------------------------------------ */

/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine is called when the program is acting in the role
```

```
 * of test initiator, it assigns a channel to VOTS, which allocates
 * a test block and makes an outbound connection request to the
 * test responder. That completes when the ast routine fires, the
 * tester then waits for data from the test responder.
 *
 * FORMAL PARAMETERS:
 *
 * None.
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */
 {
 unsigned int i, status;
 struct testblock *test;

 for (i=0;i<RANDOM$K_TEST_COUNT;i++)
  if ((test = assign()) == 0)
   status = SS$_INSFMEM;
  else
   {
   test->flags.initiator = TRUE;
   /*
   ** Block AST's while updating these structures.
   */
   SYS$SETAST(BLOCK_AST);
   random$gl_tests++;
   test->next = random$ga_tests;
   random$ga_tests = test;
   status = request(test);
   SYS$SETAST(ENABLE_AST);
   }
/*
 * return the status to the caller
 */
 return (status);
 }     /* end routine initiator */
/
```

E.1.4. Routine Called for Responder

```
/* ------------------------------------------------------------------------ */

signed int responder (ncb)
```

```
/* ------------------------------------------------------------------------ */

struct dsc$descriptor *ncb;/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine is called when the program is acting in the role
 * of test responder, it assigns a channel to VOTS, which allocates
 * a test block and accepts the inbound connection request from the
 * test initiator. That completes when the ast routine fires, the
 * tester then transmits data to the test initiator.
 *
 * FORMAL PARAMETERS:
 *
 * ncb - descriptor of ncb gained from logical translation
 *    of SYS$NET
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */ { unsigned int status; struct testblock
 *test; long t; if ((test = assign()) == 0)  status = SS$_INSFMEM; else
/*
 * we have a channel to VOTS, initialise the random transmit
 * buffer size generator, build an input item list and accept
 * the inbound connection request
 */
  {
  test->flags.initiator = FALSE;
  t = time();
  srand (t);

  test->in_desc.dsc$b_dtype = DSC$K_DTYPE_T;
  test->in_desc.dsc$b_class = DSC$K_CLASS_S;
  test->in_desc.dsc$w_length = OSIT$K_MAX_OUTPUT_ITEM_LIST;
  test->in_desc.dsc$a_pointer = (char *)&(test->input_list);
  bldlst_ncb (&test->in_desc, ncb);
  random$ga_tests = test;  status = accept(test, &test->in_desc);
  }

/*
 * return the status to the caller
 */
 return (status);
 }     /* end routine responder */
/
```

E.1.5. AST Routine to Check Status of Outbound Connection Request

```
/* ------------------------------------------------------------------- */

unsigned int request_ast(test)
/* ------------------------------------------------------------------*/

struct testblock
 *test;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This ast routine checks the status of the outbound connection
 * request made and, if it worked, continues with the test.
 *
 * FORMAL PARAMETERS:
 *
 * test - pointer at a test block for this thread test code
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */
 {
/*
 * check to see if it worked
 */ if ($error (test->iosb [0]))
  report_error (test->iosb[0], test); else
/*
 * display the output item list and perform a read on the
 * established channel
 */
  {
  printf ("[%d] connection established\n",
      test->unit);
  printf ("\toutput item list for an outbound CR:\n");
  display_list (test->output_list, test->iosb[1]);
  readdata (test);
  }
 }     /* end routine request_ast *
```

E.1.6. Initiate Outbound Connection Request

```
/* ------------------------------------------------------------------- */
unsigned int request(test)
/* ------------------------------------------------------------------- */
struct testblock *test;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine makes an outbound transport connection request
 * to tsap RANDOM on node TEST.
 *
 * FORMAL PARAMETERS:
 *
 * test - pointer at a test block for this thread test code
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */
 {
 /*
 **  Initialize the input descriptor
 */
 test->in_desc.dsc$b_dtype = DSC$K_DTYPE_T;
 test->in_desc.dsc$b_class = DSC$K_CLASS_S;
 test->in_desc.dsc$w_length = OSIT$K_MAX_OUTPUT_ITEM_LIST;
 test->in_desc.dsc$a_pointer = (char *)&(test->input_list);
 /*
 **  Initialize the output descriptor
 */
 test->out_desc.dsc$b_dtype = DSC$K_DTYPE_T;
 test->out_desc.dsc$b_class = DSC$K_CLASS_S;
 test->out_desc.dsc$w_length = OSIT$K_MAX_OUTPUT_ITEM_LIST;
 test->out_desc.dsc$a_pointer = (char *)&(test->output_list);
/*
 *  build an input item list and make the connection request
 */
 bldlst_connect(&test->in_desc);
/*
 * now make the outbound connection request
 */
 printf ("[%d] making an outbound connection request\n",
     test->unit);
 status = SYS$QIO (0, test->channel, IO$_ACCESS,
```

```
      &(test->iosb), &request_ast, test, &test->in_desc, 0,
      &test->out_desc, 0, 0, 0);
/*
 * return the status to the caller
 */ return (status);
 }     /* end routine request */
/
```

E.1.7. Assign a Channel to OSI Transport

```
/*------------------------------------------------------------------------ */
struct testblock *assign()
/* ------------------------------------------------------------------------ */
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine allocates a test block and assigns a channel
 * to VOTS
 *
 * FORMAL PARAMETERS:
 *
 * None.
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */ { struct testblock *test;
 static unsigned int unit, l;
/*
 * GETDVI item list structure and item list
 */
 struct itmlst
  {
  short int itm_len; /* Length of buffer in bytes */
  short int itm_code; /* Item to be extracted */
  char *itm_buffer; /* Buffer to fill with information */
  int *itm_retlen; /* Address of longword for length */
  };
 struct itmlst vots_unit_itm [2] =
  {
  {4, DVI$_UNIT, (char *)&unit, (int *)&l},
  {0,0,0,0}
  };
/*
 * allocate a testblock
 */
```

```
 if ((test = malloc(sizeof(struct testblock))) == 0)
  printf("error: failed to allocate a testblock\n");
 else
  {
  printf ("[-] assigning a VOTS channel\n");
  if ($error (status = SYS$ASSIGN (&osi, &(test->channel), 0,
       &mbx_name)))
       {
       report_error (status, test);
       free (test);
           }
  else
   {
/*
 * find out the unit number of the device that we've just allocated
 * we're quoted this in any mailbox messages, so use it to figure
 * out which TC the mailbox message was for
 */
   status = SYS$GETDVIW (SYNC_EFN, test->channel, 0,
       &vots_unit_itm, 0, 0, 0, 0);
   if ($error (status))
    report_error (status, test);
   else
    {
/*
 * set up the unit number in the testblock
 */
    test->unit = unit;
/*
**  Initialize the next pointer.
*/
      test->next = NULL;
    }
   }
  }   /* end of testblock allocated */
 return (test);
 }    /* end routine assign */
/
```

E.1.8. Create Mailbox and Post a Read

```
/* ------------------------------------------------------------------------ */
unsigned int create_mailbox()
/* ------------------------------------------------------------------------ */
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine creates a mailbox and posts a read on it
 *
 * FORMAL PARAMETERS:
 *
 * None.
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
```

```
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */
 {
 struct mailblock *mail;
/*
 * GETDVI item list structure and item list
 */
 struct itmlst
  {
  short int itm_len; /* Length of buffer in bytes */
  short int itm_code; /* Item to be extracted */
  char *itm_buffer; /* Buffer to fill with information */
  int *itm_retlen; /* Address of longword for length */
  };
 struct itmlst mbx_nam_itm [2] =
  {
  {RANDOM$K_MBX_NAMLEN, DVI$_DEVNAM, (char *)&mbx_name_text,
      (int *)&mbx_name.dsc$w_length},
  {0,0,0,0}
  };
/*
 * allocate a mailbox message block
 */
 if ((mail = malloc(sizeof(struct mailblock))) == 0)
  printf("error: failed to allocate a mailblock\n");
 else
  {
  printf ("creating a mailbox\n");
  if ($error (status = SYS$CREMBX (0, &(mail->channel),
      RANDOM$K_MBX_MSG_SIZE, 0, 0, 0)))
       {
       printf("error: failed to create a mailbox\n");
       free (mail);
           }
  else
   {
   printf ("mailbox channel (%d) assigned\n",
        mail->channel);
/*
 * find out the device name of the mailbox that we've just created,
 * we use this when we allocate channels to VOTS which associate
 * themselves with that mailbox.
 */
   status = SYS$GETDVIW (SYNC_EFN, mail->channel, 0,
       &mbx_nam_itm, 0, 0, 0, 0);
   if ($error (status))
    {
    printf ("error, failed to get mailbox's name\n");
    LIB$STOP (status);
```

```
    }
   else
    {
/*
 * post a read on the associated mailbox
 */
    post_mailbox_read (mail);
    }
   }
  }   /* end of testblock allocated */
 return (status);
 }    /* end routine create_mailbox */
```

E.1.9. Deassign a Channel

```
/* ------------------------------------------------------------------------ */
unsigned int deassign(test)
/* ------------------------------------------------------------------------ */
struct testblock *test;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine deassigns a channel, it's only called by the
 * responder, so we don't need to worry about tidying up the
 * linked list of test blocks.
 *
 * FORMAL PARAMETERS:
 *
 * None.
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */
 {
 printf ("[%d] deassigning the channel to VOTS\n",
     test->unit);
 if ($error (status = SYS$DASSGN (test->channel)))
  report_error (status, test);
 free(test);
  }    /* end routine deassign *
```

E.1.10. Check Status of Disconnection

```
/* ------------------------------------------------------------------------ */
```

```
unsigned int disconnect_ast(test)
/* ------------------------------------------------------------------------ */
struct testblock *test;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This ast routine checks the status of the disconnection made
 * and, if it worked, and we're the initiator, it starts up
 * another test session
 *
 * FORMAL PARAMETERS:
 *
 * test - pointer at a test block for this thread test code
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */
 {
     unsigned int initiator_flag;
/*
 * check to see if it worked
 */
 if ($error (test->iosb[0]) && (test->iosb[0] != SS$_FILNOTACC))
  report_error (test->iosb[0], test);
 else
/*
 * deassign the channel
 */
  {
  printf ("[%d] disconnection succeeded\n",
      test->unit);
  /*
  **  Save the value of flags.initiator since the test block
  **  will be deallocated as part of the deassign call.
  */
  initiator_flag = test->flags.initiator;
  /*
  **  Remove this testblock from the linked list. First need
  **  to find the address of the linked block before this.
  */
    {
      struct testblock *prevtest;
      prevtest = random$ga_tests;
      while ((prevtest->next != test) && (prevtest->next != NULL))
      {
```

```
       prevtest = prevtest->next;
      }
      if ((prevtest != NULL) && (prevtest->next == test))
      {
       /*
   **  Unlink test from this list.
   */
     prevtest->next = test->next;
      }
      else if (test == random$ga_tests)
      {
       /*
   **  test was the first element in the list.
   */
     random$ga_tests = test->next;
      }
  }
  deassign (test);
  if (initiator_flag)
   {
   random$gl_tests-;
   if (random$gl_tests == 0)
    SYS$WAKE(0,0);
   }
  else
   SYS$WAKE(0,0);
  }
 }     /* end routine disconnect_ast */
```

E.1.11. Disconnect Current Transport Connection

```
/* ------------------------------------------------------------------------ */
unsigned int disconnect(test)
/* ------------------------------------------------------------------------ */
struct testblock *test;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 *  This routine disconnects the current TC.
 *
 * FORMAL PARAMETERS:
 *
 * test - pointer at a testblock
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
```

```
 * None.
 */
 {
/*
 * disconnect the tc
 */
 printf ("[%d] performing a disconnection\n",
     test->unit); status = SYS$QIO (0, test->channel, IO$_DEACCESS,
      &(test->iosb), &disconnect_ast, test, 0, 0, 0,
      0, 0, 0); if ($error (status))
   report_error (status, test);
  }    /* end routine disconnect */
```

E.1.12. Free Write Buffer When Write Request Completes

```
/* ------------------------------------------------------------------------ */
unsigned int write_ast (data)
/* ------------------------------------------------------------------------ */
struct datablock *data;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This function is called when a write request on a connection
 *  has been completed.  It frees off the buffer when it's finished
 * with it.
 *
 * FORMAL PARAMETERS:
 *
  * data - pointer at a data buffer
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 *
 */
 {
 struct testblock *test;

 test = data->test;
 if ($error (data->iosb [0]))
  report_error (data->iosb[0], test);
 else
  {
  test->tx_total = test->tx_total + data->iosb [1];
```

```
  test->tx++;
  free (data);
/*
 * if we've sent all of the data, then just set the waiting for
 * disconnection flag and wait for a mailbox message telling
 * us that the connection has gone away
 */
  if (test->tx_total >= RANDOM$K_TOTAL_DATA)
   {
   printf ("[%d] %d blocks of average size %d transmitted\n",
       test->unit, test->tx,
       test->tx_total/test->tx);
   test->flags.wfdis = TRUE;
   }
  else
   writedata (test);
  }
 }    /* end routine write_ast *
```

E.1.13. Send Data on the Transport Connection

```
/* ------------------------------------------------------------------------ */
unsigned int writedata(test)
/* ------------------------------------------------------------------------ */
struct testblock *test;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine sends a random sized block of data on the TC
 *
 * FORMAL PARAMETERS:
 *
 * test - pointer at a testblock
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */
 {
 struct datablock *data;
 unsigned int size;

 if ((data = malloc(sizeof(struct datablock))) == 0)
  {
  printf("[%d] error: failed to allocate a write block\n",
      test->unit);
```

```
  report_error (SS$_INSFMEM, test);
  }
 else
  {
  data->test = test;
  size = rand () % RANDOM$K_MAX_DATA_SIZE;
  if ($error (status = SYS$QIO (ASYNC_EFN, test->channel,
       IO$_WRITEVBLK, &(data->iosb), &write_ast, data,
       &(data->data), size, 0, 0, 0, 0)))
   report_error (status, test);
  }   /* end else allocated a data block */
 }    /* end routine writedata */
```

E.1.14. Disconnect After Read Is Complete

```
/* ------------------------------------------------------------------------ */
unsigned int read_ast (data)
/* ------------------------------------------------------------------------ */
struct datablock *data;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This function is called when a read request on a connection
 *  has been completed. If it completed successfully, it increments
 * the read data count, and if all of the data has been received,
 * it disconnects the connection.
 *
 * FORMAL PARAMETERS:
 *
 * data - pointer at a datablock containing the iosb and
 *    the data read
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 *
 */
 {
 struct testblock *test;
 test = data->test;
 if ($error (data->iosb [0]))
  report_error (data->iosb[0], test);
 else
  {
  test->rx_total = test->rx_total + data->iosb [1];
  test->rx++;
```

```
  free (data);
/*
 * if that was the last of the data to be received, then print
 * the results and disconnect
 */
  if (test->rx_total >= RANDOM$K_TOTAL_DATA)
   {
   printf ("[%d] %d blocks of average size %d received\n",
       test->unit, test->rx,
       test->rx_total/test->rx);
   disconnect (test);
   }
  else
   readdata (test);
  }
 }    /* end routine read_ast */
/
```

E.1.15. Read Data

```
/* ------------------------------------------------------------------------ */
unsigned int readdata(test)
/* ------------------------------------------------------------------------ */
struct testblock *test;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine reads a block of data from the TC.
 *
 * FORMAL PARAMETERS:
 *
 * test - pointer at the test description block to use
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */
 {
 struct datablock *data;
 if ((data = malloc(sizeof(struct datablock))) == 0)
  {
  printf("[%d] error: failed to allocate a read block\n",
      test->unit);
  report_error (SS$_INSFMEM, test);
  }
 else
```

```
  {
  data->test = test;
  if ($error (status = SYS$QIO (ASYNC_EFN, test->channel,
       IO$_READVBLK, &(data->iosb), &read_ast, data,
       &(data->data), RANDOM$K_MAX_DATA_SIZE, 0, 0, 0)))
   report_error (status, test);
  }   /* end else allocated a data block */
 }    /* end routine readdata *
```

E.1.16. Check Acceptance of Inbound Connection

```
/* ------------------------------------------------------------------------ */
unsigned int accept_ast(test)
/* ------------------------------------------------------------------------ */
struct testblock *test;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This ast routine checks the status of the connection
 * acceptence made and, if it worked, continues with the test.
 *
 * FORMAL PARAMETERS:
 *
 * test - pointer at a test block for this thread test code
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */
 {
/*
 * check to see if it worked
 */
 if ($error (test->iosb [0]))
  report_error (test->iosb[0], test);
 else
/*
 * display the output item list produced and perform a
 * write on the established channel
 */
  {
  printf("[%d] inbound connection accepted\n",
      test->unit);
  printf ("\toutput item list for an inbound CR:\n");
  display_list (test->output_list, test->iosb[1]);
  writedata (test);
```

```
  }
 }    /* end routine accept_ast */
```

E.1.17. Accept an Inbound Connection

```
/* ------------------------------------------------------------------------ */
unsigned int accept(test, in_desc)
/* ------------------------------------------------------------------------ */
struct testblock *test;
struct dsc$descriptor *in_desc;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 *  This routine accepts an inbound connection
 *
 * FORMAL PARAMETERS:
 *
 * test - pointer a testblock
 * in_desc - address of a descriptor of an item list describing the
 *    inbound connection request
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */
 {
/*
 *  accept the connection request
 */
 test->out_desc.dsc$b_dtype = DSC$K_DTYPE_T;
 test->out_desc.dsc$b_class = DSC$K_CLASS_S;
 test->out_desc.dsc$w_length = OSIT$K_MAX_OUTPUT_ITEM_LIST;
 test->out_desc.dsc$a_pointer = (char *)&(test->output_list);
 printf ("[%d] accepting the inbound connection request\n",
     test->unit);
 status = SYS$QIO (0, test->channel, IO$_ACCESS,
     &(test->iosb), &accept_ast, test, in_desc, 0, &test->out_desc,
     0, 0, 0);
/*
 * return the status to the caller
 */
 return (status);
 }    /* end routine accept */
/
```

E.1.18. Build Input Item List

```
/* ------------------------------------------------------------------------ */
unsigned int *add_string_item(list, code, string, size)
/* ------------------------------------------------------------------------ */
unsigned char *list,*string;
unsigned int code, size;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine builds an input item list
 *
 * FORMAL PARAMETERS:
 *
 * list - a pointer to the list that we are building
 * code - the item code of the item that we're adding
 * string  - a pointer to the string that we are adding
 * size - the size of "string" in bytes
 *
 * IMPLICIT INPUTS:
 *
 * in_desc and input_list
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * A pointer to the first byte following the item just added
 *
 * SIDE EFFECTS:
 *
 * None.
 */
    {
    struct _osit_item *item;  /* pointer at an individual item */
    item = (struct _osit_item *)list;
    item -> osit$w_item_type = code;
    item -> osit$w_item_length = OSIT$K_ITEM_HEADER_SIZE + size;
    LIB$MOVC3 (&size,string,&(item->osit$r_item_value.osit
$t_item_string[0]));
    return (unsigned int *)((unsigned int)list + size +
                            OSIT$K_ITEM_HEADER_SIZE);
    }     /* end routine add_string_item */
```

E.1.19. Analyze NCB and Build Input Item List

```
/* ------------------------------------------------------------------------ */
unsigned int bldlst_ncb(in_desc, ncb)
/* ------------------------------------------------------------------------ */
struct dsc$descriptor *in_desc, *ncb;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine builds an input item list containing an NCB
 *
```

```
 * FORMAL PARAMETERS:
 *
 * None.
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */
 {
 unsigned short int ncblist_length;
 unsigned int status;
 struct _osit_item *item;  /* pointer at an individual item */

/* add in the ncb */

 status =
      LIB$PARSE_NCB
  (
  ncb,
   in_desc,
   &ncblist_length
  );
 if $error (status)
   LIB$STOP (status);
/* add in the protocol version - mandatory parameter */
 item = (struct _osit_item *)((unsigned int)in_desc->dsc$a_pointer +
                                      ncblist_length);
 item -> osit$w_item_length = OSIT$K_ITEM_HEADER_SIZE + 4;
 item -> osit$w_item_type = OSIT$K_ITEM_PROTOCOL_TYPE;
 item -> osit$r_item_value.osit$l_item_long = OSIT$K_OSI_PROTOCOL;

 /* set up the length in the input item list */
 in_desc->dsc$w_length
   = ncblist_length + OSIT$K_ITEM_HEADER_SIZE + 4;
/* display the list */

 printf ("\tinput item list to accept an inbound CR:\n");
 display_list (in_desc->dsc$a_pointer, in_desc->dsc$w_length);
 }     /* end routine bldlst_ncb */
```

E.1.20. Build Input Item List for a Connection Request

```
/* ------------------------------------------------------------------------ */
unsigned int bldlst_connect(in_desc)
/* ------------------------------------------------------------------------ */
struct dsc$descriptor *in_desc;
```

```
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine builds an input item list for a connection request
 *
 * FORMAL PARAMETERS:
 *
 * None.
 *
 * IMPLICIT INPUTS:
 *
 * in_desc
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */
 {
 int i;     /* general */
 int size;    /* size of built list in bytes */
 struct _osit_item *item;  /* pointer at an individual item */

 size = 0;
 item = (struct _osit_item *)in_desc->dsc$a_pointer;

/* add in the protocol version - mandatory parameter */

 item -> osit$w_item_length = OSIT$K_ITEM_HEADER_SIZE + 4;
 item -> osit$w_item_type = OSIT$K_ITEM_PROTOCOL_TYPE;
 item -> osit$r_item_value.osit$l_item_long = OSIT$K_OSI_PROTOCOL;
 item = (struct _osit_item *)((unsigned int)item +
                                      item->osit$w_item_length);

 /* add in the address of the request */

 item = (struct _osit_item *)add_string_item
  (
  item,   OSIT$K_ITEM_ADDRESS,
  "TEST",
  4
  );

/* add in the called tsap */

 item = (struct _osit_item *)add_string_item
  (
  item,
  OSIT$K_ITEM_CALLED_TSAP,
  "RANDOM",
  6
```

```
  );

/* add in the calling tsap */

 item = (struct _osit_item *)add_string_item
  (
  item,
  OSIT$K_ITEM_CALLING_TSAP,
  "RANDOM-INITIATOR",
  16
  );

/* add in the prefered classes */

 item -> osit$w_item_length = OSIT$K_ITEM_HEADER_SIZE + 4;
 item -> osit$w_item_type = OSIT$K_ITEM_CLASS;
 item -> osit$r_item_value.osit$l_item_long
      = OSIT$M_CLASS_0 + OSIT$M_CLASS_2 + OSIT$M_CLASS_4;
 item = (struct _osit_item *)((unsigned int)item +
                  OSIT$K_ITEM_HEADER_SIZE + 4);

 /* add in the prefered options */

 item -> osit$w_item_length = OSIT$K_ITEM_HEADER_SIZE + 4;
 item -> osit$w_item_type = OSIT$K_ITEM_OPTIONS;
 item -> osit$r_item_value.osit$l_item_long
      = OSIT$M_EXTENDED + OSIT$M_CHECKSUM + OSIT$M_EXPEDITED;
 item = (struct _osit_item *)((unsigned int)item +
                                      OSIT$K_ITEM_HEADER_SIZE + 4);

/* set up the length in the input item list */
 in_desc->dsc$w_length
      = (int)item - (int)in_desc->dsc$a_pointer;
 }    /* end routine bldlst_connect */
```

E.1.21. Display Output Item List

```
/* ------------------------------------------------------------------------ */
unsigned int display_list(list, size)
/* ------------------------------------------------------------------------ */
char *list;int size;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine dumps the contents of an output item list to
 *
 the console
 *
 * FORMAL PARAMETERS:
 *
 * list - a pointer to the list
 * size - the length of the list in bytes
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
```

```
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */
 {
 struct _osit_item *item;  /* pointer at an individual item */
 unsigned char *ptr;   /* byte pointer */
 int offset;

 item = (struct _osit_item *)list;
 if (item -> osit$w_item_length == 0)
  printf ("\titem list is empty\n");
 else
  {
  offset = 0;  while (offset < size)
   {   ptr = (unsigned char *)((unsigned int)list + offset);
   item = (struct _osit_item *)ptr;
   display_item (item);
   offset = offset + item -> osit$w_item_length;
   }  /* end do while not end of list */
  }   /* item list has contents */
 }    /* end routine display_list */
```

E.1.22. Displays a Specified Item

```
/* ------------------------------------------------------------------------ */
unsigned int display_item(item)
/* ------------------------------------------------------------------------ */
struct _osit_item *item;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine dumps the contents of an item in an item list
 *
 * FORMAL PARAMETERS:
 *
 * item   - a pointer to the item
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
```

```
 *
 * None.
 */
    {
    short int i, class, service;
    struct _osit_optmsk *mask;
    switch (item -> osit$w_item_type)
 {
 case OSIT$K_ITEM_PROTOCOL_TYPE :
     printf ("\tprotocol type - %d\n",
  item -> osit$r_item_value.osit$l_item_long);
     break; case OSIT$K_ITEM_TC_ID :
     printf ("\ttc id - %d\n",
  item -> osit$r_item_value.osit$l_item_long);
     break; case OSIT$K_ITEM_PROTOCOL_VERSION :
     printf ("\tprotocol version - %d\n",
  item -> osit$r_item_value.osit$l_item_long);
     break; case OSIT$K_ITEM_USER_DATA :
     printf ("\tuser data - %.*s\n",
  item -> osit$r_item_value.osit$r_item_wcs.osit$w_wcs_length,
         &(item -> osit$r_item_value.osit$r_item_wcs.osit$t_wcs_text));
     break;
 case OSIT$K_ITEM_CLASS :
     class = item -> osit$r_item_value.osit$l_item_long;
     printf("\tclass - ");
     for (i=0;i<=4;i++) if (class & (1<<i)) printf(" %d",i);
     printf("\n");
     break;
case OSIT$K_ITEM_NETWORK_SERVICE:
     service = item->osit$r_item_value.osit$l_item_long;
     printf("\tnetwork service - %d\n",
  item->osit$r_item_value.osit$l_item_long);
     break;
case OSIT$K_ITEM_OPTIONS :
     printf ("\toptions - ");
     mask = (struct _osit_optmsk *)&(item->osit$r_item_value.osit
$l_item_long);
     if (mask -> osit$v_extended)
  printf("extd format, ");
     else
  printf("normal format, ");
     printf("checksums ");
     if (mask -> osit$v_checksum)
  printf("on, ");
     else
  printf("off, ");
     if (!(mask -> osit$v_expedited))
  printf("no ");
     printf("expd data, ");
     printf("flow cntrl ");
     if (mask -> osit$v_flow_control)
  printf("on.\n");
     else
  printf("off.\n");
     break;
  case OSIT$K_ITEM_ADDRESS :
     printf ("\taddress - %.*s\n",
  (item -> osit$w_item_length) - OSIT$K_ITEM_HEADER_SIZE,
```

```
  &(item -> osit$r_item_value));
     break;
  case OSIT$K_ITEM_CALLED_TSAP :
     printf("\tcalled tsap - %.*s\n",
  (item -> osit$w_item_length) - OSIT$K_ITEM_HEADER_SIZE,
  &(item -> osit$r_item_value));
     break;
case OSIT$K_ITEM_CALLING_TSAP :
     printf("\tcalling tsap - %.*s\n",
  (item -> osit$w_item_length) - OSIT$K_ITEM_HEADER_SIZE,
  &(item -> osit$r_item_value));
     break;
case OSIT$K_ITEM_NETWORKPRIORITY_IN :
case OSIT$K_ITEM_NETWORKPRIORITY_OUT :
     break;
default :
     printf ("\tunrecognised type (%d)\n", item -> osit$w_item_type);
}
    } /* end routine display_item */
```

E.1.23. Report $QIO Error

```
/* ------------------------------------------------------------------------ */
unsigned int report_error(code,test)
/* ------------------------------------------------------------------------ */
short int code;struct testblock *test;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine is called wheneverf an error is given from a QIO
 * call
 *
 * FORMAL PARAMETERS:
 *
 * code - error code causing consternation
 * test - testblock owning the error
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */
 {
 short int i;

 printf ("[%d, %d] error: QIO failed:\n",
     test->unit);
```

```
 printf ("status\t%x\n", code);
 LIB$STOP (code);
     /* end routine report_error */
```

E.1.24. Read Mailbox

```
/* ------------------------------------------------------------------------ */
unsigned int post_mailbox_read(mail)
/* ------------------------------------------------------------------------ */
struct mailblock *mail;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine posts a read on the mailbox associated with the
 * mail block supplied.
 *
 * FORMAL PARAMETERS:
 *
 * mail - pointer to a mailblock
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */
  {
/*
 * post another read on the mailbox
 */
 if ($error (status = SYS$QIO (ASYNC_EFN, mail->channel,
      IO$_READVBLK, &(mail->iosb), &mailbox_ast, mail,
      &(mail->msg.type), RANDOM$K_MBX_MSG_SIZE,
      0, 0, 0, 0)))
  {
  printf("error, failed to post read on mailbox\n");
  LIB$STOP (status);
  }
 }     /* end routine post_mailbox_read */
```

E.1.25. Report Mailbox Message Type

```
/* ------------------------------------------------------------------------ */
unsigned int process_mailbox_message(mail)
/* ------------------------------------------------------------------------ */
struct mailblock *mail;
/*
```

```
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine reports the message type and takes the appropriate
 * actions.
 *
 * FORMAL PARAMETERS:
 *
 * mail - pointer to a mailblock
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */
  {
 unsigned char *info_size;
 struct testblock *test;

 info_size = &mail->msg.nam [mail->msg.nct];
/*
 * find the test block for the VOTS channel associated with
 * this mailbox message
 */

 test = random$ga_tests;
 while (test->unit != mail->msg.unit) test = test->next;
/*
 * print out the type of the message
 */
 printf ("[%d] mailbox ", mail->msg.unit);
 switch (mail->msg.type)
 {
 case MSG$_DISCON:
         printf ("hangup message\n");
  break;
 case MSG$_CONFIRM :
  printf ("confirmation message\n");
  break;
 case MSG$_CONNECT :
  printf ("connect message\n");
  break;
 case MSG$_ABORT :
  printf ("abort message\n");
  break;
 case MSG$_PROTOCOL :
  printf ("protocol message\n");
  break;
```

```
 case MSG$_PATHLOST :
  printf ("pathlost message\n");
  break;
 case MSG$_TIMEOUT :
  printf ("timeout message\n");
  break;
 case MSG$_THIRDPARTY :
  printf ("third party message\n");
  break;
 case MSG$_REJECT :
  printf ("reject message\n");
  break;
 case MSG$_EXIT :
  printf ("exit message\n");
  break;
 case MSG$_INTMSG :
  printf ("expedited data message\n");
  break;
 case MSG$_NETSHUT :
  printf ("network shutdown message\n");
  break; default :
  printf ("unknown (id = %d)",mail->msg.type);
  }
/*
 * our actions depend on our role, state and the message
 */
 switch (mail->msg.type)
  {
  case MSG$_REJECT :
  case MSG$_CONFIRM :
   break;
  case MSG$_ABORT :
  case MSG$_DISCON :
   if (test->flags.wfdis)
    {
    disconnect (test);
    test->flags.wfdis = FALSE;
    break;
    }
  case MSG$_CONNECT :
  case MSG$_PROTOCOL :
  case MSG$_PATHLOST :
  case MSG$_TIMEOUT :
  case MSG$_THIRDPARTY :
  case MSG$_EXIT :
  case MSG$_INTMSG :
  case MSG$_NETSHUT :
  default :
   printf("error: unexpected message\n");
   disconnect (test);
   return (FALSE);
  }
 return (TRUE);
 }     /* end routine process_mailbox_message */
```

E.1.26. Wait for Mailbox Message and Read Mailbox

```
/* ------------------------------------------------------------------ */
```

```
unsigned int mailbox_ast(mail)
/* ------------------------------------------------------------------------ */
struct mailblock *mail;
/*
 * FUNCTIONAL DESCRIPTION:
 *
 * This routine waits on the mailbox assigned to VOTS for a message
 * and then tells the user what the message was and posts another
 * read on the mailbox.
 *
 * FORMAL PARAMETERS:
 *
 * mail - pointer to a mailblock
 *
 * IMPLICIT INPUTS:
 *
 * None.
 *
 * IMPLICIT OUTPUTS:
 *
 * None.
 *
 * ROUTINE VALUE:
 *
 * None.
 *
 * SIDE EFFECTS:
 *
 * None.
 */
  {
/*
**  If there are not any active tests, then don't bother processing the
**  mailbox message.
*/

printf("[%d] mailbox read completed\n", mail->msg.unit);

if (random$ga_tests != NULL)
{
    /*
    * if we need to perform any more reads on the mailbox, then drain it
    * of any other reads in the pipeline
    */ if (process_mailbox_message (mail))
  {
  do
   {
   if ($error (status = SYS$QIOW (ASYNC_EFN,mail->channel,
        IO$_READVBLK | IO$M_NOW, &(mail->iosb), 0, mail,
        &(mail->msg.type), RANDOM$K_MBX_MSG_SIZE,
        0, 0, 0, 0)))
    {
    printf("error, failed to post mailbox read\n");
    LIB$STOP (status);
    }
   else
    {
    if (mail->iosb [0] != SS$_ENDOFFILE)
```

```
     process_mailbox_message (mail);
    }
   }
   while (mail->iosb [0] != SS$_ENDOFFILE);
  post_mailbox_read (mail);
  }
}
 }     /* end routine mailbox_ast */
/
```