VMS DECwindows Transport Manual

Order Number: AA–PABWA–TE

October 1989

This document describes the theory-of-operation and interconnection of each component of the VMS DECwindows transport layer. This document also describes the recommended coding procedures that you should follow when augmenting the DECwindows transport with a third-party transport.

Revision/Update Information:	This is a new manual.
Software Version:	VMS Version 5.3

digital equipment corporation
maynard, massachusetts

October 1989

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

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

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

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

CDA
DDIF
DEC
DECnet
DECUS
DECwindows
DECwriter
DEQNA
DIGITAL
GIGI
LiveLink
LN03
MASSBUS
MicroVAX
PrintServer 40
Q-bus
ReGIS
ULTRIX
UNIBUS
VAX
VAXcluster
VAX RMS
VAXserver
VAXstation
VMS
VT
XUI
digital™

The following are third-party trademarks:

PostScript is a registered trademark of Adobe Systems, Inc.

UNIX is a registered trademark of American Telephone & Telegraph Company.

X Window System, Version 10 and its derivations (X, X10, X Version 10, X Window System) are trademarks of the Massachusetts Institute of Technology.

X Window System, Version 11 and its derivations (X, X11, X Version 11, X Window System) are trademarks of the Massachusetts Institute of Technology.

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by Digital. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use Digital-supported devices, such as the LN03 laser printer and PostScript printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

Contents

INDEX

EXAMPLES

FIGURES

TABLES

Preface

The *VMS DECwindows Transport Manual* provides information needed to write a DECwindows transport interface that runs under VMS Version 5.3 and to load it under DECwindows. Digital makes no guarantee that transport interfaces written using these guidelines will execute without modification on future versions of the operating system. Because this is the first version of the VMS system that supports user-written transport layers, it is likely that the existing programming interface will change.

Intended Audience

This document is intended for programmers who need information about the components and interfaces of the VMS DECwindows transport layer. It describes the theory-of-operation and interconnection of each component of the VMS DECwindows transport layer. It also describes the recommended coding procedures that you should follow when augmenting the DECwindows transport layer with a third-party transport.

You should read this document before modifying or replacing the VMS DECwindows transport.

This document assumes that you are familiar with the overall design of the VMS DECwindows implementation.

Document Structure

The *VMS DECwindows Transport Manual* is organized into the following chapters:

- Chapter 1 provides an overview of the VMS DECwindows transport layer.
- Chapter 2 describes how X11 protocol requests, events, errors, and replies are generated and transmitted in the VMS DECwindows environment. Although the transport layer itself does not interpret the data that it transfers, you should read this chapter to become familiar with how the transport layer supports the X11 protocol.
- Chapter 3 describes the theory-of-operation and interconnection of each component of the VMS DECwindows transport layer. This chapter includes a description of the functions performed by the common and specific components and how they interact.
- Chapter 4 describes a walk-through of typical transport layer activities.
- Chapter 5 describes the transport-common routines that a transport-specific component needs to call. You should read this chapter to become familiar with the operation and use of these routines.
- Chapter 6 describes the transport-specific routines that you must implement if you write your own transport-specific component.

- Chapter 7 describes the transport-layer utility routines that you can use. These routines are provided for your convenience; there is no requirement that you use them, but you must implement similar functions.
- Chapter 8 describes a cookbook approach to writing your own transport-specific routines. The chapter includes a sample BLISS-32 code example for each of the transport-specific routines that you must write. Your own implementation of these routines may differ depending on your chosen network service.

Associated Documents

For more information about DECwindows, see the VMS DECwindows documentation set.

Conventions

The following conventions are used in this manual:

. . .

In examples, a horizontal ellipsis 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 format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.

[ ]

In format descriptions, brackets indicate that whatever is enclosed within the brackets is optional; you can select none, one, or all of the choices. (Brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)

{ }

In format descriptions, braces surround a required choice of options; you must choose one of the options listed.

boldface text

Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason.

italic text

Italic text represents information that can vary in system messages (for example, Internal error *number*).

UPPERCASE TEXT

Uppercase letters indicate that you must enter a command (for example, enter OPEN/READ), or they indicate the name of a routine, the name of a file, the name of a file protection code, or the abbreviation for a system privilege.

-

Hyphens in coding examples indicate that additional arguments to the request are provided on the line that follows.

numbers

Unless otherwise noted, all numbers in the text are assumed to be decimal. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1 Overview of the VMS DECwindows Transport Layer

The VMS DECwindows transport layer is separated into transport-common and transport-specific components. The routines that comprise the transport-common layer are network independent and are used to buffer and queue data to be sent between a client and server. The transport-common routines then call routines that are specific to a particular network service, such as DECnet and TCP/IP, to actually move the data across the network.

The transport-specific routines perform the following functions:

- Initialize (attach) a specific network service
- Connect a client to a server
- Write data to the network service
- Read data from the network service
- Close a connection and release connection resources

To implement your own transport layer, you must write the transport-specific routines to support your particular network service and link them as a VMS shareable image that can be accessed by the transport-common component.

Chapter 8 includes examples of transport-specific routines that implement a TCP/IP transport layer for DECwindows. You can use these code examples as a starting point when writing your own transport-specific routines.

Depending upon the network service on which you are building, you will probably find that the routines that initialize a transport and establish a connection require the most modification. You may also find that routines that primarily insert and remove buffers from the queues can be used with minimal changes.

The remainder of this chapter provides an overview of the VMS DECwindows transport layer. Subsequent chapters describe the transport layer components and their interconnection in greater detail.

1.1 The Transport Layer Function

The function of the transport layer is to move X Window System, Version 11 (X11) protocol requests between an application, called the client, and the X11 server in an efficient manner. The transport layer transmits data over network transports. VMS DECwindows currently supports three network transports: DECnet, TCP/IP, and a high-performance local transport.

In the X11 environment, the mechanism for sending information from a client to the server is by way of a “connection” to the server. Creating a client/server connection is known as opening a **display**; when you open a display, you open a connection.

The transport layer is a general data-transfer mechanism; it does not interpret or understand the format of the data that it transfers. The transport layer operates symmetrically on both ends of the client/server connection: it buffers and sends output requests from Xlib to the server and buffers and sends input events, errors, and replies to Xlib. The buffers are maintained in a series of communication queues.

The transport layer maintains status (communication context) on a per-connection basis.

1.2 Transport Common/Specific Architecture

The VMS DECwindows transport layer is installed as part of the DECwindows common component; that is, the transport layer is always installed when DECwindows is installed. The DECwindows DECW$STARTUP.COM procedure installs all of the transport images—DECW$TRANSPORT_COMMON.EXE, DECW$TRANSPORT_LOCAL.EXE, DECW$TRANSPORT_DECNET.EXE, and DECW$TRANSPORT_TCPIP.EXE—each time DECwindows is started.

Xlib and the server both initialize and attach a network-specific transport for their respective side of the connection.

The transport layer is separated into transport-common and transport-specific functions. The transport-common functions, which are contained in DECW$TRANSPORT_COMMON.EXE, provide the generic services needed by Xlib or the server. The transport-common functions buffer and queue the data to be sent between the client and server and then call functions in the various images (DECW$TRANSPORT_LOCAL.EXE, DECW$TRANSPORT_DECNET.EXE, DECW$TRANSPORT_TCPIP.EXE) that are specific to a particular transport service, such as local, DECnet, and TCP/IP, respectively, to actually move the data.

This architecture allows the transport-common component to present a common buffer/queue interface to other DECwindows components while the transport-specific component “hides” the details of how data is actually transmitted.

The transport-common/transport-specific architecture is particularly important because the transport layer must be flexible enough to efficiently support two types of client/server connections:

- If the client and server are executing on two different VMS systems, a transport image physically executes on both the client system and the server system. The two transports then establish a network connection between themselves to pass the data. The transport layer buffers data to reduce the number of required network round-trips and their associated overhead.

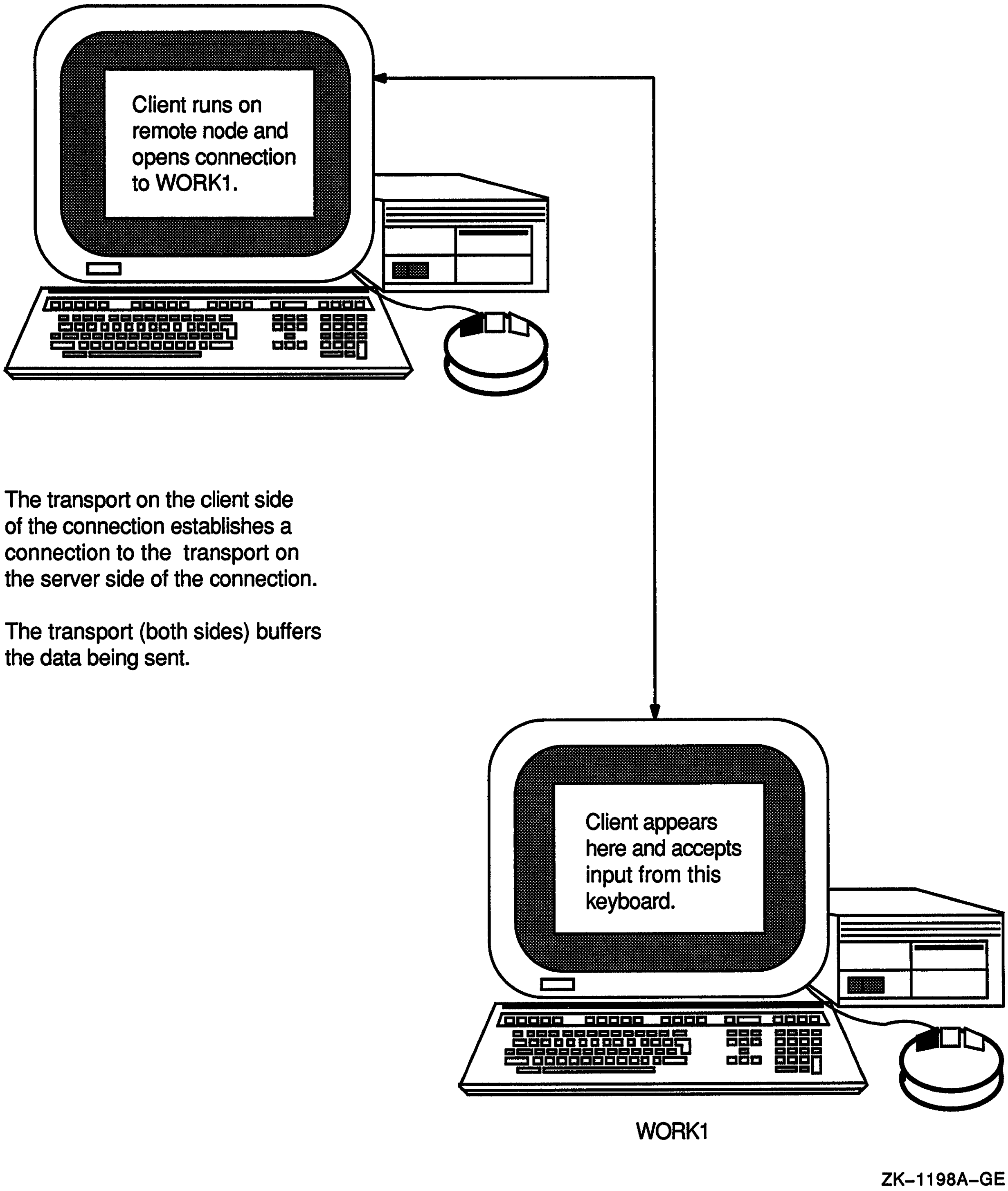
  This is called a **remote** connection.

- If the client and server are executing on the same VMS system, there is no need to establish a network connection, but the client must still open a logical connection to the server to pass data. This is called a **local** connection. Xlib and the server both initialize the transport-common code for their respective side of the client/server connection just as they would in the case of a remote connection. However, Xlib and the server then attach a local transport to transmit data.

The VMS DECwindows transport layer establishes remote or local connections to a server based on the display name argument passed by a client in the Xlib OPEN DISPLAY routine or through information specified in a SET DISPLAY command. (By passing a null value in the call to the OPEN DISPLAY routine, a client need not hard code the display name. See the *VMS DECwindows Xlib Programming Volume* for more information.)

In a remote transport connection, a node name other than "0" indicates that a remote workstation node in the network is to be used as the display; the client application and the server do not execute on the same physical machine. The transport layer performs its buffering functions and calls a transport such as DECnet to send the data across the network, as shown in Figure 1–1.

Figure 1–1 Remote Transport Connection



Client runs on remote node and opens connection to WORK1.

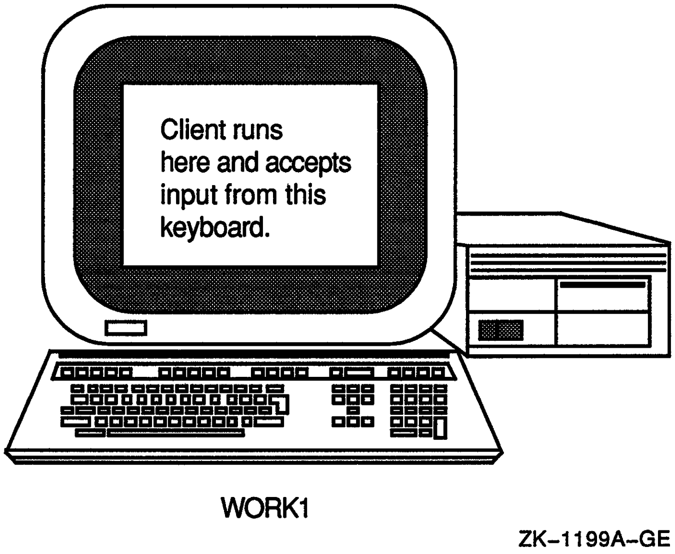
The transport on the client side of the connection establishes a connection to the transport on the server side of the connection.

The transport (both sides) buffers the data being sent.

Client appears here and accepts input from this keyboard.

In a local transport connection, a node name of "0" indicates that the client and server are executing on the same hardware, as shown in Figure 1–2. There is no need for the transport layer to send the data across the network, but Xlib and the server must still communicate. The local transport layer implements a shared-memory transport. The shared-memory transport performs functions that are similar to those performed by the DECnet or TCP/IP transport-specific components, but does not incur the network overhead.

Figure 1–2 Local Transport Connection



2 X11 Protocol Overview

The X Window System, Version 11 (X11) standard defines a common protocol for all communications between client applications and implementations of the X11 server. This standard protocol makes it possible to mix client/server pairs from different operating-system and vendor environments.

The protocol defines the format of the data, such as the request, reply, error, and event formats, that is passed between the client and server; it does not dictate the mechanism for transporting this data. The protocol nests inside transport mechanisms that move the protocol requests between clients and servers.

This chapter describes how X11 protocol requests, events, errors, and replies are generated and transmitted in the VMS DECwindows environment. You should read this chapter to become familiar with how the transport layer supports the X11 protocol. Later chapters describe the transport layer's architecture and role in supporting the X11 protocol in greater detail.

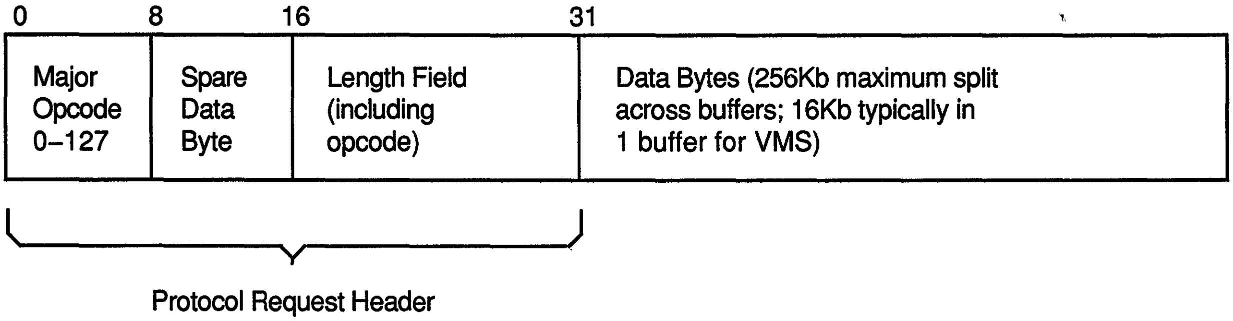
For more information about the X11 protocol, see the *X Window System C Library and Protocol Reference* by Robert W. Scheifler, James Gettys, and Ron Newman.

2.1 Generating a Protocol Request

When a DECwindows client program needs to generate output on a screen connected to a display, the client calls an Xlib or XUI Toolkit routine to perform the output. Xlib translates these routines into one or more protocol requests. The protocol request is sent from a client to server to invoke some operation in the server. Requests may be synchronous (the client waits until the server sends a reply packet) or asynchronous (no reply is generated and the client may send more requests).

The protocol request is either one of the core protocol requests or a protocol request that is interpreted by an extension. The format of the core protocol request is predefined to ensure portability of core requests across various implementations of the X Window System. The format of the protocol request is shown in Figure 2–1.

Figure 2–1 Protocol Request Format


0
8
16
31
Major Opcode 0–127
Spare Data Byte
Length Field (including opcode)
Data Bytes (256Kb maximum split across buffers; 16Kb typically in 1 buffer for VMS)
Protocol Request Header

ZK–1200A–GE

Every request consists of a 4-byte header, which contains an 8-bit "major" opcode, a spare data byte used that is usually used for the "minor" opcode of an extension, and a 16-bit length field. The header is followed by zero or more additional bytes of data; unused bytes are not required to be zero, except in a few special cases such as image text.

The core protocol requests use only seven of the eight available bits of the opcode field in the request header; major opcodes 0 through 127 are reserved for core requests. Extensions use all eight bits, and opcodes 128 through 255 are reserved for extensions.

Because the 128 opcodes available for extensions could be consumed fairly quickly, extensions usually have an additional 8-bit minor opcode encoded in the spare data byte of the request header. This minor opcode increases the number of requests that can be associated with a major opcode. By convention, each extension uses one major opcode.

Note: **The placement and interpretation of the minor opcode, and all other nonheader fields in extension requests, are not defined by the core protocol.**

The length field is a 16-bit value that defines the total number of longwords in the request, including the header. For example, if the value of the length field was 4096, multiply 4096 times 4 to compute a request length of 16,384 bytes. The maximum size of a protocol request that a server is willing to accept is server dependent; the server communicates this maximum request length to the client as part of the connection setup when a client opens a connection.

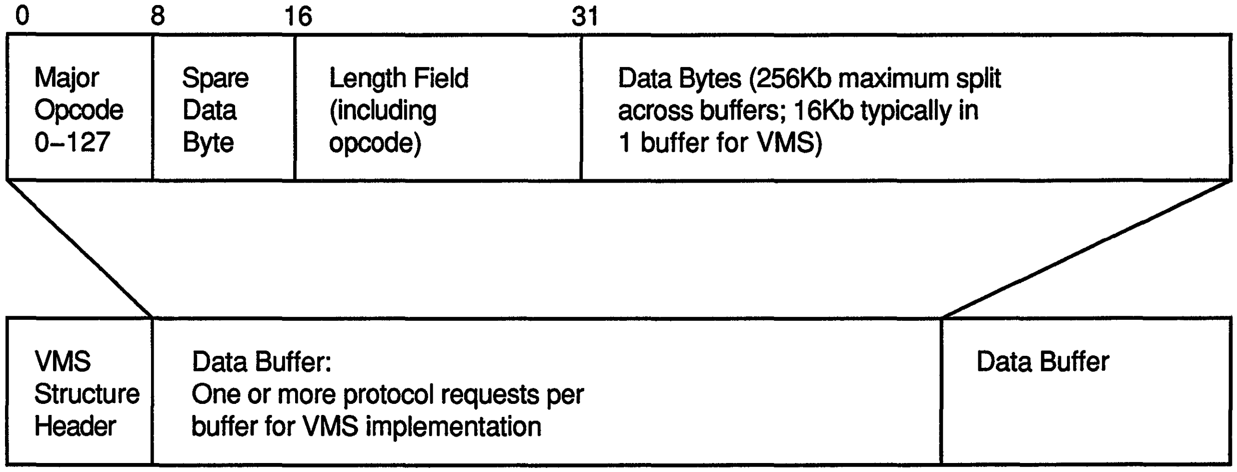
The length field must be exactly the number of longwords in the request. If the specified length is shorter or longer than the actual length, the protocol is corrupted and the default error handler may generate a fatal or nonfatal error. Under other circumstances an inconsistent length field can hang the server.

2.2 Xlib Output Buffering and Synchronization

Most of the Xlib routines add protocol requests to an output buffer; these protocol requests are later sent to the server when the buffer fills or is explicitly flushed by the client. The transport layer maintains the buffers on a per-connection basis. If a client explicitly flushes a buffer, only the output requests for that connection are affected; output buffered for other connections, either to the same or a different server, is not affected.

The total number of available output buffers is set by the server and Xlib when the transport layer is initialized. The Xlib output buffers for an individual connection are established when the connection to the server is opened. The format of the output request buffer is shown in Figure 2–2.

Figure 2–2 Output Buffer Containing a Protocol Request


0
8
16
31
Major Opcode 0–127
Spare Data Byte
Length Field (including opcode)
Data Bytes (256Kb maximum split across buffers; 16Kb typically in 1 buffer for VMS)
VMS Structure Header
Data Buffer: One or more protocol requests per buffer for VMS implementation
Data Buffer

ZK–1201A–GE

The number of protocol requests that Xlib sends in a single output buffer depends on the amount of data that is associated with each request. Because the VMS implementation of Xlib tries not to split protocol requests across output buffers, Xlib adds requests to a buffer until a protocol request does not fit completely within the remaining space. Xlib then flushes the current buffer and adds the waiting protocol request to the head of a new buffer.

Note: **The server negotiates the size of the largest request for each connection when the connection is opened. The VMS DECwindows server accepts up to a 16Kb protocol request.**

The maximum size of a protocol request allowed by the X11 protocol is 256Kb (65,535 times 4). The core protocol allows Xlib to split protocol requests across output request buffers, and other Xlib implementations are likely to do this. It is therefore possible that a read request will not return a complete protocol message and Xlib and the server must handle this case.

Xlib provides routines with which clients can control output buffering. If you do not want an output request to wait for the buffer to fill, you can follow it with an explicit call to a routine such as FLUSH, which sends all buffered output for a connection.

Xlib also includes routines that allow clients to synchronize output requests. Xlib accomplishes this synchronization by immediately flushing the client's Xlib output buffer after each output request and then calling a synchronization routine that generates a return.

2.3 Transporting the Protocol Request

Once the transport layer receives the output request buffer from Xlib, it arranges to have the data transmitted to the server. The transport layer on the server side of the connection receives the data and notifies the server that data is available.

Note: **The Xlib output request buffer is actually a transport layer communication buffer (XTCB). The XTCB data structure is described in Section 3.1.4.**

The connection could be across the network to the display, or within a workstation in the case where the client and server have established a local connection. The networking service is assumed to be reliable; Xlib, the transport layer, and the server assume that the data arrives intact and error-free.

The transport layer notifies the scheduler at user-AST level that it has data available for a connection. The dispatcher calls the transport layer to get the data.

2.4 Client Input

Input to clients comes in three forms: input events, errors, and replies. An event packet is the X11 protocol message sent from a server to a client that gives information about some event in the server, such as a windowing operation, a keyboard key transition, or a mouse movement. Clients can also use request packets (XSendEvent) to send events to another client by way of the server. An event packet consists of exactly 32 bytes.

An error packet is the X11 protocol message sent from a server to a client that indicates an error state in the server. An error packet consists of exactly 32 bytes.

A reply packet is the X11 protocol message sent from a server to a client in response to a client protocol request that generates data (for example, GetImage). A reply can be any length with a maximum size of some 16 gigabytes.

2.4.1 Handling Input

Most input events are reported to clients relative to windows. Events are usually sent to the smallest enclosing window in which the pointer is located that is interested in the type of event being sent. It is also possible to assign the keyboard input focus to a specific window. When the input focus is attached to a window, keyboard events go to the client that has selected input on that window, rather than to the window in which the pointer is located.

The input component of the server services the XEvent queue and interacts with the window manager to determine which window is associated with the position recorded in the input queue entry. The window ID to which the event is to be delivered is then recorded in the event packet. The input component calls the events component to send the input events to clients that want to know about input in this window ID.

Client programs can use the **event_mask** argument of the XSelectInput routine to select the events for which they want to receive notification. The server does not send events to a client unless the client has specifically asked to be informed of that type of event. One exception to this rule is that one client can use the XSendEvent routine to force events to be sent to other clients.

An interest list is attached to the window ID data structure. Each entry on the interest list describes filtering parameters defined by the client's event masks. If there are multiple windows interested in the event, the event is sent to multiple windows.

The events component calls the transport layer to transmit the processed input event packets to the Xlib input queue. The transport layer buffers the events packets on a per-connection basis and delivers them to the Xlib input queues. As in the case for output, the transport layer does not interpret the data. Input normally arrives at the client as an I/O completion. Typically, the client processes the input event and generates output requests.

The input event is retained by Xlib until a client requests the next, or the next matching, event. It is the responsibility of the client to request the event; if the client does not request the event, the event remains in the queue.

Xlib provides routines such as XPeekIfEvent and XCheckIfEvent that client programs can call to check for particular types of events on their respective input queues. These routines require client programs to supply a procedure that determines if the next event in the queue matches the one that the client wants.

3 Transport Layer Architecture

This chapter describes the theory-of-operation and interconnection of each component of the VMS DECwindows transport layer. The relationship between the transport-common and transport-specific layers, as described in Section 3.3, is of particular importance.

The chapter begins with a description of the transport layer data structures.

3.1 Transport Layer Data Structures

The transport layer data structures maintain the state of the transport and each of the established connections. The transport layer data structures are created in stages as the transport-common code is initialized, a transport-specific mechanism is attached, and connections are established.

The XTPB, IXTCC, and XTDB data structures are allocated from memory pages that have user-mode read access/executive-mode write access (UREW) to prevent modification by less privileged access levels, including the transport-specific code that runs in user mode. The transport-common code depends on the accuracy of the contents of the XTPB, IXTCC, and XTDB data structures.

The transport layer data structures are described in Table 3–1.

Table 3–1 Transport Layer Data Structures

Name	When Created	Write Access Mode	Description
XTPB	Initialization Transport attach Connection open	exec	Transport parameter block (XTPB) contains default transport parameters. There is a three-level hierarchy of XTPB data structures.
IXTCC	Connection open	exec	Internal transport communications context (IXTCC) describes an established connection for executive-mode routines within the transport layer. The IXTCC is used by executive-mode routines to store protected data.

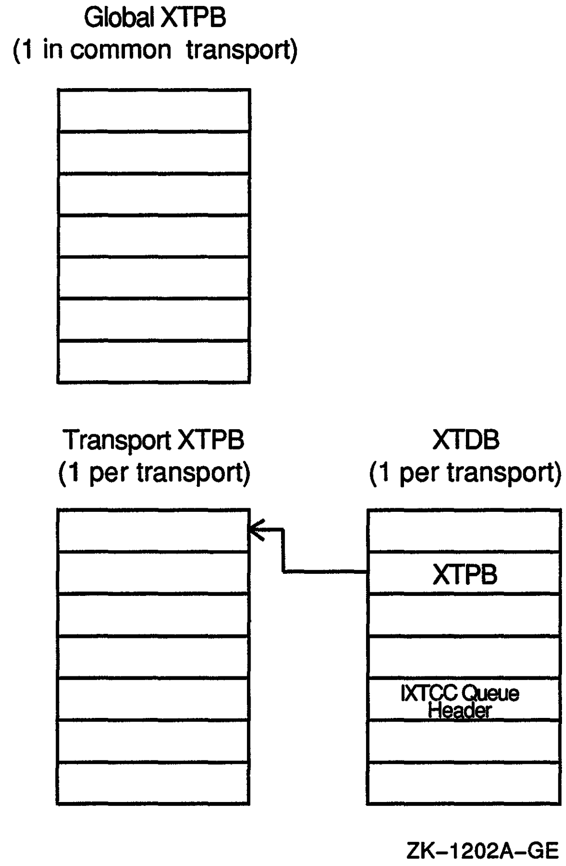
(continued on next page)

Table 3–1 (Cont.) Transport Layer Data Structures

Name	When Created	Write Access Mode	Description
XTCC	Connection open	user exec	Transport communications context (XTCC) describes an established connection. The XTCC is used to pass user-mode data to executive-mode routines. The XTCC may be modified by user-mode code and is visible to either the server or Xlib.
XTDB	Transport attach	exec	Transport descriptor block (XTDB) describes each attached transport.
XTCQ	Connection open	user exec	Transport communication queue (XTCQ) contains six per-connection communication queues and their states.
XTCB	Connection open	user exec	Transport communication buffers (XTCB) pass data in the transport layer.
XTFT	Transport attach	exec	Transport function table (XTFT) contains the addresses of the transport-specific routines. The XTFT is the transport-common code's link to the transport-specific code.

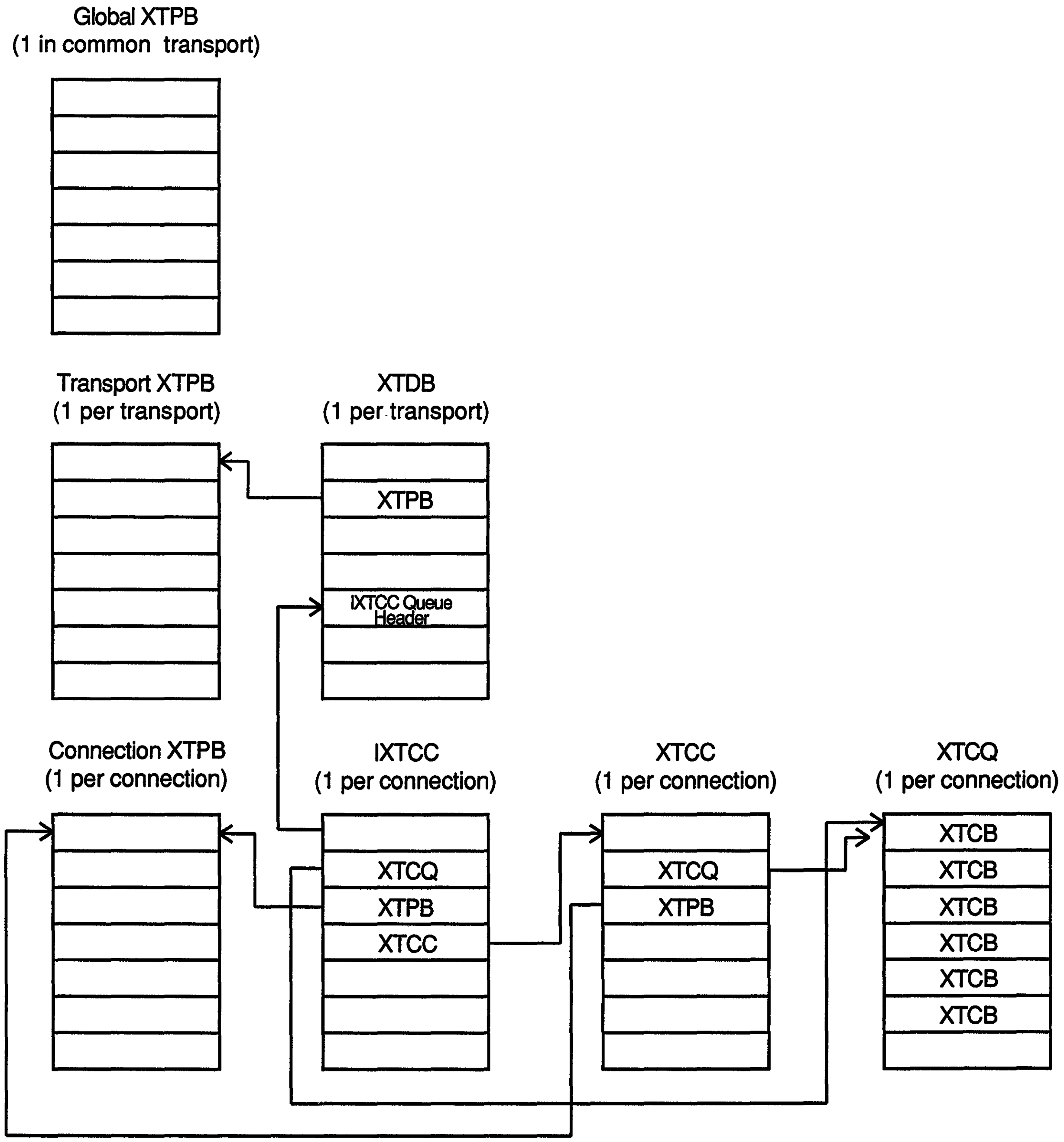
The transport-common code creates the global XTPB structure at initialization time. The transport-common DECW$XPORT_ATTACH_TRANSPORT routine creates XTDB, XTPB, and XTCB data structures, as shown in Figure 3–1. The XTCC, XTCQ, and XTCB data structures are allocated from memory pages that are user-writable so that routines running in either user or executive mode can modify them.

Figure 3–1 Transport Attach Data Structures



The transport-common open routine creates IXTCC and XTPB data structures. The transport-specific connection open routine creates XTCC, XTCQ, and XTCB data structures, as shown in Figure 3–2.

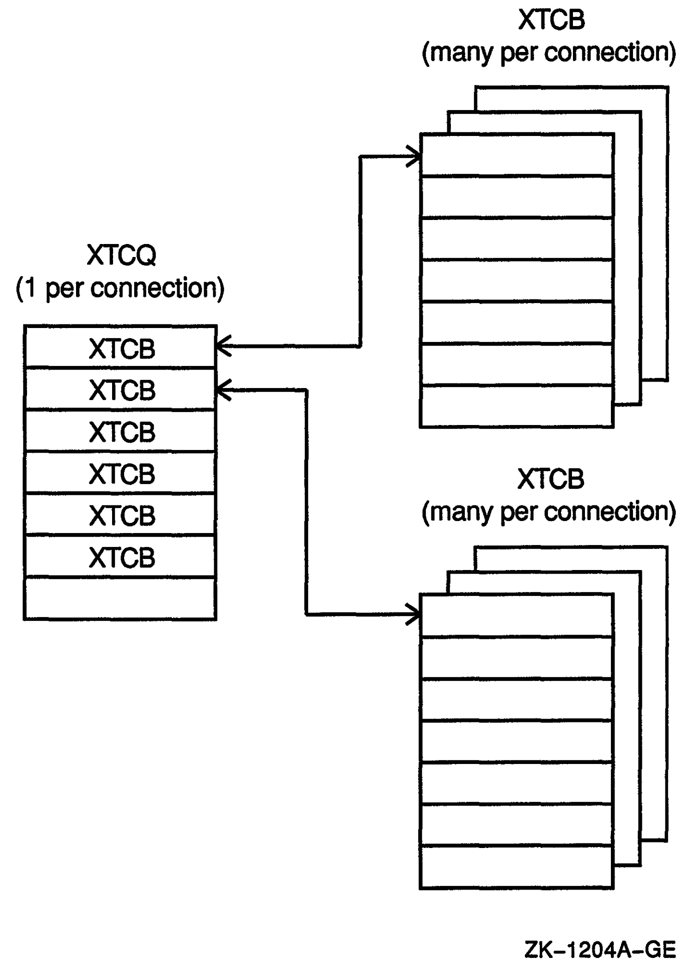
Figure 3–2 Transport Connection Open Data Structures



ZK-1203A-GE

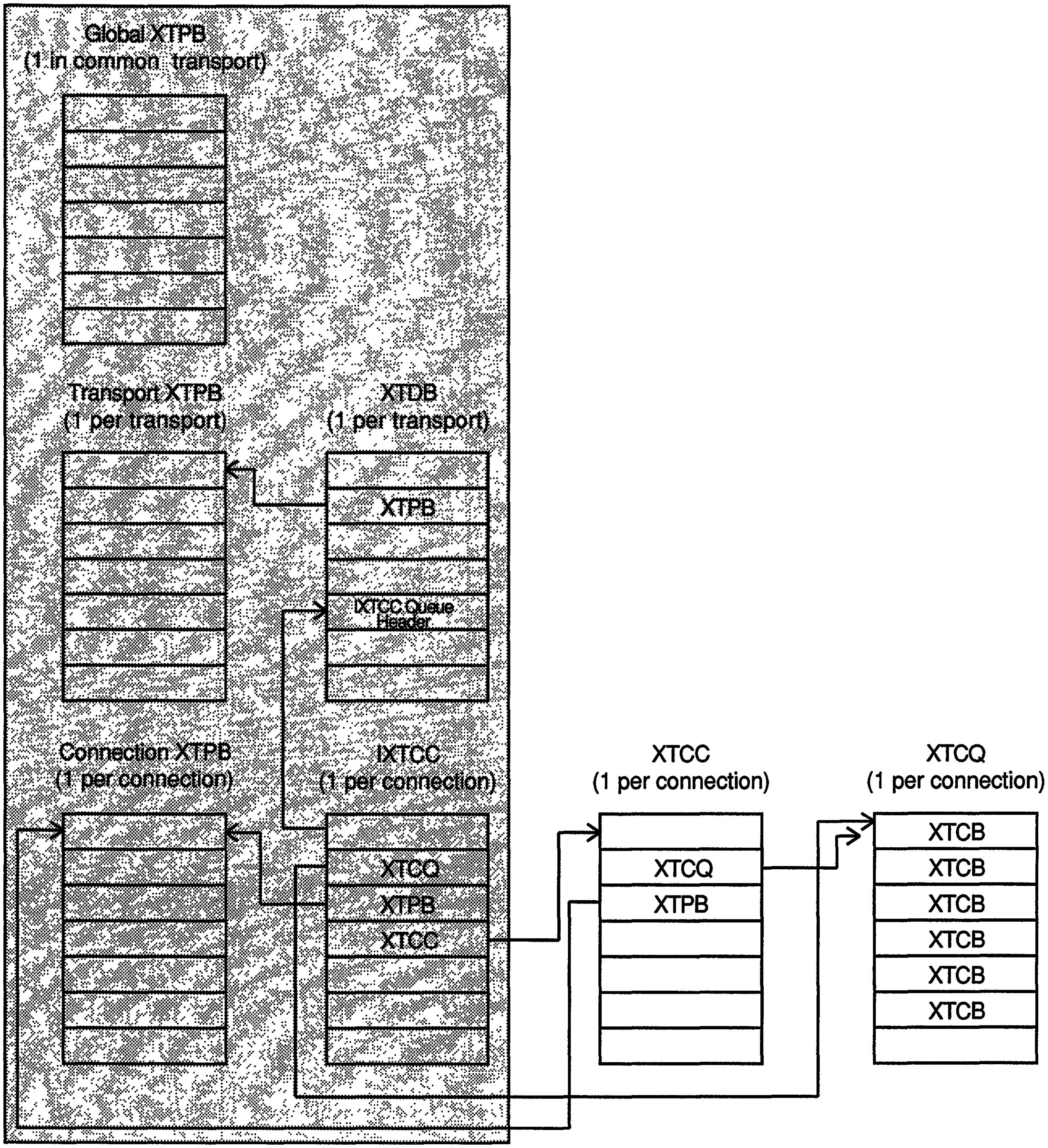
The transport-specific connection open routine also creates XTCB data structures, as shown in Figure 3–3.

Figure 3–3 Transport Connection Open XTCB Data Structures



The shaded areas of Figure 3–4 show the transport layer data structures that have user-mode read access/executive-mode write access.

Figure 3–4 Transport Data Structures



3.1.1 XTPB Data Structure

When the transport layer is initialized, the transport-common code creates a global transport parameter block (XTPB) data structure that contains default parameters. The default parameters are inherited by every network-specific transport that is subsequently attached. The attached transport can override the defaults.

Three levels of XTPB data structures are eventually built:

- A global XTPB data structure
- A transport-specific XTPB data structure
- A connection-specific XTPB data structure

The contents of the global XTPB data structure are copied to the transport-specific XTPB data structure when a transport is attached, and from the transport-specific XTPB data structure to a connection-specific XTPB data structure when a connection is established. The server and Xlib can override this inheritance of XTPB parameter values. After an XTPB has been created, changes to its parent's values do not change its values.

All XTPB data structures are allocated from memory pages that have user-mode read access/executive-mode write access (UREW) to protect them from modification by any less privileged access mode.

The XTPB data structure is shown in Figure 3–5.

Figure 3–5 XTPB Data Structure

XTPB$A_FLINK			0
XTPB$A_BLINK			4
XTPB$B_SUBTYPE	XTPB$B_TYPE	XTPB$W_SIZE	8
XTPB$W_DISPLAY_NUM		XTPB$W_FLAGS	12
XTPB$A_I_NOTIFY_RTNADR			16
XTPB$L_I_NOTIFY_RTNPRM			20
XTPB$A_O_NOTIFY_RTNADR			24
XTPB$L_O_NOTIFY_RTNPRM			28
XTPB$W_ON_EFN		XTPB$W_IN_EFN	32
XTPB$W_LRP_SIZE		XTPB$W_SRP_SIZE	36
XTPB$L_I_TIMEOUT			40
XTPB$W_I_LRP_COUNT		XTPB$W_I_SRP_COUNT	44

(continued on next page)

Figure 3–5 (Cont.) XTPB Data Structure

XTPB$W_O_LRP_COUNT	XTPB$W_O_SRP_COUNT	48
XTPB$L_O_TIMEOUT		52
XTPB$L_I_TICKS		56
XTPB$L_O_TICKS		60

Table 3–2 shows the contents of the XTPB data structure.

Table 3–2 XTPB Data Structure

Field	Use
XTPB$A_FLINK	Reserved for use by Digital.
XTPB$A_BLINK	Reserved for use by Digital.
XTPB$W_SIZE	Total length of this XTPB in bytes.
XTPB$B_TYPE	Constant 255.
XTPB$B_SUBTYPE	Constant DECW$C_DYN_XTPB (7).
XTPB$W_FLAGS	See the following list.
The following fields are defined within XTPB$W_FLAGS:	
XTPB$V_MODE	Type of transport connection. Possible values are: • DECW$K_XPORT_REMOTE_SERVER (0) • DECW$K_XPORT_REMOTE_CLIENT (1) • DECW$K_XPORT_LOCAL_SERVER (2) • DECW$K_XPORT_LOCAL_CLIENT (3)
XTPB$V_VALID	If 1, this XTPB is valid.
XTPB$W_DISPLAY_NUM	The server number. Valid only when the transport caller is a server.
XTPB$A_I_NOTIFY_RTNADR	Address of an AST routine to call for input notification.
XTPB$L_I_NOTIFY_RTNPRM	A longword value to be passed to the input notification AST routine.
XTPB$A_O_NOTIFY_RTNADR	Address of an AST routine to call for output notification.

(continued on next page)

Table 3–2 (Cont.) XTPB Data Structure

Field	Use
XTPB$L_O_NOTIFY_RTNPRM	Argument to be passed to the output notification AST routine.
XTPB$W_IN_EFN	Event flag to be set as part of input notification. If 0, no flag is set.
XTPB$W_ON_EFN	Event flag to be set as part of output notification. If 0, no flag is set.
XTPB$W_SRP_SIZE	Size, in bytes, of the data area of a small XTCB.
XTPB$W_LRP_SIZE	Size, in bytes, of the data area of a large XTCB.
XTPB$L_I_TIMEOUT	Number of milliseconds that a common transport waits for a blocking input function to complete. If 0, the common transport waits indefinitely.
XTPB$W_I_SRP_COUNT	Number of small XTCBs to allocate for input operations.
XTPB$W_I_LRP_COUNT	Number of large XTCBs to allocate for input operations.
XTPB$W_O_SRP_COUNT	Number of small XTCBs to allocate for output operations.
XTPB$W_O_LRP_COUNT	Number of large XTCBs to allocate for output operations.
XTPB$L_O_TIMEOUT	Number of milliseconds that a common transport waits for a blocking output function to complete. If 0, the common transport waits indefinitely.
XTPB$L_I_TICKS	XTPB$L_I_TIMEOUT converted to ticks of the internal watchdog timer.
XTPB$L_O_TICKS	XTPB$L_O_TIMEOUT converted to ticks of the internal watchdog timer.

Some fields in the global XTPB have hard-coded defaults that provide a minimal working environment. Xlib and the server change these values for better performance when they initialize a transport. The global XTPB defaults are shown in Table 3–3.

Table 3–3 XTPB Default Values

Field	Default Value	Description
XTPB$W_LRP_SIZE	16384	Default large XTCB size. The minimum and maximum values are 16 and 65408, respectively.
XTPB$W_SRP_SIZE	1408	Default small XTCB size. The minimum and maximum values are 16 and 65408, respectively.

(continued on next page)

Table 3–3 (Cont.) XTPB Default Values

Field	Default Value	Description
XTPB$W_ON_EFN	0	Do not set an event flag on output.
XTPB$W_IN_EFN	0	Do not set an event flag on input.
XTPB$L_I_TIMEOUT and XTPB$L_O_TIMEOUT	0	Do not time out.
XTPB$W_I_SRP_COUNT	8	The total number of small input XTCBs. The minimum and maximum values are 0 and 128, respectively.
XTPB$W_I_LRP_COUNT	1	The total number of large input XTCBs. The minimum and maximum values are 0 and 128, respectively.
XTPB$W_O_SRP_COUNT	8	The total number of small output XTCBs. The minimum and maximum values are 0 and 128, respectively.
XTPB$W_O_LRP_COUNT	1	The total number of large output XTCBs. The minimum and maximum values are 0 and 128, respectively.

3.1.2 IXTCC Data Structure

The internal transport communications context (IXTCC) data structure is created when a connection is created and describes the established connection. There are actually two per-connection structures that store the communications context:

- The IXTCC data structure is allocated from memory pages that have user-mode read access/executive-mode write access (UREW). On the client side of the connection, the transport-common DECW$XPORT_OPEN routine creates one IXTCC data structure per connection; on the server side of the connection, a transport-specific routine (such as the sample TRANSPORT_READ_AST routine described in Chapter 6) creates one IXTCC data structure per connection.

  The common transport uses the IXTCC structure to refer to a connection. The IXTCC structure exists during the lifetime of a connection.

- An XTCC data structure is user-modifiable. The server and Xlib use the XTCC to identify a connection when calling the transport layer. The XTCC structure exists during the lifetime of a connection.

The IXTCC data structure is shown in Figure 3–6.

Figure 3–6 IXTCC Data Structure

<table>
<tr><td colspan="3">IXTCC$A_FLINK</td><td>0</td></tr>
<tr><td colspan="3">IXTCC$A_BLINK</td><td>4</td></tr>
<tr><td>IXTCC$B_SUBTYPE</td><td>IXTCC$B_TYPE</td><td>IXTCC$W_SIZE</td><td>8</td></tr>
<tr><td colspan="3">IXTCC$A_TCQ</td><td>12</td></tr>
<tr><td colspan="3">IXTCC$A_TPB</td><td>16</td></tr>
<tr><td colspan="3">IXTCC$A_TDB</td><td>20</td></tr>
<tr><td colspan="3">IXTCC$A_TCC</td><td>24</td></tr>
<tr><td colspan="3">IXTCC$A_USER_REGION</td><td>28</td></tr>
<tr><td colspan="3">IXTCC$A_BUFFER_REGION</td><td>36</td></tr>
<tr><td colspan="3">IXTCC$L_ICI</td><td>44</td></tr>
<tr><td colspan="3">IXTCC$A_IW_QUEUE</td><td>48</td></tr>
<tr><td colspan="3">IXTCC$A_IFS_QUEUE</td><td>52</td></tr>
<tr><td colspan="3">IXTCC$A_IFL_QUEUE</td><td>56</td></tr>
<tr><td colspan="3">IXTCC$A_OW_QUEUE</td><td>60</td></tr>
<tr><td colspan="3">IXTCC$A_OFS_QUEUE</td><td>64</td></tr>
<tr><td colspan="3">IXTCC$A_OFL_QUEUE</td><td>68</td></tr>
<tr><td colspan="3">IXTCC$A_TCQ_FLAGS</td><td>72</td></tr>
<tr><td colspan="3">IXTCC$L_IWQ_FLAG</td><td>76</td></tr>
<tr><td colspan="3">IXTCC$L_IFSQ_FLAG</td><td>80</td></tr>
<tr><td colspan="3">IXTCC$L_IFLQ_FLAG</td><td>84</td></tr>
<tr><td colspan="3">IXTCC$L_OWQ_FLAG</td><td>88</td></tr>
<tr><td colspan="3">IXTCC$L_OFSQ_FLAG</td><td>92</td></tr>
<tr><td colspan="3">IXTCC$L_OFLQ_FLAG</td><td>96</td></tr>
</table>

(continued on next page)

Figure 3–6 (Cont.) IXTCC Data Structure

IXTCC$W_UNIT	IXTCC$W_SCREEN	100
IXTCC$Q_XPORT_RESERVED		104
IXTCC$L_DEC_RESERVED		112
IXTCC$A_XPORT_TABLE		116

Table 3–4 shows the contents of the IXTCC data structure.

Table 3–4 IXTCC Data Structure

Field	Use
IXTCC$A_FLINK	Forward, absolute pointer to the next IXTCC in the XTDB's queue of IXTCCs.
IXTCC$A_BLINK	Backward, absolute pointer to the previous IXTCC in the XTDB's queue of IXTCCs.
IXTCC$W_SIZE	Length of IXTCC in bytes.
IXTCC$B_TYPE	Constant 255.
IXTCC$B_SUBTYPE	Constant DECW$C_DYN_IXTCC (8).
IXTCC$A_TCQ	Address of XTCQ used by this connection.
IXTCC$A_TPB	Address of XTPB private to this connection.
IXTCC$A_TDB	Address of XTDB that owns this connection.
IXTCC$A_TCC	Address of XTCC associated with this connection.
IXTCC$A_USER_REGION	Range of addresses allocated by DECW$XPORT_ALLOC_INIT_QUEUES.
IXTCC$A_BUFFER_REGION	Range of addresses for communication buffers.
IXTCC$L_ICI	Internal connection identifier. IXTCC$L_ICI is a protected copy of XTCC$L_ICI.
IXTCC$A_IW_QUEUE	Address of input work queue in the XTCQ. For clients this is the event work queue; for servers this is the request work queue.
IXTCC$A_IFS_QUEUE	Address of the small XTCB input free queue in the XTCQ.
IXTCC$A_IFL_QUEUE	Address of the large XTCB input free queue in the XTCQ.

(continued on next page)

Table 3–4 (Cont.) IXTCC Data Structure

Field	Use
IXTCC$A_OW_QUEUE	Address of the output work queue in the XTCQ. For clients this is the request work queue; for servers this is the event work queue.
IXTCC$A_OFS_QUEUE	Address of the small XTCB output free queue in the XTCQ.
IXTCC$A_OFL_QUEUE	Address of the large XTCB output free queue in the XTCQ.
IXTCC$A_TCQ_FLAGS	Address of the flags longword in the XTCQ.
IXTCC$L_IWQ_FLAG	Bit position of the input work queue notification request flag in the flags longword in the XTCQ.
IXTCC$L_IFSQ_FLAG	Bit position of the small XTCB input free queue notification request flag.
IXTCC$L_IFLQ_FLAG	Bit position of the large XTCB input free queue notification request flag.
IXTCC$L_OWQ_FLAG	Bit position of the output work queue notification request flag.
IXTCC$L_OFSQ_FLAG	Bit position of the small XTCB output free queue interest flag.
IXTCC$L_OFLQ_FLAG	Bit position of the large XTCB output free queue interest flag.
IXTCC$W_SCREEN	Reserved for use by Digital.
IXTCC$W_UNIT	Reserved for use by Digital.
IXTCC$Q_XPORT_RESERVED	Quadword reserved for use by specific transport.
IXTCC$L_DEC_RESERVED	Reserved for use by Digital.
IXTCC$A_XPORT_TABLE	Address of XTFT structure for this specific transport.

3.1.3 XTCC Data Structure

As described in Section 3.1.2, the transport communications context (XTCC) data structure describes an established connection. The server and Xlib use the XTCC to identify a connection when calling the DECwindows transport layer. The XTCC structure exists during the lifetime of a connection. This is a user-writable structure.

The XTCC data structure is shown in Figure 3–7.

Figure 3–7 XTCC Data Structure

XTCC$A_FLINK			0
XTCC$A_BLINK			4
XTCC$B_SUBTYPE	XTCC$B_TYPE	XTCC$W_SIZE	8
XTCC$L_FLAGS			12
XTCC$A_TCQ			16
XTCC$A_TPB			20
XTCC$L_ICI			24
XTCC$A_IW_QUEUE			28
XTCC$A_IFS_QUEUE			32
XTCC$A_IFL_QUEUE			36
XTCC$A_OW_QUEUE			40
XTCC$A_OFS_QUEUE			44
XTCC$A_OFL_QUEUE			48
XTCC$A_TCQ_FLAGS			52
XTCC$L_IWQ_FLAG			56
XTCC$L_IFSQ_FLAG			60
XTCC$L_IFLQ_FLAG			64
XTCC$L_OWQ_FLAG			68
XTCC$L_OFSQ_FLAG			72
XTCC$L_OFLQ_FLAG			76
PAD			80
XTCC$Q_USER_RESERVED			88
XTCC$L_ERR_STATUS			96

(continued on next page)

Figure 3–7 (Cont.) XTCC Data Structure

XTCC$L_REM_USER_LEN	100
XTCC$A_REM_USER	104
XTCC$L_REM_NODE_LEN	108
XTCC$A_REM_NODE	112
XTCC$L_LCL_USER_LEN	116
XTCC$A_LCL_USER	120
XTCC$W_IN_IOSB	124
XTCC$W_ON_IOSB	132
XTCC$W_OW_IOSB	140
XTCC$L_IN_WAIT_TICKS	148
XTCC$L_OUT_WAIT_TICKS	152

Table 3–5 shows the contents of the XTCC data structure.

Table 3–5 XTCC Data Structure

Field	Use
XTCC$A_FLINK	Reserved for use by Digital.
XTCC$A_BLINK	Reserved for use by Digital.
XTCC$W_SIZE	Length of XTCC in bytes.
XTCC$B_TYPE	Constant 255.
XTCC$B_SUBTYPE	Constant DECW$C_DYN_XTCC (1).
XTCC$L_FLAGS	See the following list.

(continued on next page)

Table 3–5 (Cont.) XTCC Data Structure

Field	Use
The following fields are defined within XTCC$L_FLAGS:	
XTCC$V_MODE	Type of transport connection. Possible values are: • DECW$K_XPORT_REMOTE_SERVER (0) • DECW$K_XPORT_REMOTE_CLIENT (1) • DECW$K_XPORT_LOCAL_SERVER (2) • DECW$K_XPORT_LOCAL_CLIENT (3)
XTCC$V_ACTIVE	Connection has been established.
XTCC$V_DYING	Connection is aborting and no further operations should be allowed.
XTCC$V_INPUT_IN_PROG	Internal flag used by common transport to check usage consistency.
XTCC$V_OUTPUT_IN_PROG	Internal flag used by common transport to check usage consistency.
XTCC$V_MARK_FOR_CLOSE	Deferred close operation requested. Used privately by the common transport.
XTCC$V_ERR_STS_VALID	If 1, XTCC$L_ERR_STATUS field contains additional information about the cause of connection termination.
XTCC$V_LRP_ON_INPUT	If 1, specific transport is using large XTCBs for input operations, otherwise, small XTCBs.
XTCC$V_LRP_ON_OUTPUT	If 1, transport caller is using large XTCBs for output operations, otherwise, small XTCBs.
XTCC$V_WAIT_ON_WRITE	If 1, common transport is stalling to wait for specific transport to empty the output work queue so that a write_user operation can be initiated. (Write synchronization flag.)
XTCC$V_IN_AST_IN_PROG	If clear, the previous input notification AST has been delivered and no AST is in progress. This flag prevents EXQUOTA errors due to excess use of ASTs.
XTCC$V_OUT_AST_IN_PROG	Reserved for use by Digital.
XTCC$A_TCQ	Address of XTCQ used by this connection.
XTCC$A_TPB	Address of XTPB specific to this connection.
XTCC$L_ICI	Internal connection identifier. Used to find the corresponding IXTCC.

(continued on next page)

Table 3–5 (Cont.) XTCC Data Structure

Field	Use
XTCC$A_IW_QUEUE	Address of input work queue in the XTCQ. For clients this is the event work queue; for servers this is the request work queue.
XTCC$A_IFS_QUEUE	Address of the small XTCB input free queue in the XTCQ.
XTCC$A_IFL_QUEUE	Address of the large XTCB input free queue in the XTCQ.
XTCC$A_OW_QUEUE	Address of the output work queue in the XTCQ.
XTCC$A_OFS_QUEUE	Address of the small XTCB output free queue in the XTCQ.
XTCC$A_OFL_QUEUE	Address of the large XTCB output free queue in the XTCQ.
XTCC$A_TCQ_FLAGS	Address of the flags longword in the XTCQ.
XTCC$L_IWQ_FLAG	Bit position of the input work queue notification request flag in the flags longword in the XTCQ.
XTCC$L_IFSQ_FLAG	Bit position of the small XTCB input free queue notification request flag.
XTCC$L_IFLQ_FLAG	Bit position of the large XTCB input free queue notification request flag.
XTCC$L_OWQ_FLAG	Bit position of the output work queue notification request flag.
XTCC$L_OFSQ_FLAG	Bit position of the small XTCB output free queue notification request flag.
XTCC$L_OFLQ_FLAG	Bit position of the large XTCB output free queue notification request flag.
XTCC$Q_USER_RESERVED	Quadword reserved for use by transport caller.
XTCC$L_ERR_STATUS	Additional condition code information in case of connection failure.
XTCC$L_REM_USER_LEN	Length of the string identifying the remote user. (Valid for server only.)
XTCC$A_REM_USER	Address of the string identifying the remote user. (Valid for server only.)
XTCC$L_REM_NODE_LEN	Length of the string identifying the remote node. (Valid for server only.)
XTCC$A_REM_NODE	Address of the string identifying the remote node. (Valid for server only.)
XTCC$L_LCL_USER_LEN	Reserved for use by Digital.
XTCC$A_LCL_USER	Reserved for use by Digital.
XTCC$W_IN_IOSB	IOSB used in the $SYNCH system service when waiting for input operations to complete.

(continued on next page)

Table 3–5 (Cont.) XTCC Data Structure

Field	Use
XTCC$W_ON_IOSB	IOSB used in the $SYNCH system service when waiting for output operations to complete.
XTCC$W_OW_IOSB	IOSB used in the $SYNCH system service when stalling the transport caller so that the output work queue can be emptied prior to a write_user operation. (Write synchronization operation.)
XTCC$L_IN_WAIT_TICKS	Zero minus the number of watchdog-timer ticks remaining before input operation times out. Negative-to-zero transition causes the timeout to occur.
XTCC$L_OUT_WAIT_TICKS	Zero minus the number of watchdog-timer ticks remaining before output operation times out. Negative-to-zero transition causes the timeout to occur.

3.1.4 XTCB Data Structure

The transport layer uses transport communication buffers (XTCBs) to pass data between the server or Xlib and the underlying transport. Each connection can have small and large sizes of XTCBs, and the sizes may be different for different connections. A connection's XTCBs exist for the duration of the connection and are user-modifiable.

The XTCB structure is shown in Figure 3–8.

Figure 3–8 XTCB Data Structure

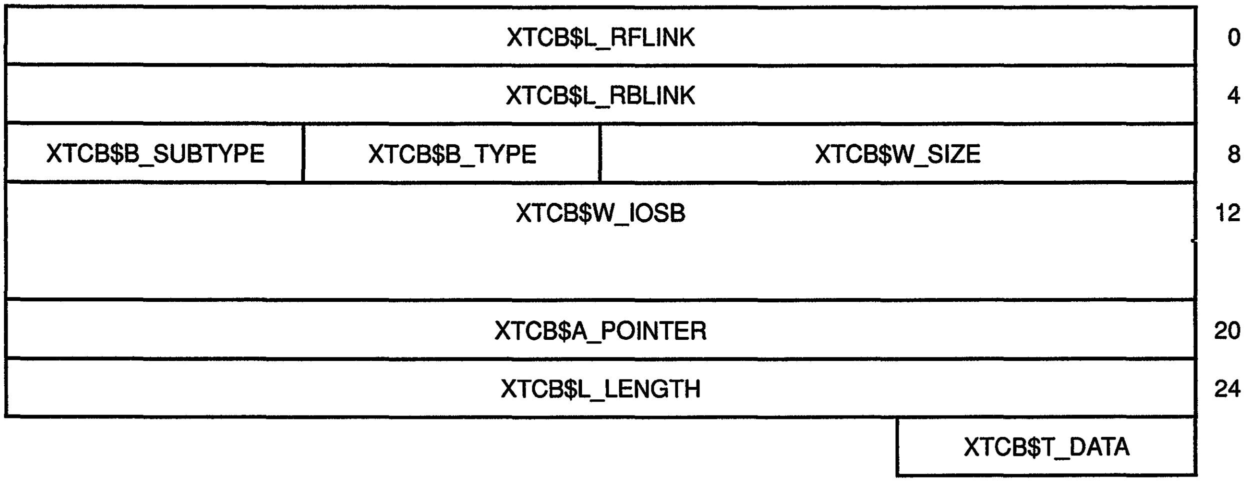


Table 3–6 shows the contents of the XTCB data structure.

Table 3–6 XTCB Data Structure

Field	Use
XTCB$L_RFLINK	Forward, relative pointer to next XTCB in the queue.
XTCB$L_RBLINK	Backward, relative pointer to previous XTCB in the queue.
XTCB$W_SIZE	Length of XTCB in bytes.
XTCB$B_TYPE	Constant 255.
XTCB$B_SUBTYPE	Constant DECW$C_DYN_XTCB_SRP (3) or DECW$C_DYN_XTCB_LRP (4). DECW$C_DYN_XTCB_SRP represents a small XTCB; DECW$C_DYN_XTCB_LRP represents a large XTCB.
XTCB$W_IOSB	IOSB for use by specific transport when performing the requested I/O operation with this XTCB.
XTCB$A_POINTER	Pointer to next unused byte in data area.
XTCB$L_LENGTH	Length, in bytes, of valid data in data area.
XTCB$T_DATA	Start of variable-length data area.

3.1.5 XTCQ Data Structure

The transport communication queue (XTCQ) data structure contains the six per-connection communication queues and their state information. The six per-connection communication queues, which are described in more detail in Section 3.2, are as follows:

- Event Work Queue. Identified by the XTCQ$L_EW_RFLINK and XTCQ$L_EW_RBLINK fields in the XTCQ.
- Event Free Queue (small and large). Identified by the XTCQ$L_EFS_RFLINK, XTCQ$L_EFS_RBLINK, XTCQ$L_EFL_RFLINK, and XTCQ$L_EFL_RBLINK fields in the XTCQ.
- Request Work Queue. Identified by the XTCQ$L_RW_RFLINK and XTCQ$L_RW_RBLINK fields in the XTCQ.
- Request Free Queue (small and large). Identified by the XTCQ$L_RFS_RFLINK, XTCQ$L_RFS_RBLINK, XTCQ$L_RFL_RFLINK, and XTCQ$L_RFL_RBLINK fields in the XTCQ.

The XTCQ data structure exists during the lifetime of a connection and is user-modifiable.

The XTCQ data structure is shown in Figure 3–9.

Figure 3–9 XTCQ Data Structure

XTCQ$L_EW_RFLINK	0
XTCQ$L_EW_RBLINK	4
XTCQ$L_EFS_RFLINK	8
XTCQ$L_EFS_RBLINK	12
XTCQ$L_EFL_RFLINK	16
XTCQ$L_EFL_RBLINK	20
XTCQ$L_RW_RFLINK	24
XTCQ$L_RW_RBLINK	28
XTCQ$L_RFS_RFLINK	32
XTCQ$L_RFS_RBLINK	36
XTCQ$L_RFL_RFLINK	40
XTCQ$L_RFL_RBLINK	44
XTCQ$L_FLAGS	48

Table 3–7 shows the contents of the XTCQ data structure.

Table 3–7 XTCQ Data Structure

Field	Use
XTCQ$L_EW_RFLINK	Event work queue relative queue header.
XTCQ$L_EW_RBLINK	Event work queue relative queue header.
XTCQ$L_EFS_RFLINK	Small XTCB event free queue relative queue header.
XTCQ$L_EFS_RBLINK	Small XTCB event free queue relative queue header.
XTCQ$L_EFL_RFLINK	Large XTCB event free queue relative queue header.
XTCQ$L_EFL_RBLINK	Large XTCB event free queue relative queue header.
XTCQ$L_RW_RFLINK	Request work queue relative queue header.
XTCQ$L_RW_RBLINK	Request work queue relative queue header.
XTCQ$L_RFS_RFLINK	Small XTCB request free queue relative queue header.

(continued on next page)

Table 3–7 (Cont.) XTCQ Data Structure

Field	Use
XTCQ$L_RFS_RBLINK	Small XTCB request free queue relative queue header.
XTCQ$L_RFL_RFLINK	Large XTCB request free queue relative queue header.
XTCQ$L_RFL_RBLINK	Large XTCB request free queue relative queue header.
XTCQ$L_FLAGS	See the following list.
The following fields are defined within XTCQ$L_FLAGS:	
XTCQ$V_EWQ_FLAG	When 1, notification is desired when an XTCB is inserted on an empty event work queue.
XTCQ$V_EFSQ_FLAG	When 0, notification is desired when an XTCB is inserted on an empty small XTCB event free queue.
XTCQ$V_EFLQ_FLAG	When 0, notification is desired when an XTCB is inserted on an empty large XTCB event free queue.
XTCQ$V_RWQ_FLAG	When 1, notification is desired when an XTCB is inserted on an empty request work queue.
XTCQ$V_RFSQ_FLAG	When 0, notification is desired when an XTCB is inserted on an empty small XTCB request free queue.
XTCQ$V_RFLQ_FLAG	When 0, notification is desired when an XTCB is inserted on an empty large XTCB request free queue.
XTCQ$V_ABORT_FLAG	Reserved for use by Digital.

3.1.6 XTDB Data Structure

The transport descriptor block (XTDB) data structure describes each attached transport. The XTDB keeps track of resources required by transport-specific functions, such as the known-object mailbox channel in the case of a DECnet server, and objects that will be cloned on a per-connection basis, such as the transport-specific XTFT function table.

The XTDB structure exists during the lifetime of a transport. The XTDB structure is allocated from memory pages that have user-mode read access/executive-mode write access (UREW). The common transport layer maintains a queue of all XTDBs.

The XTDB data structure is shown in Figure 3–10.

Figure 3–10 XTDB Data Structure

XTDB$A_FLINK			0
XTDB$A_BLINK			4
XTDB$B_SUBTYPE	XTDB$B_TYPE	XTDB$W_SIZE	8
XTDB$L_FLAGS			12
XTDB$A_TPB			16
XTDB$L_REF_COUNT			20
XTDB$Q_RESERVED			24
XTDB$L_DEC_RESERVED			32
XTDB$A_ITCC_FLINK			36
XTDB$A_ITCC_BLINK			40
XTDB$A_CONNECT_ABORT			44
XTDB$A_CONNECT_REQUEST			48
XTDB$L_FAMILY_NAME_LEN			52
≈ XTDB$T_FAMILY_NAME (16 bytes) ≈			56
XTDB$L_CHK_OBJTYP			72
XTDB$L_CHK_OBJNAM_LEN			76
XTDB$A_CHK_OBJNAM_ADR			80
XTDB$L_CHK_ACCESS			84
XTDB$L_CHK_FLAGS			88
XTDB$A_XPORT_TABLE			92

Table 3–8 shows the contents of the XTDB data structure.

Table 3–8 XTDB Data Structure

Field	Use
XTDB$A_FLINK	Forward, absolute pointer to next XTDB in queue.
XTDB$A_BLINK	Backward, absolute pointer to previous XTDB in queue.
XTDB$W_SIZE	Length of XTDB in bytes.
XTDB$B_TYPE	Constant 255.
XTDB$B_SUBTYPE	Constant DECW$C_DYN_XTDB (5).
XTDB$L_FLAGS	See the following list.
The following fields are defined within XTDB$L_FLAGS:	
XTPB$V_MODE	Type of transport connection. Possible values are: • DECW$K_XPORT_REMOTE_SERVER (0) • DECW$K_XPORT_REMOTE_CLIENT (1) • DECW$K_XPORT_LOCAL_SERVER (2) • DECW$K_XPORT_LOCAL_CLIENT (3)
XTDB$V_ACTIVE	Specific transport image has been activated and is running.
XTDB$V_DYING	Specific transport is aborting and no further operations should be allowed. Connections using this transport will be disconnected.
XTDB$A_TPB	Address of the transport-specific XTPB.
XTDB$L_REF_COUNT	Number of connections using this transport. Should be equivalent to the number of IXTCCs enqueued on the IXTCC queue header.
XTDB$Q_RESERVED	Quadword reserved for use by specific transport.
XTDB$L_DEC_RESERVED	Reserved for use by Digital.
XTDB$A_ITCC_FLINK	Absolute queue header for IXTCCs of connections using this transport.
XTDB$A_ITCC_BLINK	Absolute queue header for IXTCCs of connections using this transport.
XTDB$A_CONNECT_ABORT	Address of either the server or Xlib connection abort notification AST routine. Called with one argument: the XTCC by reference.
XTDB$A_CONNECT_REQUEST	Address of the server's connection request notification AST routine. Called with one argument: the XTCC by reference.

(continued on next page)

Table 3–8 (Cont.) XTDB Data Structure

Field	Use
XTDB$L_FAMILY_NAME_LEN	Length of the transport name string.
XTDB$T_FAMILY_NAME	Contains the transport family name string (for example, "DECNET" or "TCPIP"). The maximum size of the family name is 16 bytes, as determined by the constant XTDB$S_FAMILY_NAME. XTDB$S_FAMILY_NAME is defined in DECW$EXAMPLES:XPORTEXAMPLEDEF.R32.
XTDB$A_XPORT_TABLE	Address of the XTFT structure for the specific transport. Returned as a value of the DECW$TRANSPORT_INIT routine, which is provided as a global name in every transport-specific image.

3.1.7 XTFT Data Structure

Each specific transport shareable image provides a set of transport-specific routines that are used for all connections using that transport. The transport function table (XTFT) data structure contains the addresses of these routines. During a transport attach operation, the transport-specific DECW$TRANSPORT_INIT routine initializes and returns that transport's XTFT.

The XTFT data structure exists during the lifetime of a specific transport and is accessible by the common transport. The XTFT data structure is shown in Figure 3–11.

Figure 3–11 XTFT Data Structure

XTFT$L_REQUIRED0	0
XTFT$L_RESERVED0	4
XTFT$A_EXECUTE_WRITE	8
XTFT$A_WRITE	12
XTFT$A_WRITE_USER	16
XTFT$A_EXECUTE_FREE	20
XTFT$A_FREE_INPUT_BUFFER	24
XTFT$A_CLOSE	28
XTFT$A_OPEN	32

(continued on next page)

Figure 3–11 (Cont.) XTFT Data Structure

XTFT$A_ATTACH_TRANSPORT	36
XTFT$A_RUNDOWN	40
XTFT$L_XTCC_LENGTH	44
XTFT$L_XTPB_LENGTH	48
XTFT$L_XTDB_LENGTH	52
XTFT$L_IXTCC_LENGTH	56
XTFT$L_REQUIRED1	60

Table 3–9 shows the contents of the XTFT data structure.

Table 3–9 XTFT Data Structure

Field	Use
XTFT$L_REQUIRED0	Longword that must contain the value XTFT$K_REQUIRED0 (–1515870811 decimal).
XTFT$L_RESERVED0	Longword reserved for use by a specific transport.
XTFT$A_EXECUTE_WRITE	Address of execute-write routine.
XTFT$A_WRITE	Address of write routine.
XTFT$A_WRITE_USER	Address of write-user routine.
XTFT$A_EXECUTE_FREE	Address of execute-free routine.
XTFT$A_FREE_INPUT_BUFFER	Address of free-input routine.
XTFT$A_CLOSE	Address of close routine.
XTFT$A_OPEN	Address of open routine.
XTFT$A_ATTACH_TRANSPORT	Address of attach routine.
XTFT$A_RUNDOWN	Address of rundown routine.
XTFT$L_XTCC_LENGTH	Length, in bytes, of XTCCs used by this transport. Must be at least XTCC$W_SIZE.
XTFT$L_XTPB_LENGTH	Length, in bytes, of XTPBs used by this transport. Must be at least XTPB$W_SIZE.
XTFT$L_XTDB_LENGTH	Length, in bytes, of XTDBs used by this transport. Must be at least XTDB$W_SIZE.

(continued on next page)

Table 3–9 (Cont.) XTFT Data Structure

Field	Use
XTFT$L_IXTCC_LENGTH	Length, in bytes, of IXTCCs used by this transport. Must be at least IXTCC$W_SIZE.
XTFT$L_REQUIRED1	Longword that must contain the value XTFT$K_REQUIRED1 (–1768515946 decimal).

3.2 Transport Layer Communication Queues

When a connection is created, a specific transport calls the DECW$XPORT_ALLOC_INIT_QUEUES routine to create six communication queues that pass XTCBs between servers and clients and between transport-common and transport-specific components. The communication queues are relative queues as used by the VAX REMQxI and INSQxI instructions. With the exception of the local transport, the queues are private to one side of the connection.

The communication queue headers are stored in the XTCQ data structure. Each queue is named according to its function, as shown in Table 3–10.

A queue element (XTCB) must be removed from its queue before its contents can be modified by either the server or Xlib or by transport layer internal functions.

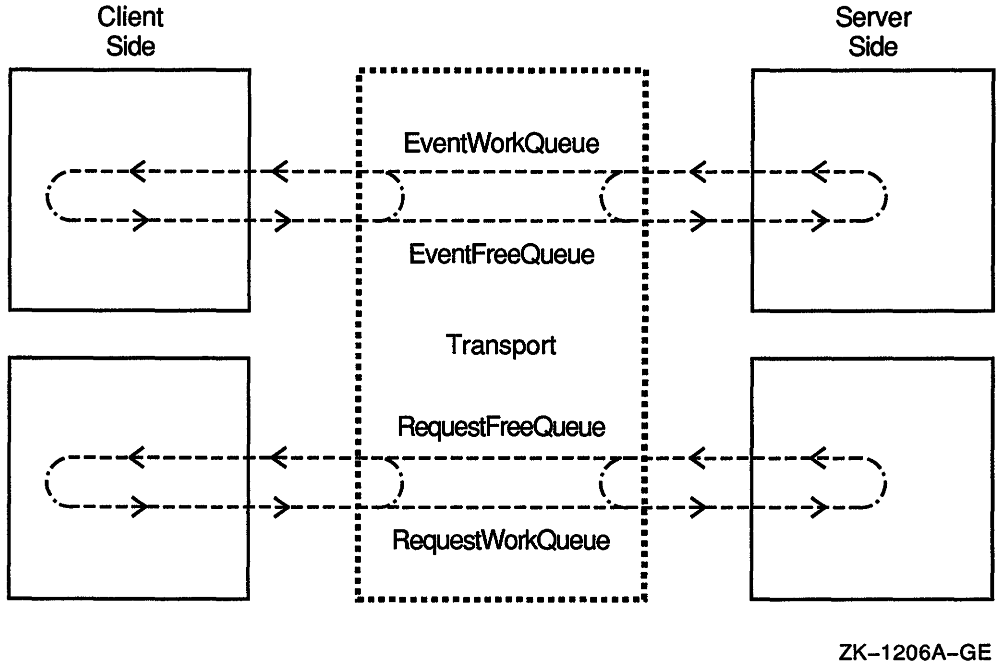
The component that adds XTCBs to a queue is called the **producer**; the component that removes XTCBs from a queue is called the **consumer**.

Table 3–10 Transport Layer Communication Queues

Queue Name	Function
EventWorkQueue	Queue of XTCBs containing events, errors, and replies to be sent to Xlib
EventFreeQueue large and small	Return queue for processed XTCBs from the EventWorkQueue
RequestWorkQueue	Queue of XTCBs containing requests to be sent to a server
RequestFreeQueue large and small	Return queue for processed XTCBs from the RequestWorkQueue

The transport layer communication queues are shown in Figure 3–12. For the purpose of simplicity, Figure 3–12 combines the free queues; there are actually separate free queues for large and small XTCBs.

Figure 3–12 Transport Layer Communication Queues



You can consider the buffers in the transport layer communication queues to be in an infinite loop similar to a ski lift; once a buffer is received at one end of the connection it is emptied and added to the returning loop.

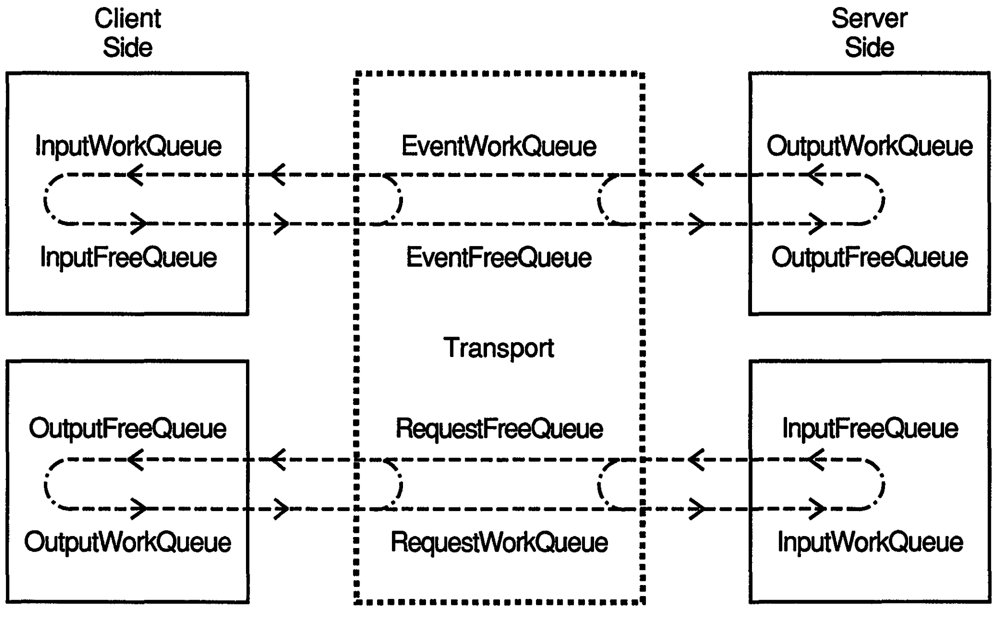
The transport layer further distinguishes between the queues so that terminology is consistent on both sides of the client/server connection; when the term "input" is used in the context of a client, it means "event." Conversely, "input" in the context of the server means "request." Table 3–11 shows the delineation of the queues.

Table 3–11 Views for the Client/Server Communication Queue

General Queue Name	Client View	Server View
EventWorkQueue	InputWorkQueue	OutputWorkQueue
EventFreeQueue	InputFreeQueue	OutputFreeQueue
RequestWorkQueue	OutputWorkQueue	InputWorkQueue
RequestFreeQueue	OutputFreeQueue	InputFreeQueue

Figure 3–13 shows the result of adding the client/server views to the model of the queues shown in Figure 3–12.

Figure 3–13 Client/Server Communication Queue Views



In the case of local communication where the client and server are executing on the same hardware, the queues are shared between the client and server. Each queue has two meanings depending upon whether a server or a client is looking at the queue. For example, the EventWorkQueue is simultaneously the client's InputWorkQueue and the server's OutputWorkQueue.

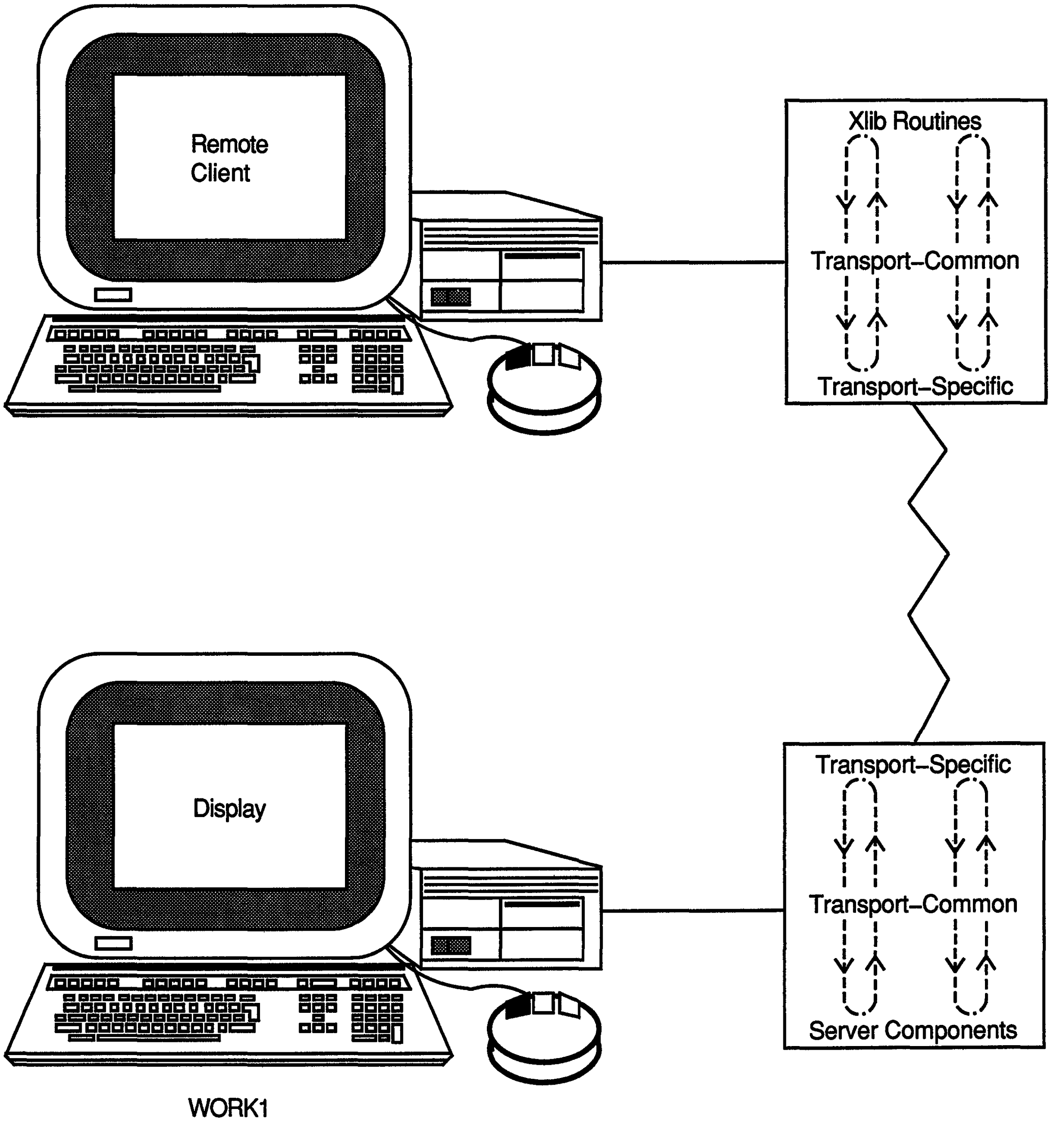
3.2.1 Transport Common/Specific Queue Relationship

The transport layer seems to be a single entity that somehow moves data across the wire, but there are actually two transport layers involved: a transport layer on the client side and a transport layer on the server side. Each side of the client/server connection initializes the common transport and attaches one or more specific transports. Each transport layer builds and maintains its own set of connection queues; the queues are private to each side of the wire and are not actually shared across a network connection. The local transport is an exception; the queues are built in shared global sections.

General discussions of the relationship between the transport layer and other DECwindows components commonly treat the transport layer as a single entity that exists on both sides of the client/server connection. However, the transport layer consists of the transport-common and transport-specific components.

The transport-common and transport-specific components pass data to each other by means of a queue implementation that is similar to the queues employed between Xlib and the transport-common code. This intratransport queuing implementation is shown in Figure 3–14.

Figure 3–14 Transport Common/Specific Connection



ZK–1207A–GE

3.2.2 Adding and Removing Buffers from the Queues

The server and Xlib call transport-common routines such as DECW$XPORT_READ, DECW$XPORT_FREE_INPUT_BUFFER, GET_OUTPUT_BUFFER, and DECW$XPORT_CHAINED_WRITE to remove or add buffers from the queues, as described in Section 3.3.1.6 and Section 3.3.1.7.

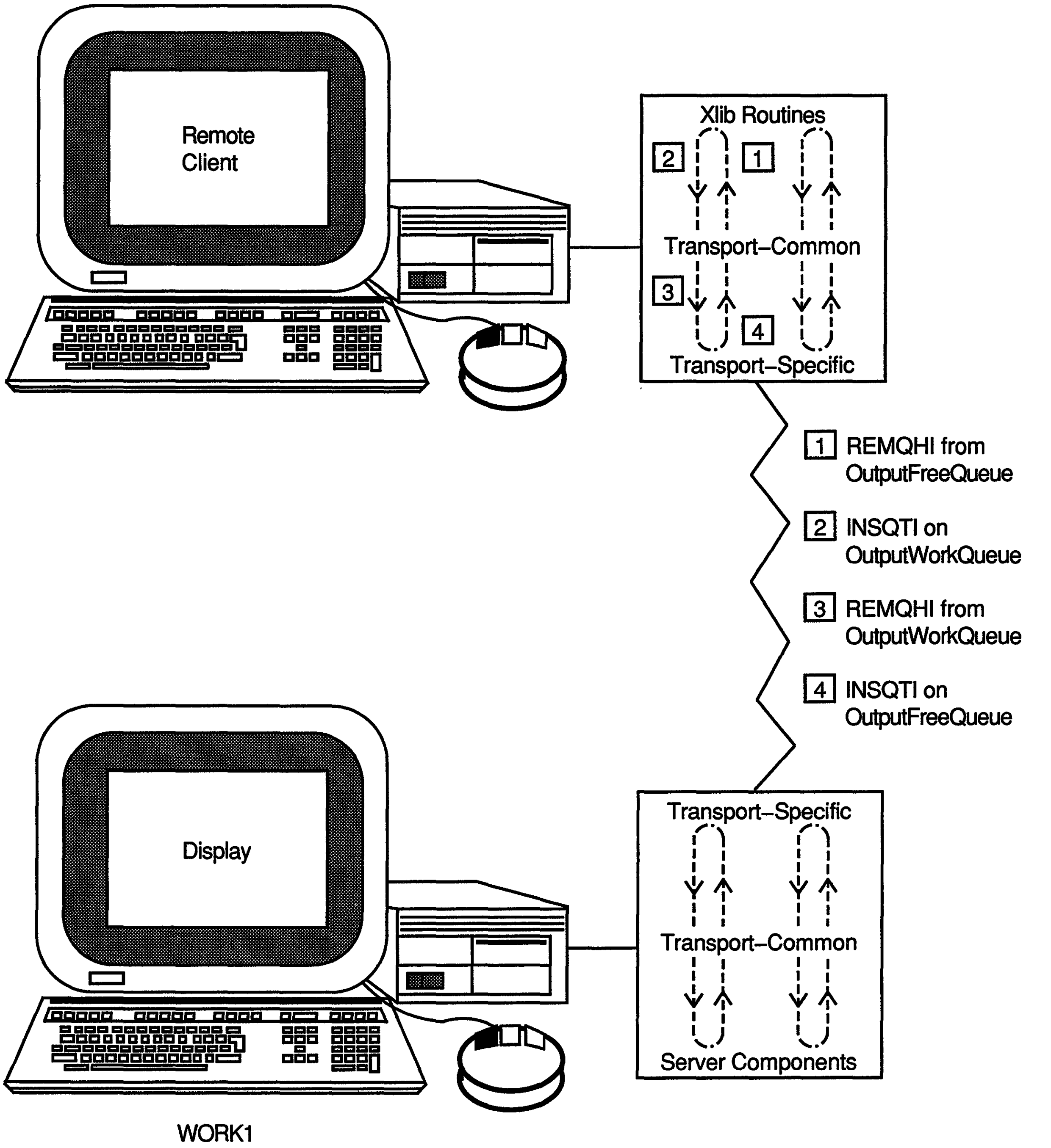
Some of the transport-common routines may call transport-specific routines such as XTFT$A_WRITE to actually get the buffer from the queue. Other transport-common routines such as DECW$XPORT_READ operate only in the common layer and communicate with the specific transport only by manipulating data structures.

An example of the transport-common/transport-specific queue process is as follows:

1. Xlib calls the DECW$XPORT_GET_OUTPUT_BUFFER routine to get a free XTCB. DECW$XPORT_GET_OUTPUT_BUFFER removes (REMQHI) an XTCB from the OutputFreeQueue.
2. Xlib copies data to the XTCB and calls DECW$XPORT_CHAINED_WRITE.
3. DECW$XPORT_CHAINED_WRITE inserts (INSQTI) the XTCB on the OutputWorkQueue. If the queue was empty, the XTFT$A_WRITE routine is called to determine if an I/O should be initiated.
4. When the write completes, the XTCB is inserted on the tail of the OutputFreeQueue. Xlib is then notified if it is waiting for an output buffer. If the OutputWorkQueue is not empty, another I/O is initiated.

Figure 3–15 adds the transport-common/transport-specific queue process to Figure 3–14.

Figure 3–15 Transport Common/Specific Queues



ZK–1208A–GE

On the server side of the connection, the underlying transport notifies the transport-specific layer that data is available. For example, DECnet delivers this notification in the form of a $QIO read completion for the connection; that is, the specific transport receives a read-completion AST in response to a $QIO read for a given connection.

When the specific transport receives a read completion, it performs the following steps:

1 Inserts (INSQTI) the XTCB that now has data in it on the InputWorkQueue.

2 If input notification is requested, it sends the server an input-notify AST to indicate that an XTCB is available for the connection.

3 Removes (REMQHI) an XTCB from the InputFreeQueue.

4 Initiates a new read into the XTCB.

The server calls DECW$XPORT_READ to remove (REMQHI) the XTCB from the InputWorkQueue. When the dispatcher is finished with the XTCB, it calls DECW$XPORT_FREE_INPUT_BUFFER to insert the XTCB on the InputFreeQueue, and the queue cycle is complete.

If the InputFreeQueue is empty and input for the connection is enabled when DECW$XPORT_FREE_INPUT_BUFFER inserts the XTCB, DECW$XPORT_FREE_INPUT_BUFFER calls XTFT$A_EXECUTE_FREE to remove (REMQHI) the XTCB it just placed on the InputFreeQueue. In the case of DECnet or TCP/IP, the XTCB is then used to store the result of the next QIO read operation for the connection.

3.2.3 Communication Queue Notification Flags

It is possible that at any given time all of a connection's input or output XTCBs will be in use. To provide for this possibility, there is a flag associated with each queue that tracks the state of the queue. When a queue has been exhausted, the component that is removing XTCBs from the queue (the consumer) sets the state flag to indicate that it wants to be notified when an XTCB becomes available.

If the component that is adding XTCBs to the queue (the producer) notices that the queue was empty, it tests the state flag and, if set, notifies the consumer that an XTCB is available.

The notification is context dependent. When the consumer is a transport-specific layer, as described in Section 3.2.1, the notification causes I/O to be initiated using the recently inserted queue element.

When the consumer is the transport-common layer, notification consists of delivering an AST or setting an event flag. For example, the transport-specific code on the server side of the connection sends the transport-common layer an AST to say that input is available for a connection.

The notification flags are shown in Table 3–12.

Table 3–12 Communication Queue Notification Flags

Flag Name	Meaning
EventWorkQueueFlag	Event consumer awaits event XTCBs
EventFreeQueueFlag	Event producer awaits returned event XTCBs
RequestWorkQueueFlag	Request consumer awaits request XTCBs
RequestFreeQueueFlag	Request producer awaits returned request XTCBs

As in the case of the local communication queues, each interest flag in a local connection has two meanings depending upon whether a server or a client is looking at the queue: the EventWorkQueue is the client's InputWorkQueue and the server's OutputWorkQueue; the RequestFreeQueueInterest bit is the client's OutputFreeQueueInterest bit and the server's InputFreeQueueInterest bit.

A queue element consumer functions as follows:

1 Sets the queue notification flag (in case the queue is empty).

2 Removes an element from the queue—

- If the queue is empty, the consumer can wait for notification that data is available or it can return without the buffer.
- If the queue is not empty, the consumer clears the notification flag and returns with the buffer.

A queue element producer functions as follows:

1 Inserts an element on the queue.

2 If the queue was empty and the queue flag was set, clears the queue flag and generates a notification to tell the consumer that data is available.

3.2.4 Preventing Queue Access Conflict

The transport layer uses interlocked queues to synchronize access between the queue producer and consumer. Server or Xlib attempts to remove and insert buffers result in the interlocked instructions being used in user mode, possibly at AST level.

The queues are also accessed by inner-mode routines using the example_ queue emulations found in DECW$EXAMPLES:XPORT_EXAMPLE_ QUEUE. Executive-mode code cannot use the REMQxI and INSQxI built-ins to access a relative queue that is modifiable by user-mode code because of the possibility of overwriting user-write-protected memory.

The XPORT_EXAMPLE_QUEUE.MAR module provides emulations of the REMQxI and INSQxI instructions that perform appropriate probing. The REMQxI and INSQxI emulations use the VAX PROBEW instruction to probe the memory occupied by a queue entry to determine if it has user-mode write access and return an ACCVIO status if the memory is not accessible.

The REMQxI and INSQxI emulations also perform a spin-and-wait to see if the queue is locked by another transport user.

3.2.4.1 Special-Case Queue Conditions

There are two special-case queue conditions:

- As described in Section 3.2.3, the queue consumer sets the notification (notify-if-empty) queue flag before attempting a REMQHI from the queue. If the queue is not empty, the consumer removes the queue element and clears the flag.

  The producer checks this flag after adding an element to the queue. If the producer is able to insert a queue element between the time when the consumer sets and clears the flag, the producer generates an input-available notification when none is necessary. This is called **spurious notification**, and queue consumers must be prepared for it.

- A second condition occurs when the consumer can preempt the producer thread, such as the output half of a remote transport.

  For example, assume that the producer does an INSQTI of some element on the queue and is about to test the queue flag. However, before the producer can test the queue flag, the consumer does the following:

  1. Removes (REMQHI) an XTCB and processes it
  2. Fails when it attempts to remove another XTCB
  3. Sets the queue interest flag and quits

  The producer does not know that the queue is now empty; it continues to run and attempts to notify the consumer. In this case, however, notification consists of removing the head queue element and issuing a transport write operation from data in the queue element. Because the element is no longer on the queue, the notification procedure must be prepared to handle the empty queue.

3.3 Transport-Common and Transport-Specific Components

As described in Section 1.2, the transport layer is separated into transport-common and transport-specific functions. The routines that support the transport-common functions have names of the form **DECW$XPORT_xxx** and provide the generic services needed by Xlib or the server.

Some transport-common routines only select and call the correct transport-specific routine. Other transport-common routines perform substantial processing prior or subsequent to invoking the associated transport-specific routine.

A special set of utility routines and macro definitions perform thread suspension and resumption, global section mapping and maintenance, queue maintenance, and communication between transport layer components for use by transport layer developers. See Chapter 7 for more information.

The transport-common routines call transport-specific routines that are private to a particular transport service, such as DECnet. The addresses of the transport-specific routines are contained in the XTFT data structure and, along with VMS packaging requirements, comprise the interface to which a transport developer must program.

3.3.1 Transport-Common Functions

The transport-common code performs the following functions:

- Initializes the transport-specific layer
- Attaches the transport-specific layer
- Opens a connection
- Gets and sets transport-common and per-connection attributes
- Allocates memory
- Reads from the input queue
- Writes to the output queue
- Closes a connection

Subsequent sections describe the transport-common functions.

3.3.1.1 Initializing the Transport-Common Layer

The DECW$XPORT_INITIALIZE routine is called before any other DECwindows transport-common routine. Xlib and the server call the DECW$XPORT_INITIALIZE routine to initialize the transport-common code. The common transport knows whether it was called by Xlib or the server by a **caller_type** argument that identifies the caller as a client (DECW$K_XPORT_CLIENT) or as the server (DECW$K_XPORT_SERVER).

DECW$XPORT_INITIALIZE initializes the global XTPB data structure, from which other XTPB data structures inherit their default values. (See Section 3.1.1.)

Transports and connections that are subsequently attached or opened inherit the parameters set at initialization time unless they override them in an **itmlst** argument to the DECW$XPORT_INITIALIZE or DECW$XPORT_ATTACH_TRANSPORT routines. Both Xlib and the server override some of these defaults.

The DECW$XPORT_INITIALIZE routine calls the DECW$XPORT_SET_ATTRIBUTES routine to load the global XTPB data structure with the attributes passed in the **itmlst** argument.

The **itmlst** argument can specify the following parameters:

- The address of a procedure to call when an input operation is completed on the connection and input notification has been enabled
- The address of a procedure to call when an output operation is completed on the connection and output notification has been enabled

- The size of the data area of the large communication buffers used by transport
- The size of the data area of the small communication buffers used by transport
- A value to be passed to the input notification routine
- A value to be passed to the output notification routine
- The number of the event flag to be set for input notification
- The number of the event flag to be set for output notification
- The number of seconds the default waiting procedures are allowed to wait for output completion
- The number of seconds the default waiting procedures are allowed to wait for input completion

3.3.1.2 Attaching a Transport-Specific Layer

On the client side of the connection, Xlib calls DECW$XPORT_ATTACH_TRANSPORT to attach and initialize only those transports to which it wants to connect. Xlib determines the transports to attach and initialize as follows:

- If the call to the Xlib OPEN DISPLAY routine specifies a display name, Xlib parses the display name to determine the transport to initialize.
- If the call to the Xlib OPEN DISPLAY routine specifies a null display name, Xlib uses the result of the last SET DISPLAY command to determine the transport to initialize.

On the server side of the connection, the server must tell the common transport which specific transports to attach and initialize. The server uses the logical name DECW$SERVER_TRANSPORTS (see SYS$MANAGER:DECW$PRIVATE_SERVER_SETUP.TEMPLATE) to accomplish this.

For example, DECW$SERVER_TRANSPORTS could translate to "DECNET,LOCAL,TCPIP". The server calls the common transport DECW$XPORT_ATTACH_TRANSPORT routine for each transport identified by the logical name. The **transport_name** argument specifies the transport, such as "DECNET".

DECW$XPORT_ATTACH_TRANSPORT needs a way to associate the transport name specified in the **transport_name** argument with a specific transport's image. When called by either Xlib or the server, DECW$XPORT_ATTACH_TRANSPORT attempts to locate and activate an image with a name in the form SYS$SHARE:DECW$TRANSPORT_**transport_name**.EXE. If it does not find one, DECW$XPORT_ATTACH_TRANSPORT looks for a name in the form SYS$SHARE:DECW_TRANSPORT_**transport_name**.EXE.

For example, if DECW$XPORT_ATTACH_TRANSPORT could not find an image with the name SYS$SHARE:DECW$TRANSPORT_FOO.EXE, it would look for SYS$SHARE:DECW_TRANSPORT_FOO.EXE.

Note: Transport names that contain a dollar sign ($) character, such as SYS$SHARE:DECW$TRANSPORT_DECNET, are reserved for transport images supplied by Digital.

Transport names that do not contain a "$" character are reserved for third-party and customer transport images. These transport names must be in the form SYS$SHARE:DECW_TRANSPORT_transport_name.EXE.

If the image activation is successful, DECW$XPORT_ATTACH_TRANSPORT builds an XTDB, initializes the common fields, and calls the transport-specific DECW$TRANSPORT_INIT routine to complete the initialization by initializing the XTFT data structure with the addresses of the transport-specific routines.

Every transport-specific image (SYS$SHARE:DECW$TRANSPORT_**transport_name**.EXE or SYS$SHARE:DECW_TRANSPORT_**transport_name**.EXE) must provide an implementation of DECW$TRANSPORT_INIT for its initialization routine. A transfer vector to the DECW$TRANSPORT_INIT routine must be placed in the first image section of the transport-specific image.

The DECW$XPORT_ATTACH_TRANSPORT routine performs the following functions:

- Validates the **transport_name** argument.
- Checks to see if the transport-specific image is already attached.
- Builds the transport-specific image file name.
- Activates the transport-specific image.
- Calls the transport-specific DECW$TRANSPORT_INIT initialization routine to get the address of the XTFT.
- Allocates memory for XTDB and XTPB. Copies the contents of the global XTPB to the new XTPB.
- Attaches the XTPB to the XTDB and sets the XTDB attributes from the **itmlst** argument.
- Fills in the XTDB contents, such as the transport family name.
- Enqueues the XTDB on the global XTDB queue.
- Calls the transport-specific attach routine, XTFT$A_ATTACH_TRANSPORT, and returns the status.

3.3.1.3 Opening a Connection

Xlib clients call the Xlib OPEN DISPLAY routine to establish a connection with a server. Xlib in turn calls the DECW$XPORT_ATTACH_TRANSPORT and DECW$XPORT_OPEN routines.

The DECW$XPORT_OPEN routine performs the following functions:

- Tries to find a transport that will initiate a connection. DECW$XPORT_OPEN searches the global XTDB queue for a transport whose family name matches the **xportnam** argument. **xportnam** is typically the transport specified in the last use of the SET DISPLAY command (for example, "DECNET").

- Copies the attached transport parameters to connection-specific parameters. Allocates IXTCC and XTPB data structures for the connection and partially establishes the connection context.
- Updates the connection parameters if an **itmlst** argument was specified.
- Calls the transport-specific open routine, XTFT$A_OPEN, to try to open the connection.

3.3.1.4 Getting and Setting Transport Attributes

The DECW$XPORT_SET_ATTRIBUTES and DECW$XPORT_GET_ATTRIBUTES routines get and set XTPB and XTCC attributes associated with a connection-specific XTPB or the global XTPB. (It is not possible to specify the transport-specific XTPB.) When the **XTCC** argument specifies an active connection, the parameters of that connection are modified or returned; when **XTCC** is zero, the global parameters are modified or returned.

DECW$XPORT_SET_ATTRIBUTES performs some checks to detect invalid combinations of parameters.

3.3.1.5 Allocating Transport Memory

The common transport uses the routines described in Table 3–13 to allocate and deallocate memory.

Note: **The VMS Run-Time Library memory allocation routines may not be used from inner access modes.**

Table 3–13 Transport Memory Allocation Routines

Routine Name	Description
DECW$XPORT_ALLOC_INIT_QUEUES	Allocates storage for an XTCC, XTCQ, and all of the XTCBs for a connection. These structures are user-writable. Places all of the XTCBs on the appropriate free queues.
DECW$XPORT_ALLOC_PMEM	Allocation routine for protected (user-readable, executive-writable) structures. Allocates a block of storage of a given size.
DECW$XPORT_DEALLOC_PMEM	Memory deallocation routine (companion to DECW$XPORT_ALLOC_PMEM). Deallocates previously allocated structure.
DECW$XPORT_DEALLOC_QUEUES	Deallocates a block of storage previously allocated for the queues of a connection.

3.3.1.6 Common Transport Read Routines

The common transport performs a read operation to remove an input XTCB or return an input XTCB to the queue. There are two read routines: DECW$$XPORT_FREE_INPUT and DECW$XPORT_READ. The read routines are described in Table 3–14.

Table 3–14 Transport Read Routines

Routine Name	Description
DECW$$XPORT_FREE_INPUT	Returns an XTCB aquired with DECW$XPORT_READ to the input free queues. If no XTCB was on the queue and notification is desired, DECW$$XPORT_FREE_INPUT removes the XTCB and initiates a read operation on the connection; that is, it calls the transport-specific XTFT$A_FREE_INPUT_BUFFER routine, which in turn does a read operation for the underlying transport.
DECW$XPORT_READ	Attempts to remove an XTCB from the head of the input work queue. If the attempt succeeds, the address of the XTCB is returned to the caller and it returns with successful status. If DECW$XPORT_READ fails, one of the following actions occurs: • If it fails and is a nonblocking operation, input notification is enabled and it returns failed status. • If it fails and is a blocking operation, the call uses a $SYNCH system service call to wait for an input buffer to become available.

3.3.1.7 Writing to the Transport

The common transport includes routines to write data from the user's environment to a connection. The transport-common routines in turn call transport-specific routines to send the data across the wire.

The common transport write routines are described in Table 3–15.

Table 3–15 Common Transport Write Routines

Name	Description
DECW$XPORT_GET_OUTPUT_BUFFER	Gets an XTCB from the output free queue. The mode argument modifies the operation: • If the no-block bit is set and no XTCB is available when the call is made, the routine returns a buffer-not-available status and output notification is enabled. • If the no-block bit is clear, a call to $SYNCH is made to wait for a buffer to become available. If timeouts are enabled, it is possible for the $SYNCH call to time out. The data-length argument provides a hint as to what size buffer the caller should receive.

(continued on next page)

Table 3–15 (Cont.) Common Transport Write Routines

Name	Description
DECW$XPORT_WRITE	Initiates a writing operation on the connection associated with an XTCC. The DECW$XPORT_WRITE routine copies the data from a buffer provided by the user to XTCBs. DECW$XPORT_WRITE supports both blocking and nonblocking modes. DECW$XPORT_WRITE may perform multiple callback operations to the user's callback routine, specified by the **copy_rtnadr** argument, to get data copied from the caller's environment into the XTCBs. **copy_rtnadr** is called with the address of an XTCB as an argument and the user-specified **copy_rtnarg** argument. The transport user is expected to partially fill the XTCB and return with the status of the write request. Works similarly to DECW$XPORT_COPY_AND_WRITE but allows the caller to copy data from noncontiguous structures or compute the data dynamically when that is more practical.
DECW$XPORT_COPY_AND_WRITE	Initiates a writing operation on the connection associated with the XTCC. DECW$XPORT_COPY_AND_WRITE performs a buffered write operation and returns the size of the data actually copied in the **retbuflen** argument.
DECW$XPORT_CHAINED_WRITE	Initiates a writing operation on the connection associated with the XTCC. The **itmlst** argument specifies a number of buffers to be written to the connection. Two types of buffers are supported: XTCBs and user buffers. If the specific transport being used does not support writing from the user's buffer, DECW$XPORT_CHAINED_WRITE performs a copy operation using DECW$XPORT_COPY_AND_WRITE.

3.3.1.8 Transport Layer Timer Mechanism

The transport layer timer mechanism is used to create an inner-mode AST at 5-second intervals so that the transport-common layer can search through the transport descriptors and connection contexts to find work that needs to be done. This timer mechanism is particularly useful in generating timeouts for $SYNCH operations, such as XPORT_IN_NOTIFY_WAIT, that have gone on too long.

When the time period expires, the timer finds connections that have been waiting for the number of ticks specified in the connection's XTPB. The wait operations are completed by assigning a SS$_TIMEOUT status to the appropriate I/O status block (IOSB), setting an event flag, and setting the XTCC$V_DYING bit in the XTCC.

3.3.1.9 Closing a Connection

A transport connection is closed in response to a call to the Xlib CLOSE DISPLAY routine, or the server closing the connection. The XTCC data structure XTCC$V_DYING field is set when the connection is to be closed.

The DECW$XPORT_CLOSE routine terminates a connection and releases the resources that are associated with the connection. The server and Xlib are expected to return any XTCBs acquired through DECW$XPORT_READ and DECW$XPORT_GET_OUTPUT_BUFFER before calling DECW$XPORT_CLOSE. After calling DECW$XPORT_CLOSE, the structures used by the connection (XTCC, XTCQ, IXTCC, XTPB, and XTCBs) must not be referenced.

The DECW$XPORT_CLOSE routine calls the transport-specific connection close routine to actually break the network link.

3.3.2 Transport-Specific Functions

The transport-specific code performs the following functions:

- Initializes and returns the address of the XTFT
- Initializes (attaches) a specific transport
- Connects a client to a server by means of the chosen transport
- Writes data from XTCBs to the transport
- Reads data into XTCBs from the transport
- Closes a connection and releases connection structures
- Initiates image rundown processing for the connection

Subsequent sections describe the transport-specific functions.

3.3.2.1 Initializing the Transport

The XTFT data structure contains the addresses of the transport-specific routines. The common transport must therefore always be able to find the transport-specific XTFT structures. To make sure that the common transport can find the XTFT structures, every transport-specific image must provide a routine, DECW$TRANSPORT_INIT, as the initialization routine of that image.

A transfer vector to the DECW$TRANSPORT_INIT routine must be provided in the first image section of the transport-specific image. See Section 8.3.19 for an example of this transfer vector.

The DECW$XPORT_ATTACH_TRANSPORT routine calls the DECW$TRANSPORT_INIT routine to initialize and return the XTFT. Once the XTFT is initialized, DECW$XPORT_ATTACH_TRANSPORT calls the XTFT$A_ATTACH_TRANSPORT routine found in this data structure to complete the transport-specific initialization.

3.3.3 Attaching the Specific Transport

As described in Section 3.3.2.1, the DECW$XPORT_ATTACH_TRANSPORT routine calls the XTFT$A_ATTACH_TRANSPORT routine to complete the transport-specific initialization.

XTFT$A_ATTACH_TRANSPORT functions differently depending on whether the server or Xlib called it. If Xlib called it, XTFT$A_ATTACH_TRANSPORT performs relatively little transport-specific initialization.

If the server called it, XTFT$A_ATTACH_TRANSPORT performs additional transport-specific initialization. For example, for DECnet, XTFT$A_ATTACH_TRANSPORT might function as follows:

- Create a mailbox
- Assign a channel to the network device and associate the mailbox with this channel
- Associate an object name (for example, X$X0) to which clients may refer with the network channel
- Issue a $QIO read to the mailbox to receive notification of connection attempts by clients

The TRANSPORT_READ_QUEUE and TRANSPORT_READ_AST routines in the example transport are called to initiate a read on the transport channel. TRANSPORT_READ_QUEUE is called by XTFT$A_ATTACH_TRANSPORT to perform the first $QIO read on the newly attached connection. The XTFT$A_ATTACH_TRANSPORT routine assigns a network channel for the transport and then calls TRANSPORT_READ_QUEUE to listen on the channel for a connection attempt from a client.

TRANSPORT_READ_AST is a sample read-completion AST routine for the transport's network channel.

3.3.3.1 Opening a Connection

The XTFT$A_OPEN routine tries to connect a client to a server.

The transport-common DECW$XPORT_OPEN routine attempts to locate a transport with a name matching the one passed in the **xportnam** argument (for example, "DECNET"). If a matching transport is found, the XTFT$A_OPEN routine is called with the server number and workstation node name arguments, and an IXTCC and XTPB that have been partially initialized. (Xlib calls DECW$XPORT_OPEN with an item list that gives its desired defaults.)

XTFT$A_OPEN is responsible for the allocation and initialization of the XTCC, XTCQ, and all necessary XTCBs, and starting an initial read if needed. Parameters that would affect these operations are found in the XTPB attached to the IXTCC by means of the IXTCC$A_TPB field.

3.3.3.2 Writing XTCBs to a Transport

There are three transport-specific routines that are called through the XTFT data structure for writing to the transport:

- XTFT$A_EXECUTE_WRITE
- XTFT$A_WRITE
- XTFT$A_WRITE_USER

Many transport-specific images will also need a write-completion routine. A sample write-completion AST routine, WRITE_AST, is shown in Chapter 8.

The transport-specific write routines are described in Table 3–16.

Table 3–16 Transport-Specific Write Routines

Routine	Description
XTFT$A_EXECUTE_WRITE	Writes the contents of an XTCB to a connection. XTFT$A_EXECUTE_WRITE is called when the common transport inserts an XTCB on an empty output work queue. XTFT$A_EXECUTE_WRITE must decide whether to call DECW$$XPORT_WRITE so that an I/O operation can be started in executive mode.
XTFT$A_WRITE	Attempts to write an XTCB to the connection associated with the XTCC. XTFT$A_WRITE writes the contents of XTCBs across the wire. If there is nothing to write, that is, the XTCBs are empty, XTFT$A_WRITE inserts the XTCBs on the appropriate (small or large) output free queue. This is a method of populating the free queues. If the write operation fails, XTFT$A_WRITE puts the XTCB back at the head of the output work queue and sets the connection status to dying.
XTFT$A_WRITE_USER	Attempts to write a buffer in the user's address space to a transport connection. XTFT$A_WRITE_USER can use the common routine DECW$XPORT_COPY_AND_WRITE to copy the user's buffer into XTCBs and queue them for writing, or wait for the output work queue to empty and issue $QIOs directly from the user's buffer.

3.3.3.3 Reading XTCBs from a Transport

There are two transport-specific routines that are called through the XTFT data structure for reading from the transport: XTFT$A_EXECUTE_FREE and XTFT$A_FREE_INPUT_BUFFER.

Many transport-specific images will also need a read-completion routine. A sample read-completion AST routine, FREE_INPUT_AST, is shown in Chapter 8.

The transport-specific read routines are described in Table 3–17.

Table 3–17 Transport-Specific Read Routines

Routine	Description
XTFT$A_EXECUTE_FREE	Returns an XTCB to a local connection. DECW$XPORT_EXECUTE_FREE calls XTFT$A_EXECUTE_FREE to remove the buffer just placed on the input free queue. In the case of DECnet or TCP/IP, the buffer is then used to store the result of the next $QIO read operation for the connection.
XTFT$A_FREE_INPUT_BUFFER	In the case of DECnet or TCP/IP, XTFT$A_FREE_INPUT_BUFFER does a $QIO read operation for a connection into the provided buffer. If there is nothing to read for the connection, XTFT$A_FREE_INPUT_BUFFER inserts the XTCB on the free queue and sets the connection state to dying.

3.3.3.4 Closing a Connection

The connection close routines close a connection and release the structures associated with the connection. The common transport layer begins deallocation of all connection resources including, but not limited to, channels, XTCC, XTCBs, XTPB, and transport-private data. After this is done, the transport user must not refer to any structures associated with the connection.

There are two transport-specific routines that work together to close a connection: XTFT$A_CLOSE and XTFT$A_RUNDOWN. XTFT$A_CLOSE uses an additional routine, CLOSE_AND_DEALLOCATE_AST, to clean up after aborted I/O operations. A sample CLOSE_AND_DEALLOCATE_AST routine is shown in Chapter 8.

The transport-specific connection close routines are described in Table 3–18.

Table 3–18 Transport-Specific Connection Close Routines

Routine	Description
XTFT$A_CLOSE	Marks the connection as dying and cancels and deassigns the channel to the connection. XTFT$A_CLOSE declares an AST to the CLOSE_AND_DEALLOCATE_AST routine that is executed after any completion ASTs. This performs the final cleanup operations such as structure invalidation and deallocation.
XTFT$A_RUNDOWN	Invoked by the common transport when the current image exits. Each specific transport must release any resources necessary for a clean exit.

3.3.3.5 The Transport-Specific Callback

When a specific transport receives a connection request, it completely sets up the new connection and then calls the server connection-request routine, identified by the XTDB$A_CONNECT_REQUEST field, to see if the server accepts the connection. The sample transport shown in Chapter 8 uses the TRANSPORT_READ_AST, TRANSPORT_READ_QUEUE, and TRANSPORT_OPEN_CALLBACK routines to accomplish this task.

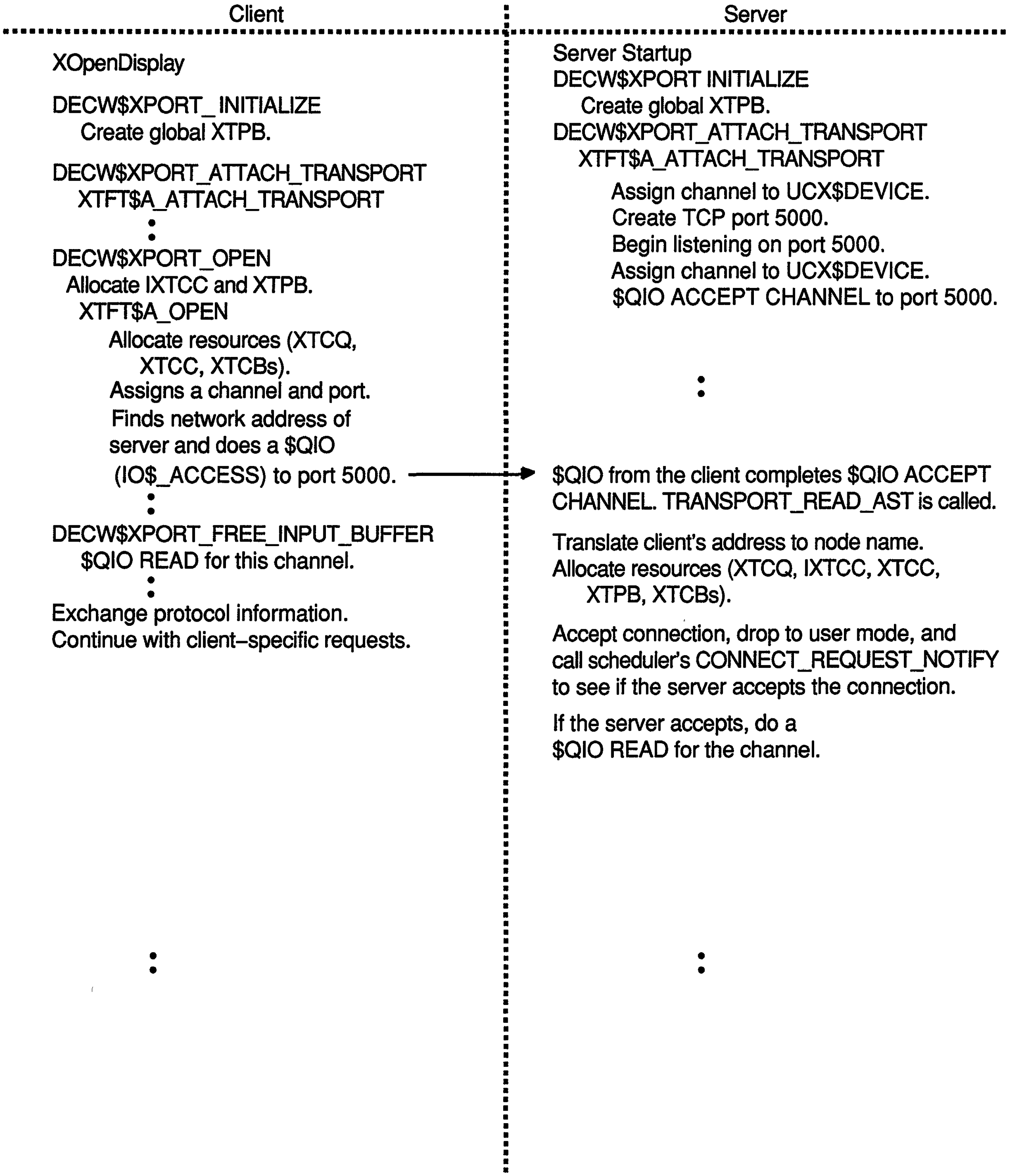
4 Transport Walk-Through

This chapter describes a walk-through of transport layer activities, including transport initialization, for both the server and Xlib. The walk-through gives an overview of the transport layer activities; it does not describe every step of the process. The walk-through is based on the sample TCP/IP transport.

Note: **The boxed numbers in the illustrations correspond to the buffers, or XTCBs, that are being queued.**

The convention for TCP/IP is that server number 0 listens on port 6000. Port 5000 is used in this example to prevent collision with a "real" TCP/IP transport.

Transport Walk-Through



Client

XOpenDisplay

DECW$XPORT_ INITIALIZE
Create global XTPB.

DECW$XPORT_ATTACH_TRANSPORT
XTFT$A_ATTACH_TRANSPORT

DECW$XPORT_OPEN
Allocate IXTCC and XTPB.
XTFT$A_OPEN
Allocate resources (XTCQ, XTCC, XTCBs).
Assigns a channel and port.
Finds network address of server and does a $QIO (IO$_ACCESS) to port 5000. →

DECW$XPORT_FREE_INPUT_BUFFER
$QIO READ for this channel.

Exchange protocol information.
Continue with client–specific requests.

Server

Server Startup
DECW$XPORT INITIALIZE
Create global XTPB.
DECW$XPORT_ATTACH_TRANSPORT
XTFT$A_ATTACH_TRANSPORT
Assign channel to UCX$DEVICE.
Create TCP port 5000.
Begin listening on port 5000.
Assign channel to UCX$DEVICE.
$QIO ACCEPT CHANNEL to port 5000.

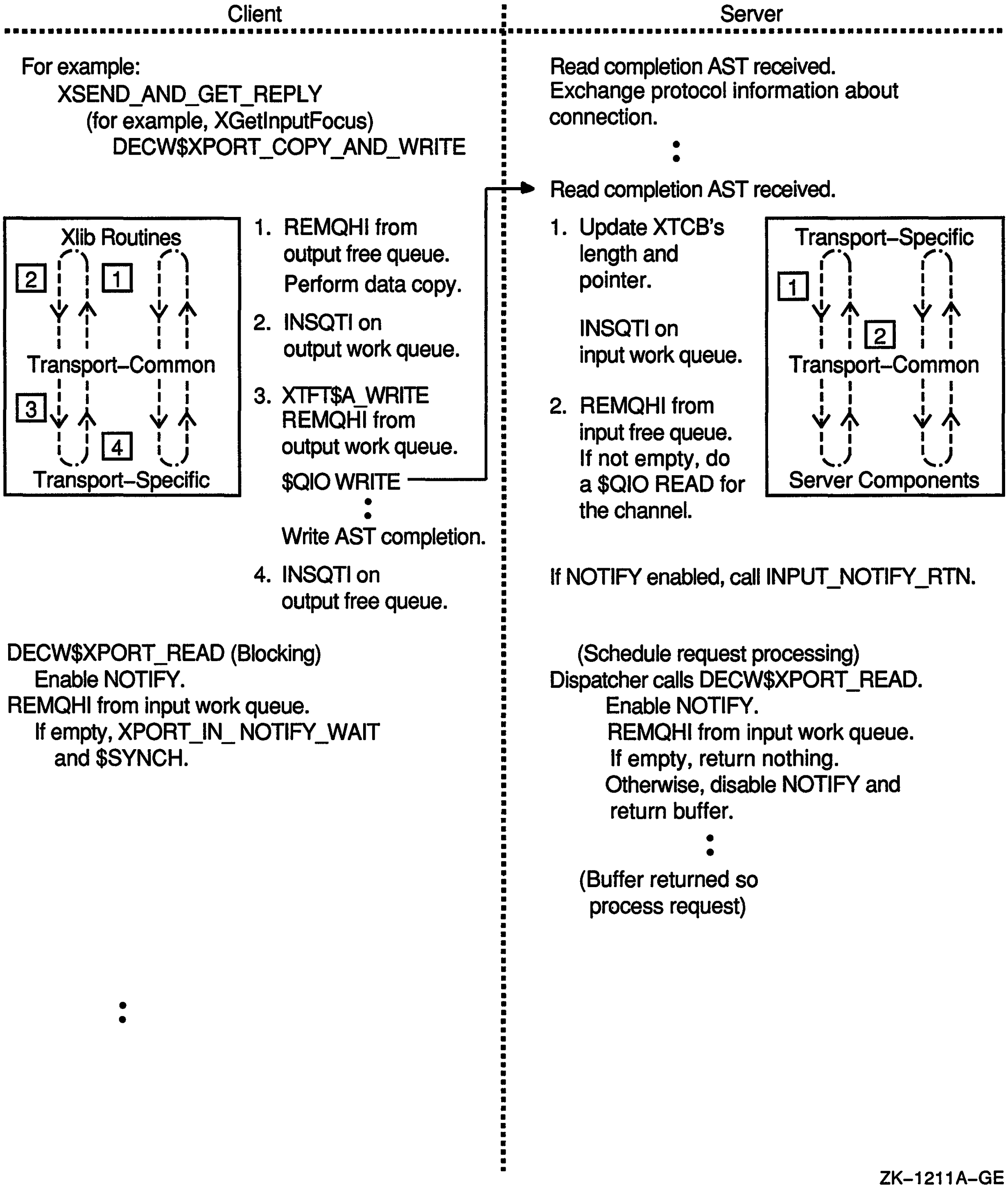
$QIO from the client completes $QIO ACCEPT CHANNEL. TRANSPORT_READ_AST is called.

Translate client's address to node name.
Allocate resources (XTCQ, IXTCC, XTCC, XTPB, XTCBs).

Accept connection, drop to user mode, and call scheduler's CONNECT_REQUEST_NOTIFY to see if the server accepts the connection.

If the server accepts, do a $QIO READ for the channel.

ZK–1210A–GE



Client

For example:
XSEND_AND_GET_REPLY
(for example, XGetInputFocus)
DECW$XPORT_COPY_AND_WRITE

1. REMQHI from output free queue. Perform data copy.
2. INSQTI on output work queue.
3. XTFT$A_WRITE REMQHI from output work queue. $QIO WRITE

Write AST completion.

4. INSQTI on output free queue.

DECW$XPORT_READ (Blocking)
Enable NOTIFY.
REMQHI from input work queue.
If empty, XPORT_IN_ NOTIFY_WAIT
and $SYNCH.

Server

Read completion AST received.
Exchange protocol information about connection.

Read completion AST received.

1. Update XTCB's length and pointer.

   INSQTI on input work queue.

2. REMQHI from input free queue. If not empty, do a $QIO READ for the channel.

If NOTIFY enabled, call INPUT_NOTIFY_RTN.

(Schedule request processing)
Dispatcher calls DECW$XPORT_READ.
Enable NOTIFY.
REMQHI from input work queue.
If empty, return nothing.
Otherwise, disable NOTIFY and
return buffer.

(Buffer returned so
process request)

ZK–1211A–GE

Client

:

(Enter read completion AST)
Update XTCB's length and pointer.
INSQTI on input work queue.
If NOTIFY enabled, send notification.

REMQHI on input free queue.
If empty and input enabled, $QIO
READ for data.

Return

$SYNCH returns from wait-in-LEF state.
DECW$XPORT_READ
REMQHI from input work queue.

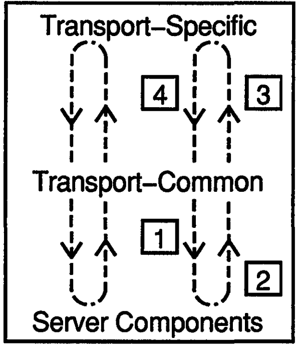
Return buffer.
(Copy reply into user space)

:

XCloseDisplay ()
DECW$XPORT_CLOSE
XTFT$A_CLOSE
$DASSGN CHANNEL →
Return

Server

Generate reply.
DECW$XPORT_WRITE
1 REMQHI from output free queue.
Perform data copy.
Calls WRITE_BUFFER.
2 INSQTI buffer on output work queue.
Calls XTFT$A_EXECUTE WRITE.
3 REMQHI from output work queue.
$QIO WRITE. ←
4 WRITE_COMPLETION INSQTIs on
output free queue.



Dispatcher processes request. When request stream exhausted or request is incomplete, calls DECW$XPORT_FREE_INPUT_BUFFER.
XTFT$A_FREE_INPUT_BUFFER
INSQTI on input free queue.
If it was empty and input was
enabled, REMQHI from input free queue.
$QIO READ into buffer.
Return

:

(Enter read completion AST)
$QIO READ failed.
Mark connection as dying.
Call CONNECT_ABORT_NOTIFY().
Return

(Schedule connection rundown)
DECW$XPORT_CLOSE
XTFT$A_CLOSE
$DASSGN CHANNEL
Release resources.
Return

ZK-1212A-GE

5 Transport-Common Routines

This chapter describes the transport-common routines that are called by specific transports. If you write your own transport-specific layer, use these routines to allocate the communication queues, allocate and deallocate protected memory for structures, initiate read and write operations, and so forth.

Transport-common routines that are called only by other transport-common routines or by Xlib and the server are not described in this chapter.

Modifications to the transport-common routines are not recommended or supported.

The transport-common routines are listed in Table 5–1.

Table 5–1 Transport-Common Routines

Routine Name	Function
DECW$XPORT_ACCEPT_FAILED	Reports that the transport-specific routines could not accept a network link request.
DECW$XPORT_ALLOC_INIT_QUEUES	Allocates storage for an XTCC, XTCQ, and all of the XTCBs for a connection. Places all of the XTCBs on the appropriate free queues.
DECW$XPORT_ALLOC_PMEM	Allocation routine for protected structures.
DECW$XPORT_ATTACHED	Reports that a transport is attached.
DECW$XPORT_ATTACH_LOST	Reports that a network has shut down.
DECW$XPORT_CLOSE	Terminates a connection and releases its associated resources.
DECW$XPORT_COPY_AND_WRITE	Copies data into XTCBs and optionally starts a write operation.
DECW$XPORT_DEALLOC_PMEM	Deallocation routine for protected structures allocated with DECW$XPORT_ALLOC_PMEM.
DECW$XPORT_DEALLOC_QUEUES	Deallocates a block of storage previously allocated by DECW$XPORT_ALLOC_INIT_QUEUES.
DECW$$XPORT_FREE_INPUT	Initiates a read operation for a connection.
DECW$XPORT_IN_NOTIFY_USER	Notifies Xlib or the server that data on the input work queue is available to be read.
DECW$XPORT_REATTACH_FAILED	Reports that the transport layer cannot continue attempting to reattach.

(continued on next page)

Table 5–1 (Cont.) Transport-Common Routines

Routine Name	Function
DECW$XPORT_REFUSED_BY_ SERVER	Reports that the server did not accept a connection.
DECW$XPORT_UNEXPECTED_ MESSAG	Reports that an unexpected message was received from the underlying transport.
DECW$XPORT_UNKNOWN_LINK	Reports a message from an unknown connection.
DECW$XPORT_VALIDATE_STRUCT	Returns the address of the user write-protected IXTCC structure.
DECW$XPORT_VALIDATE_STRUCT_ JSB	JSB routine that returns the address of the user write-protected IXTCC structure.
DECW$XPORT_VALIDATE_XTCB	Validates that an XTCB is contained within the allocated storage for the connection and that it is correctly formed.
DECW$XPORT_VALIDATE_XTCB_JSB	JSB routine that validates that an XTCB is contained within the allocated storage for the connection and that it is correctly formed.
DECW$$XPORT_WRITE	Initiates a writing operation on the connection.

DECW$XPORT_ACCEPT_FAILED

Reports that the transport could not accept a network link request.

FORMAT

DECW$XPORT_ACCEPT_FAILED *length, address, status*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value in R0. Condition values returned by this routine are, listed under Condition Values Returned.

ARGUMENTS

length

VMS usage:	**longword**
type:	**longword (unsigned)**
access:	**read**
mechanism:	**value**

The length of the connection-failed message string.

address

VMS usage:	**longword**
type:	**longword (unsigned)**
access:	**read**
mechanism:	**value**

The address of the connection-failed message string, which is usually the node name of the failed connection.

status

VMS usage:	**longword**
type:	**longword (unsigned)**
access:	**read**
mechanism:	**value**

The condition value of the failed connection.

DESCRIPTION

DECW$XPORT_ACCEPT_FAILED builds a message vector that describes the nonacceptance of the connection request. The first element in the message vector is DECW$_ACCEPT_FAILED; the second element is the **status** argument.

The $PUTMSG system service is called in user mode regardless of the caller's mode:

- If DECW$XPORT_ACCEPT_FAILED was called in user mode, it calls $PUTMSG to write the error message.
- If DECW$XPORT_ACCEPT_FAILED was not called in user mode, it declares an AST to do the $PUTMSG in user mode.

DECW$XPORT_ACCEPT_FAILED is called by the TRANSPORT_READ_AST routine in Example 8–15.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
SS$_INSFMEM	There is insufficient memory to perform the operation.

Any condition value returned by $PUTMSG.

Any condition value returned by $DCLAST.

DECW$XPORT_ALLOC_INIT_QUEUES

Allocates storage for an XTCC, XTCQ, and all of the XTCBs for a connection. Places all of the XTCBs on the appropriate free queues.

FORMAT

DECW$XPORT_ALLOC_INIT_QUEUES
itcc, xtcc_length, srp_data_length, lrp_data_length, e_srp_count, e_lrp_count, r_srp_count, r_lrp_count, extra_context_length, extra_context_address

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value in R0. Condition values returned by this routine are listed under Condition Values Returned.

ARGUMENTS

itcc

VMS usage:	**record**
type:	**ixtcc**
access:	**modify**
mechanism:	**reference**

The IXTCC of the connection for which you want to allocate and initialize the queues. The IXTCC$A_TPB field must already be initialized.

xtcc_length

VMS usage:	**longword**
type:	**longword**
access:	**read**
mechanism:	**value**

The length, in bytes, of the XTCC to allocate. May be longer than a standard XTCC if the specific transport has appended additional fields. Must be at least XTCC$C_LENGTH.

srp_data_length

VMS usage:	**longword**
type:	**longword**
access:	**read**
mechanism:	**value**

The length, in bytes, of the data portion of a small XTCB. No modification of the XTCB by specific transports is allowed.

lrp_data_length

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**
The length, in bytes, of the data portion of a large XTCB. No modification of the XTCB by specific transports is allowed.

e_srp_count

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**
The number of event small XTCBs to allocate. May be 0 or greater.

e_lrp_count

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**
The number of event large XTCBs to allocate. May be 0 or greater.

r_srp_count

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**
The number of request small XTCBs to allocate. May be 0 or greater.

r_lrp_count

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**
The number of request large XTCBs to allocate. May be 0 or greater.

extra_context_length

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**
The length, in bytes, of the transport-specific space to be allocated. May be 0 or greater.

extra_context_address

VMS usage: **address**
type: **longword**
access: **write**
mechanism: **reference**
The location to receive the address of the extra transport-specific space.

DESCRIPTION

DECW$XPORT_ALLOC_INIT_QUEUES allocates a block of storage for an XTCC, XTCQ, and all of the XTCBs for a connection and places all of the XTCBs on the appropriate free queues. DECW$XPORT_ALLOC_INIT_QUEUES must allocate at least an XTCC; the other structures are optional.

If no XTCBs are requested, the XTCQ is not allocated. Otherwise, the XTCQ is allocated and initialized. The IXTCC is initialized with the addresses of the XTCC and the XTCQ, the queue headers, and the queue flags. The IXTCC$L_ICI and IXTCC$A_USER_REGION fields are also initialized.

When DECW$XPORT_ALLOC_INIT_QUEUES completes, the status of the data structures is as follows:

Data Structure	Status
IXTCC	The following fields are initialized: • IXTCC$L_ICI • IXTCC$A_TCC • IXTCC$A_TCQ • IXTCC$A_USER_REGION • IXTCC$A_BUFFER_REGION • IXTCC$A_IW_QUEUE • IXTCC$A_IFS_QUEUE • IXTCC$A_IFL_QUEUE • IXTCC$A_OW_QUEUE • IXTCC$A_OFS_QUEUE • IXTCC$A_OFL_QUEUE • IXTCC$L_IWQ_FLAG • IXTCC$L_IFSQ_FLAG • IXTCC$L_IFLQ_FLAG • IXTCC$L_OWQ_FLAG • IXTCC$L_OFSQ_FLAG • IXTCC$L_OFLQ_FLAG

Data Structure	Status
XTCC	The following fields are initialized: • XTCC$W_SIZE • XTCC$B_TYPE • XTCC$B_SUBTYPE • XTCC$A_TPB • XTCC$A_TCQ • XTCC$L_ICI • XTCC$A_IW_QUEUE • XTCC$A_IFS_QUEUE • XTCC$A_IFL_QUEUE • XTCC$A_OW_QUEUE • XTCC$A_OFS_QUEUE • XTCC$A_OFL_QUEUE • XTCC$A_TCQ_FLAGS • XTCC$L_IWQ_FLAG • XTCC$L_IFSQ_FLAG • XTCC$L_IFLQ_FLAG • XTCC$L_OWQ_FLAG • XTCC$L_OFSQ_FLAG • XTCC$L_OFLQ_FLAG
XTCBs	Completely initialized
XTCQ	Completely initialized

DECW$XPORT_ALLOC_INIT_QUEUES may be called only from executive mode.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
DECW$_NOT_INITIALIZED	The common transport is not initialized.
DECW$_BADQUEUE	A queue was corrupted during initialization.

DECW$XPORT_ALLOC_PMEM

Allocation routine for protected structures.

FORMAT

*status_return=***DECW$XPORT_ALLOC_PMEM** *size, subtype*

RETURNS

VMS usage: **address**
type: **longword**
access: **write**
mechanism: **value**

Returns 0 on failure, and a nonzero address of allocated storage if successful.

ARGUMENTS

size

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**
The size of the memory block to allocate, in bytes.

subtype

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**
User-defined subtype field used to initialize the final byte of the third longword.

DESCRIPTION

DECW$XPORT_ALLOC_PMEM is an allocation routine for protected structures. The sample transport calls DECW$XPORT_ALLOC_PMEM to allocate IXTCCs and XTPBs, structures that must not be modified by user-mode code. DECW$XPORT_ALLOC_PMEM allocates a block of storage of the size that you specify. The block is assumed to begin with a 3-longword structure prefix; the length, type, and subtype fields in the third longword are initialized to appropriate values.

DECW$XPORT_ALLOC_PMEM is called only from executive mode. The allocated memory is protected as user-read/executive-write (UREW).

DECW$XPORT_ATTACHED

Reports that a transport is attached.

FORMAT

DECW$XPORT_ATTACHED *tdb*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write**
mechanism: **value**

Returns a longword condition value in R0. Condition values returned by this routine are listed under Condition Values Returned.

ARGUMENT

tdb
VMS usage: **record**
type: **xtdb**
access: **modify**
mechanism: **reference**
The XTDB of the transport that is attached.

DESCRIPTION

DECW$XPORT_ATTACHED builds a message vector that describes the attached transport. The only element of the vector is DECW$_ATTACHED.

The $PUTMSG system service is called in user mode regardless of the caller's mode:

- If DECW$XPORT_ATTACHED was called in user mode, it calls $PUTMSG to write the message.
- If DECW$XPORT_ATTACHED was not called in user mode, it declares a user-mode AST to do the $PUTMSG.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
SS$_INSFMEM	There is insufficient memory to perform the operation.

Any condition value returned by $PUTMSG.

Any condition value returned by $DCLAST.

DECW$XPORT_ATTACH_LOST

Reports that a network has shut down.

FORMAT

DECW$XPORT_ATTACH_LOST *tdb, status*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write**
mechanism: **value**

Returns a longword condition value in R0. Condition values returned by this routine are listed under Condition Values Returned.

ARGUMENTS

tdb

VMS usage: **record**
type: **xtdb**
access: **modify**
mechanism: **reference**
The XTDB of the transport that shut down.

status

VMS usage: **longword**
type: **longword (unsigned)**
access: **read**
mechanism: **value**
The condition value of the transport that is shutting down.

DESCRIPTION

DECW$XPORT_ATTACH_LOST builds a message vector that describes the network shutdown. The first element in the message vector is DECW$_ATTACH_LOST; the second element is the **status** argument.

The $PUTMSG system service is called in user mode regardless of the caller's mode:

- If DECW$XPORT_ATTACH_LOST was called in user mode, it calls $PUTMSG to write the message.
- If DECW$XPORT_ATTACH_LOST was not called in user mode, it declares a user-mode AST to do the $PUTMSG.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
SS$_INSFMEM	There is insufficient memory to perform the operation.

Any condition value returned by $PUTMSG.

Any condition value returned by $DCLAST.

DECW$XPORT_CLOSE

Terminates a connection and releases its associated resources.

FORMAT

DECW$XPORT_CLOSE *tcc*

ARGUMENTS

tcc
VMS usage: **record**
type: **xtcc**
access: **modify**
mechanism: **reference**
The XTCC of the connection to close.

DESCRIPTION

DECW$XPORT_CLOSE terminates a connection and releases its associated resources. The transport user is expected to return any XTCBs acquired through DECW$XPORT_READ and DECW$XPORT_GET_OUTPUT_BUFFER before calling DECW$XPORT_CLOSE. After calling DECW$XPORT_CLOSE, the structures used by the connection (XTCC, XTCQ, and XTCBs) must not be referenced.

The DECW$XPORT_CLOSE routine calls the transport-specific connection close routine, XTFT$A_CLOSE, to actually break the network link.

DECW$XPORT_COPY_AND_WRITE

Copies data into XTCBs and optionally starts a write operation.

FORMAT

DECW$XPORT_COPY_AND_WRITE *tcc, mode, buffer, buflen, retbuflen, padbytes*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write**
mechanism: **value**

Returns a longword condition value in R0. Condition values returned by this routine are listed under Condition Values Returned.

ARGUMENTS

tcc

VMS usage: **record**
type: **xtcc**
access: **modify**
mechanism: **reference**
The XTCC of the connection from which you want to write.

mode

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**
Modifying flags for the write operation. The valid fields are:

Constant	Description
DECW$M_MODE_NOBLOCK	Nonblocking write. When the DECW$M_MODE_NOBLOCK flag is set, any attempt to get a transport buffer when none is available causes the the call to return the status DECW$_BUFNOTAVL and perform only a partial write operation. When this bit is clear and a nonzero timeout value was specified in DECW$XPORT_ATTACH_TRANSPORT, getting a buffer can time out, causing this routine to return with the DECW$_BUFFERTIMEOUT status.

Constant	Description
DECW$M_MODE_ NOWRTBLOCK	If the specific transport was blocked for the XTCB write operation, the last XTCB is placed on the work queue and DECW$XPORT_COPY_AND_ WRITE returns with a DECW$_BLOCKED status.

buffer

VMS usage: **char string**
type: **char string**
access: **read**
mechanism: **reference**
A buffer in the user's address space that contains data to write to the connection.

buflen

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**
The length of the data in the buffer.

retbuflen

VMS usage: **longword**
type: **longword**
access: **write**
mechanism: **reference**
Address of longword to receive the amount of data that was actually written to the connection. If DECW$M_MODE_NOBLOCK was *not* set in the **mode** argument, and there is no timeout, **retbuflen** is always the amount requested.

padbytes

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**
Number of pad bytes (zeros) to append to the copy.

DESCRIPTION

DECW$XPORT_COPY_AND_WRITE performs a buffered write operation and returns the size of the data actually copied in the **retbuflen** argument. Data from the user's buffer is always copied to transport buffers prior to processing by the transport-specific write function.

The sample transport calls DECW$XPORT_COPY_AND_WRITE from the XTFT$A_WRITE_USER routine.

DECW$XPORT_COPY_AND_WRITE must be called in user mode.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
SS$_ACCVIO	XTCC is not user-readable.
SS$_BADPARAM	XTCC argument is not an XTCC.
DECW$_CNXABORT	Connection is in abort condition.
DECW$_RECIO_OPE	Recursive I/O operation.
DECW$_BLOCKED	The transport blocked the write operation.
DECW$_BUFFERTIMEOUT	Timed out while waiting for buffer.
DECW$_NOT_INITIALIZED	The common transport is not initialized.
DECW$_INV_STRUCT_ID	The XTCC$L_ICI field is not valid.

May also return any other status set in the XTCC$L_ERR_STATUS field if the connection is aborting.

DECW$XPORT_DEALLOC_PMEM

Deallocation routine for protected structures allocated with DECW$XPORT_ALLOC_PMEM.

FORMAT

DECW$XPORT_DEALLOC_PMEM *buffer*

ARGUMENTS

buffer

VMS usage:	**address**
type:	**longword**
access:	**read**
mechanism:	**value**

The address of the buffer to deallocate.

DESCRIPTION

DECW$XPORT_DEALLOC_PMEM deallocates a structure that was previously allocated by DECW$XPORT_ALLOC_PMEM. Any errors are signaled. DECW$XPORT_DEALLOC_PMEM does not return a condition value.

DECW$XPORT_DEALLOC_PMEM must be called in executive mode.

DECW$XPORT_DEALLOC_QUEUES

Deallocates a block of storage previously allocated by DECW$XPORT_ALLOC_INIT_QUEUES.

FORMAT

DECW$XPORT_DEALLOC_QUEUES *itcc*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value in R0. Condition values returned by this routine are listed under Condition Values Returned.

ARGUMENTS

itcc

VMS usage:	**record**
type:	**ixtcc**
access:	**modify**
mechanism:	**reference**

The IXTCC of the connection for which you want to deallocate a block of storage previously allocated for the queues.

DESCRIPTION

DECW$XPORT_DEALLOC_QUEUES deallocates a block of storage that was previously allocated for the queues of a connection by DECW$XPORT_ALLOC_INIT_QUEUES.

DECW$XPORT_DEALLOC_QUEUES must be called in executive mode.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
DECW$_NOT_INITIALIZED	The common transport is not initialized.

DECW$$XPORT_FREE_INPUT

Initiates a read operation on the connection.

FORMAT

DECW$$XPORT_FREE_INPUT *tcc, tcb*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value in R0. Condition values returned by this routine are listed under Condition Values Returned.

ARGUMENTS

tcc

VMS usage:	**record**
type:	**xtcc**
access:	**modify**
mechanism:	**reference**

The XTCC of the connection from which you want to read.

tcb

VMS usage:	**record**
type:	**xtcb**
access:	**modify**
mechanism:	**reference**

DECW$$XPORT_FREE_INPUT initiates a read operation for the connection into this XTCB.

DESCRIPTION

The DECW$$XPORT_FREE_INPUT system service calls the XTFT$A_FREE_INPUT_BUFFER routine in executive mode to perform an asynchronous read operation for the connection. The read operation is performed asynchronously to avoid waiting for it to complete.

The XTFT$A_FREE_INPUT_BUFFER routine does the actual read operation for the connection into the XTCB specified in the **tcb** argument. If the read operation fails, XTFT$A_FREE_INPUT_BUFFER inserts the XTCB on the free queue and sets the connection state to dying.

DECW$$XPORT_FREE_INPUT is an executive-mode routine.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
DECW$_CNXABORT	Connection is in abort condition.

DECW$XPORT_IN_NOTIFY_USER

Notifies Xlib or the server that data on the input work queue is available to be read.

FORMAT

DECW$XPORT_IN_NOTIFY_USER *tcc*

ARGUMENT

tcc

VMS usage:	**record**
type:	**xtcc**
access:	**modify**
mechanism:	**reference**

The XTCC of the connection for which you want to limit input-notify AST delivery.

DESCRIPTION

DECW$XPORT_IN_NOTIFY_USER clears the XTCC$V_IN_AST_IN_PROG bit in the XTCC to indicate that the AST has been delivered, and calls the input notification procedure, identified by the XTPB$A_I_NOTIFY_RTNADR field.

Transport users may request notification of input data. One method of notifying transport users is to deliver an AST that calls the user's XTPB$A_I_NOTIFY_RTNADR procedure with the XTCC as an argument.

It is possible for a large number of read operations to complete before the first notification AST is delivered, particularly if an Xlib application is executing at user-AST level for a long period of time, or not reading events.

If ASTs were declared without regard for whether they were being delivered, it is possible to exceed the process AST quota. DECW$XPORT_IN_NOTIFY_USER and the XPORT_IN_NOTIFY_SEND macro provide a mechanism for limiting AST use.

The XPORT_IN_NOTIFY_SEND macro tests and sets the XTCC$V_IN_AST_IN_PROG bit. If it was set, there is already a notification AST waiting to be delivered. If the XTCC$V_IN_AST_IN_PROG bit was clear, XPORT_IN_NOTIFY_SEND declares a user-mode AST to call DECW$XPORT_IN_NOTIFY_USER.

DECW$XPORT_IN_NOTIFY_USER clears the XTCC$V_IN_AST_IN_PROG bit and then calls the user's XTPB$A_I_NOTIFY_RTNADR procedure.

DECW$XPORT_IN_NOTIFY_USER is called in user mode.

DECW$XPORT_REATTACH_FAILED

Reports that the transport layer cannot continue attempting to reattach a transport.

FORMAT

DECW$XPORT_REATTACH_FAILED *tdb, status*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value in R0. Condition values returned by this routine are listed under Condition Values Returned.

ARGUMENTS

tdb

VMS usage:	**record**
type:	**xtdb**
access:	**modify**
mechanism:	**reference**

The XTDB of the transport to which the transport layer cannot continue attempting to reattach.

status

VMS usage:	**longword**
type:	**longword (unsigned)**
access:	**read**
mechanism:	**value**

The condition value of the failed reattach attempt.

DESCRIPTION

DECW$XPORT_REATTACH_FAILED builds a message vector that describes the failed reattach attempt. The first element in the message vector is DECW$_REATTACH_FAILED; the second element is the **status** argument.

When a transport shuts down, a specific transport can attempt to reattach that transport. If the specific transport is then unable to complete the reattach attempt, DECW$XPORT_REATTACH_FAILED is called.

The $PUTMSG system service is called in user mode regardless of the caller's mode:

- If DECW$XPORT_REATTACH_FAILED was called in user mode, it calls $PUTMSG to write the error message.
- If DECW$XPORT_REATTACH_FAILED was not called in user mode, it declares a user-mode AST to do the $PUTMSG.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
SS$_INSFMEM	There is insufficient memory to perform the operation.

Any condition value returned by $PUTMSG.

Any condition value returned by $DCLAST.

DECW$XPORT_REFUSED_BY_SERVER

Reports that the server rejected a connection request.

FORMAT

DECW$XPORT_REFUSED_BY_SERVER *status*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write**
mechanism: **value**

Returns a longword condition value in R0. Condition values returned by this routine are listed under Condition Values Returned.

ARGUMENT

status

VMS usage: **longword**
type: **longword (unsigned)**
access: **read**
mechanism: **value**
The condition value of the server's rejection.

DESCRIPTION

DECW$XPORT_REFUSED_BY_SERVER builds a message vector that describes the server's rejection of a connection request. The first element in the message vector is DECW$_REFUSED_BY_SERVER; the second element is the **status** argument.

The $PUTMSG system service is called in user mode regardless of the caller's mode:

- If DECW$XPORT_REFUSED_BY_SERVER was called in user mode, it calls $PUTMSG to write the error message.
- If DECW$XPORT_REFUSED_BY_SERVER was not called in user mode, it declares a user-mode AST to do the $PUTMSG.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
SS$_INSFMEM	There is insufficient memory to perform the operation.

Any condition value returned by $PUTMSG.

Any condition value returned by $DCLAST.

DECW$XPORT_UNEXPECTED_MESSAG

Reports that an unexpected message was received from the underlying transport.

FORMAT

DECW$XPORT_UNEXPECTED_MESSAG *type*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value in R0. Condition values returned by this routine are listed under Condition Values Returned.

ARGUMENT

type

VMS usage:	**longword**
type:	**longword (unsigned)**
access:	**read**
mechanism:	**value**

The type of the unexpected message.

DESCRIPTION

DECW$XPORT_UNEXPECTED_MESSAG builds a message vector that describes an unexpected message. The only element in the message vector is DECW$_UNEXPECTED_MESSAGE.

The $PUTMSG system service is called in user mode regardless of the caller's mode:

- If DECW$XPORT_UNEXPECTED_MESSAG was called in user mode, it calls $PUTMSG to write the error message.
- If DECW$XPORT_UNEXPECTED_MESSAG was not called in user mode, it declares an AST to do the $PUTMSG.

DECW$XPORT_UNEXPECTED_MESSAG is currently used only by the DECnet transport.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
SS$_INSFMEM	There is insufficient memory to perform the operation.

Any condition value returned by $PUTMSG.

Any condition value returned by $DCLAST.

DECW$XPORT_UNKNOWN_LINK

Reports a message about an unknown connection from the underlying transport.

FORMAT

DECW$XPORT_UNKNOWN_LINK *unit*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value in R0. Condition values returned by this routine are listed under Condition Values Returned.

ARGUMENT

unit

VMS usage:	**longword**
type:	**longword (unsigned)**
access:	**read**
mechanism:	**value**

The unknown link's unit number.

DESCRIPTION

DECW$XPORT_UNKNOWN_LINK builds a message vector that describes an unknown connection. The only element of the vector is DECW$_UNKNOWN_LINK.

The $PUTMSG system service is called in user mode regardless of the caller's mode:

- If DECW$XPORT_UNKNOWN_LINK was called in user mode, it calls $PUTMSG to write the error message.
- If DECW$XPORT_UNKNOWN_LINK was not called in user mode, it declares a user-mode AST to do the $PUTMSG.

DECW$XPORT_UNKNOWN_LINK is currently used only by the DECnet transport.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
SS$_INSFMEM	There is insufficient memory to perform the operation.

Any condition value returned by $PUTMSG.

Any condition value returned by $DCLAST.

DECW$XPORT_VALIDATE_STRUCT

Returns the address of the user write-protected IXTCC structure.

FORMAT

DECW$XPORT_VALIDATE_STRUCT *id, struct*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value in R0. Condition values returned by this routine are listed under Condition Values Returned.

ARGUMENTS

id

VMS usage:	**longword**
type:	**longword (unsigned)**
access:	**read**
mechanism:	**value**

A previously registered IXTCC structure ID (XTCC$L_ICI).

struct

VMS usage:	**address**
type:	**ixtcc**
access:	**write**
mechanism:	**reference**

Returns the address of the user write-protected IXTCC structure.

DESCRIPTION

DECW$XPORT_VALIDATE_STRUCT checks the user write-protected IXTCC structure ID and, if valid, returns its corresponding address. DECW$XPORT_VALIDATE_STRUCT invokes the DECW$XPORT_VALIDATE_STRUCT_JSB routine for callers using CALLS.

DECW$XPORT_VALIDATE_STRUCT is called in both user and executive modes.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
DECW$_NOT_INITIALIZED	The common transport is not initialized.
DECW$_INV_STRUCT_ID	The structure ID is not valid.

DECW$XPORT_VALIDATE_STRUCT_JSB

JSB routine that returns the address of the user write-protected IXTCC structure.

FORMAT

DECW$XPORT_VALIDATE_STRUCT *id, struct*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write**
mechanism: **value**

Returns a longword condition value in R0. Condition values returned by this routine are listed under Condition Values Returned.

ARGUMENTS

id

VMS usage: **longword**
type: **longword (unsigned)**
access: **read**
mechanism: **value**
A previously registered IXTCC structure ID (XTCC$L_ICI).

struct

VMS usage: **address**
type: **ixtcc**
access: **write**
mechanism: **reference**
Returns the address of the user write-protected IXTCC structure.

DESCRIPTION

DECW$XPORT_VALIDATE_STRUCT_JSB checks the ID of the user write-protected IXTCC structure. If a structure exists with that ID, DECW$XPORT_VALIDATE_STRUCT_JSB returns its corresponding address in the **struct** argument.

The VALIDATE_XTCC macro calls the DECW$XPORT_VALIDATE_STRUCT_JSB routine to validate an XTCC.

DECW$XPORT_VALIDATE_STRUCT_JSB is called in both user and executive modes.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
DECW$_NOT_INITIALIZED	The common transport is not initialized.
DECW$_INV_STRUCT_ID	The structure ID is not valid.

DECW$XPORT_VALIDATE_XTCB

Validates that an XTCB is contained within the allocated storage for the connection and that it is correctly formed.

FORMAT

DECW$XPORT_VALIDATE_XTCB *itcc, tcb*

RETURNS

VMS usage:	**cond_value**
type:	**longword(unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value to R0. Possible condition values are listed under Condition Values Returned.

ARGUMENTS

itcc

VMS usage:	**record**
type:	**ixtcc**
access:	**modify**
mechanism:	**reference**

The IXTCC of the connection for which you want to validate the XTCB.

tcb

VMS usage:	**record**
type:	**xtcb**
access:	**modify**
mechanism:	**reference**

The XTCB that you want to validate.

DESCRIPTION

DECW$XPORT_VALIDATE_XTCB invokes DECW$XPORT_VALIDATE_XTCB_JSB for callers using CALLS.

DECW$XPORT_VALIDATE_XTCB validates that an XTCB is contained within the allocated buffer storage for the connection and that it is correctly formed.

Either DECW$XPORT_VALIDATE_XTCB or DECW$XPORT_VALIDATE_XTCB_JSB must be called before an XTCB is used in executive mode.

DECW$XPORT_VALIDATE_XTCB may be called in both user and executive modes.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
DECW$_NOT_INITIALIZED	The common transport is not initialized.
DECW$_NOT_XTCB	The XTCB is not in the buffer region.
DECW$_ILLFORMED_XTCB	The XTCB header is not valid.
SS$_IVBUFLEN	The XTCB length field is not valid.

DECW$XPORT_VALIDATE_XTCB_JSB

JSB routine that validates that an XTCB is contained within the allocated sto:age for the connection and that it is correctly formed.

FORMAT

DECW$XPORT_VALIDATE_XTCB_JSB *itcc, tcb*

RETURNS

VMS usage: **cond_value**
type: **longword(unsigned)**
access: **write**
mechanism: **value**

Returns a longword condition value to R0. Possible condition values are listed under Condition Values Returned.

ARGUMENTS

itcc

VMS usage: **record**
type: **ixtcc**
access: **modify**
mechanism: **reference**
The IXTCC of the connection for which you want to validate the XTCB.

tcb

VMS usage: **record**
type: **xtcb**
access: **modify**
mechanism: **reference**
The XTCB that you want to validate.

DESCRIPTION

DECW$XPORT_VALIDATE_XTCB_JSB validates that an XTCB is contained within the allocated buffer storage for the connection and that it is correctly formed.

Either DECW$XPORT_VALIDATE_XTCB_JSB or DECW$XPORT_VALIDATE_XTCB must be called before an XTCB is used in executive mode.

DECW$XPORT_VALIDATE_XTCB_JSB may be called in both user and executive modes.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
DECW$_NOT_INITIALIZED	The common transport is not initialized.
DECW$_NOT_XTCB	The XTCB is not in the buffer region.
DECW$_ILLFORMED_XTCB	The XTCB header is not valid.
SS$_IVBUFLEN	The XTCB length field is not valid.

DECW$$XPORT_WRITE

Initiates a write operation on the connection.

FORMAT

DECW$$XPORT_WRITE *tcc, xtcb, mode*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write**
mechanism: **value**

Returns a longword condition value in R0. Condition values returned by this routine are listed under Condition Values Returned.

ARGUMENTS

tcc

VMS usage: **record**
type: **xtcc**
access: **modify**
mechanism: **reference**
The XTCC of the connection from which you want to write.

xtcb

VMS usage: **record**
type: **xtcb**
access: **modify**
mechanism: **reference**
The XTCB you want to write to the connection.

mode

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**
Modifying flags for the write operation. The valid field is:

Constant	Description
DECW$M_MODE_ NOWRTBLOCK	If the specific transport was blocked for the XTCB write operation, DECW$$XPORT_WRITE returns with a DECW$_BLOCKED status.

DESCRIPTION

The system service DECW$$XPORT_WRITE initiates a write operation on the connection associated with an XTCC. DECW$$XPORT_WRITE dispatches a write operation to the transport-specific write function.

DECW$$XPORT_WRITE calls the VALIDATE_XTCC macro to validate the XTCC. If it is valid, DECW$$XPORT_WRITE gets the XTFT from the IXTCC$A_XPORT_TABLE field and calls the transport-specific XTFT$A_WRITE routine in executive mode to actually write the XTCB.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed.
DECW$_CNXABORT	Connection is in abort condition.
DECW$_RECIO_OPE	Recursive I/O operation.
DECW$_BLOCKED	The transport blocked the write operation.

6 Transport-Specific Routines

This chapter describes the transport-specific routines that you must implement if you write your own transport-specific component.

Most of the routines described in this chapter are called by the transport-common code through the XTFT data structure. However, this chapter also describes supporting routines, such as AST completion routines, that you must write if your transport design requires it.

The transport-specific routines are listed in Table 6–1.

Table 6–1 Transport-Specific Routines

Routine	Function
CLOSE_AND_DEALLOCATE_AST	Completes the connection close initiated by XTFT$A_CLOSE. Internal to specific transport.
DECW$TRANSPORT_INIT	Initializes and returns the XTFT data structure.
DETACH_AND_POLL	Detaches from the transport and starts polling for a transport restart. Internal to specific transport.
FREE_INPUT_AST	AST completion routine for transport read operations. Internal to specific transport.
REATTACH_AST	Attempts to reattach the transport when the timer interval has expired. Internal to specific transport.
TRANSPORT_OPEN_CALLBACK	Performs a callback to the client during the connection-open sequence. Internal to specific transport.
TRANSPORT_READ_AST	Read-completion AST routine for the transport's network channel. Internal to specific transport.
TRANSPORT_READ_QUEUE	Initiates an asynchronous connection-accept operation. Internal to specific transport.
WRITE_AST	AST completion routine for transport write operations. Internal to specific transport.
XTFT$A_ATTACH_TRANSPORT	Performs the transport-specific initialization functions.
XTFT$A_CLOSE	Closes a transport connection.
XTFT$A_EXECUTE_FREE	Returns an XTCB to a local connection.
XTFT$A_EXECUTE_WRITE	Writes an XTCB to a transport-specific connection.
XTFT$A_FREE_INPUT_BUFFER	Starts a read operation on a freed input buffer.

(continued on next page)

Table 6–1 (Cont.) Transport-Specific Routines

Routine	Function
XTFT$A_OPEN	Attempts to establish a connection to a server.
XTFT$A_RUNDOWN	Performs the transport-specific rundown functions required during image rundown.
XTFT$A_WRITE	Writes an XTCB buffer from the common transport to a transport-specific connection.
XTFT$A_WRITE_USER	Attempts to write a buffer in the user's address space to a transport-specific connection.

The routine descriptions document the generic functions that the transport-common component expects the transport-specific routines to perform. Your implementation of the routines depends on your underlying transport and may therefore differ in details. However, your implementation of the routines must fullfill the transport-common expectations.

See Chapter 8 for examples of TCP/IP implementations of these routines.

6.1 Condition Values

When a transport-specific routine finishes execution, a numeric status value is returned in R0. This status value is returned as the status value of the transport-common routine. Your implementation of the transport-specific routines can return any valid VMS condition value.

CLOSE_AND_DEALLOCATE_AST

Completes the connection close initiated by XTFT$A_CLOSE. Internal to specific transport.

FORMAT

CLOSE_AND_DEALLOCATE_AST *itcc*

ARGUMENT

itcc

VMS usage: **record**
type: **ixtcc**
access: **modify**
mechanism: **reference**

The IXTCC of the connection to close.

DESCRIPTION

The CLOSE_AND_DEALLOCATE_AST routine is invoked as an AST by the XTFT$A_CLOSE routine to complete the connection-close process. Once CLOSE_AND_DEALLOCATE_AST executes, it is assumed that neither the transport caller nor any part of transport will refer to this connection again.

CLOSE_AND_DEALLOCATE_AST removes the IXTCC from the IXTCC queue in the XTDB, decrements the reference count in the XTDB, and releases the XTCBs on the communication queue.

CLOSE_AND_DEALLOCATE_AST also zeroes various fields in the XTCC and IXTCC to catch any subsequent references to the connection and deallocates the remaining connection structures.

CLOSE_AND_DEALLOCATE_AST is invoked in executive mode.

Section 8.3.10 shows a sample implementation of the CLOSE_AND_DEALLOCATE_AST routine.

DECW$TRANSPORT_INIT

Initializes and returns the XTFT data structure.

FORMAT

xtft=**DECW$TRANSPORT_INIT**

RETURNS

VMS usage: **record**
type: **xtft**
access: **write**
mechanism: **reference**

Returns a pointer to the XTFT data structure.

ARGUMENTS

None.

DESCRIPTION

DECW$TRANSPORT_INIT initializes and returns the XTFT data structure through which the transport-specific routines are called.

The common transport must always be able to find the transport-specific XTFT structures. To ensure that the common transport can find the XTFT structures, all specific transports provide a transfer vector to the DECW$TRANSPORT_INIT routine as the first image section in the specific transport shareable image.

The DECW$XPORT_ATTACH_TRANSPORT routine calls DECW$TRANSPORT_INIT to initialize and return the XTFT data structure. Once the XTFT is initialized, DECW$XPORT_ATTACH_TRANSPORT calls the XTFT$A_ATTACH_TRANSPORT routine to complete the transport-specific initialization.

The DECW$TRANSPORT_INIT routine is called in executive mode.

See Section 8.3.19 for a sample implementation of the DECW$TRANSPORT_INIT routine and the associated transfer vector.

DETACH_AND_POLL

Detaches from the transport and starts polling for a transport restart. Internal to specific transport.

FORMAT

DETACH_AND_POLL *tdb*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value in R0.

ARGUMENT

tdb

VMS usage:	**record**
type:	**xtdb**
access:	**modify**
mechanism:	**reference**

The XTDB of the transport that you want to poll.

DESCRIPTION

The DETACH_AND_POLL routine releases the transport's connection-specific and connection-accept channels and then polls the transport to attempt a restart. DETACH_AND_POLL does not attempt to restart the transport if the XTDB$V_DYING bit is set. If it is unable to poll for the restart, DETACH_AND_POLL reports that the reattach attempt failed.

DETACH_AND_POLL is invoked in executive mode.

See Section 8.3.16 for a sample implementation of the DETACH_AND_POLL routine.

FREE_INPUT_AST

AST completion routine for transport read operations. Internal to specific transport.

FORMAT

FREE_INPUT_AST *xtcb*

ARGUMENT

xtcb

VMS usage: **record**
type: **xtcb**
access: **modify**
mechanism: **reference**

The XTCB to insert on the input work queue.

DESCRIPTION

FREE_INPUT_AST is used by the XTFT$A_FREE_INPUT_BUFFER routine with the XTCB buffer returned by the $QIO read. FREE_INPUT_AST inserts this XTCB on the tail of the input work queue and then attempts to remove a free input buffer and initiate a $QIO read into it.

FREE_INPUT_AST must check if this connection is aborting or the I/O failed, and if so, perform failure processing. If FREE_INPUT_AST is the first to set the dying bit, it must also perform abort notification. Abort notification consists of conditionally declaring a user-mode AST for the link abort (by means of XPORT_ABORT_SEND), sending notification that a write operation has completed to complete any output wait condition, and sending notification that a read operation has completed to complete any input wait condition.

FREE_INPUT_AST also determines whether the large or small XTCBs should be used in subsequent I/O operations. If the $QIO read operation that just completed filled a small request packet, FREE_INPUT_AST uses the large request packet. If large XTCBs were being used and the last read operation would have fit in a small XTCB, FREE_INPUT_AST shifts down to the small XTCBs.

FREE_INPUT_AST then attempts to remove an XTCB from the free queue and initiate a $QIO read operation into it. FREE_INPUT_AST specifies itself as the read-completion routine. If the $QIO read operation fails, FREE_INPUT_AST marks the connection as dying and performs connection-abort processing.

FREE_INPUT_AST is invoked in user or executive mode.

See Section 8.3.8 for a sample implementation of the FREE_INPUT_AST routine.

REATTACH_AST

Attempts to reattach the transport when the timer interval has expired. Internal to specific transport.

FORMAT

REATTACH_AST *tdb*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write**
mechanism: **value**

Returns a longword condition value in R0.

ARGUMENT

tdb
VMS usage: **record**
type: **xtdb**
access: **modify**
mechanism: **reference**
The XTDB of the transport that you want to reattach.

DESCRIPTION

REATTACH_AST is the AST completion routine for the $SETIMR system service invoked by the DETACH_AND_POLL routine. REATTACH_AST calls the transport-specific XTFT$A_ATTACH_TRANSPORT routine to reattach the transport.

REATTACH_AST is invoked in executive mode.

See Section 8.3.17 for a sample implementation of the REATTACH_AST routine.

TRANSPORT_OPEN_CALLBACK

Performs a callback to the client during the connection-open sequence. Internal to specific transport.

FORMAT

TRANSPORT_OPEN_CALLBACK *itcc*

ARGUMENT

itcc

VMS usage: **record**
type: **ixtcc**
access: **modify**
mechanism: **reference**

The IXTCC associated with this connection.

DESCRIPTION

TRANSPORT_OPEN_CALLBACK performs a callback to the client during the connection-open sequence. Transport semantics require a callback, as opposed to a simple AST, during the connection-initiation sequence of a connect-to-server operation. Because the callback cannot be performed in executive mode, TRANSPORT_OPEN_CALLBACK is invoked as a user-mode AST to complete this operation.

If the user accepts the connection, TRANSPORT_OPEN_CALLBACK populates the communication queue with transport buffers and initiates I/O. If it fails, TRANSPORT_OPEN_CALLBACK generates a message and releases the connection resources.

The TRANSPORT_OPEN_CALLBACK routine is called in user-AST mode.

See Section 8.3.15 for a sample implementation of the TRANSPORT_OPEN_CALLBACK routine.

TRANSPORT_READ_AST

Read-completion AST routine for the transport's network channel. Internal to specific transport.

FORMAT

TRANSPORT_READ_AST *tdb*

ARGUMENT

tdb

VMS usage:	**record**
type:	**xtdb**
access:	**modify**
mechanism:	**reference**

The XTDB associated with this connection.

DESCRIPTION

TRANSPORT_READ_AST is a read-completion AST routine for the transport's network channel. The TRANSPORT_READ_AST routine receives connection request notifications only. All other requests from clients are sent by means of the connection-specific channel, not the transport-specific channel, and do not follow the TRANSPORT_READ_QUEUE to TRANSPORT_READ_AST code path.

TRANSPORT_READ_AST must allocate and initialize an XTCC, put it on the XTDB for this transport, and then call the connection request action routine XTDB$A_CONNECT_REQUEST provided by the server.

The TRANSPORT_READ_AST routine is called in executive mode.

See Section 8.3.14 for a sample implementation of the TRANSPORT_READ_AST routine.

TRANSPORT_READ_QUEUE

Initiates an asynchronous connection-accept operation. Internal to specific transport.

FORMAT

TRANSPORT_READ_QUEUE *tdb*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value in R0.

ARGUMENT

tdb

VMS usage:	**record**
type:	**xtdb**
access:	**modify**
mechanism:	**reference**

The XTDB associated with this connection.

DESCRIPTION

TRANSPORT_READ_QUEUE initiates an asynchronous connection-accept operation.

The XTFT$A_ATTACH_TRANSPORT routine assigns a network channel for the transport and then calls TRANSPORT_READ_QUEUE to listen on the channel for a connection attempt from a client.

In the case of the sample transport, the TRANSPORT_READ_QUEUE routine assigns a channel for this connection and then does a $QIO read operation with function IO$_ACCESS or IO$M_ACCEPT on the transport-specific channel created by XTFT$A_ATTACH_TRANSPORT. This $QIO performs the following functions:

- "Listens" on this channel for a $QIO connection request from a client.
- When a connection request from a client is received, the $QIO gets a description of the client node from the client.
- Calls the TRANSPORT_READ_AST routine to accept the connection. The transport must accept all valid connection requests; the server later decides if it wants to reject a connection from the client.

The TRANSPORT_READ_QUEUE routine is called in executive mode.

See Section 8.3.13 for a sample implementation of the TRANSPORT_READ_QUEUE routine.

WRITE_AST

AST completion routine for transport write operations. Internal to specific transport.

FORMAT

WRITE_AST *xtcb*

ARGUMENT

xtcb

VMS usage: **record**
type: **xtcb**
access: **modify**
mechanism: **reference**
The XTCB to return to the appropriate free queue.

DESCRIPTION

WRITE_AST is an AST completion routine for transport write operations. WRITE_AST is used by the XTFT$A_WRITE routine to return an XTCB to the large or small free queue after a successful write operation and to initiate another write operation.

If the $QIO failed, or the connection is dying, WRITE_AST performs failure processing and prevents further operations on this connection. Failure processing is dependent upon the type of transport being used. In the case of the TCP/IP service provided by the ULTRIX Connection product (UCX), a failed I/O attempt to the connection indicates that a connection has aborted. When an I/O operation completes and the status indicates failure, the routine must perform all logical-link rundown operations, including setting the dying bit and error status, completing any process waits for input or output, and sending notification to the process that the connection has died.

After WRITE_AST returns the XTCB to the free queue, it attempts to remove another XTCB from the head of the output work queue and initiate a $QIO write operation. If the queue was empty, WRITE_AST clears the write disable flag for this connection by means of the XPORT_OUT_WRITE_ENABLE macro.

WRITE_AST is invoked in user or executive mode.

See Section 8.3.4 for a sample implementation of the WRITE_AST routine.

XTFT$A_ATTACH_TRANSPORT

Performs the transport-specific initialization functions.

FORMAT

XTFT$A_ATTACH_TRANSPORT *tdb*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value in R0.

ARGUMENT

tdb

VMS usage:	**record**
type:	**xtdb**
access:	**modify**
mechanism:	**reference**

XTDB structure. This will have been initialized by the caller prior to entry.

DESCRIPTION

As described in Section 3.3.2.1, the DECW$XPORT_ATTACH_TRANSPORT routine calls the XTFT$A_ATTACH_TRANSPORT routine to complete the transport-specific initialization.

XTFT$A_ATTACH_TRANSPORT functions differently depending on what calls it.

If Xlib calls it, XTFT$A_ATTACH_TRANSPORT does only some transport-specific initialization because clients attach specific transports when they open a connection.

If the server calls it, as determined by the XTDB XTDB$V_MODE field, XTFT$A_ATTACH_TRANSPORT creates a mailbox (if DECnet) and gets a channel to the device by means of $ASSGN. The specific transport uses this channel to listen for connection requests from clients; the specific transport later assigns a separate channel for each connection.

XTFT$A_ATTACH_TRANSPORT also indirectly calls the server's connection_request routine to see if the server accepts a connection from this client. If the returned status indicates failure, XTFT$A_ATTACH_TRANSPORT deallocates all resources acquired to this point and quits.

Once the channel is established, XTFT$A_ATTACH_TRANSPORT starts a $QIO read (by means of TRANSPORT_READ_QUEUE) on the channel.

XTFT$A_ATTACH_TRANSPORT is invoked in user or executive mode.

See Section 8.3.12 for a sample implementation of the XTFT$A_ATTACH_TRANSPORT routine.

XTFT$A_CLOSE

Closes a transport connection.

FORMAT

XTFT$A_CLOSE *itcc*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value in R0.

ARGUMENT

itcc

VMS usage:	**record**
type:	**ixtcc**
access:	**modify**
mechanism:	**by reference**

Address of the connection to close.

DESCRIPTION

The XTFT$A_CLOSE routine initiates a series of operations that destroy a connection and release the data structures associated with the link.

After XTFT$A_CLOSE is called, the transport user must not refer to any structures associated with the connection. The transport begins deallocation of all connection resources including, but not limited to, channels, XTCC, XTCBs, XTPB, and transport-private data.

XTFT$A_CLOSE is invoked in executive mode.

See Section 8.3.9 for a sample implementation of the XTFT$A_CLOSE routine.

XTFT$A_EXECUTE_FREE

Returns an XTCB to a local connection.

FORMAT

XTFT$A_EXECUTE_FREE *tcc, nullarg, type, free_queue*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write**
mechanism: **value**

Returns a longword condition value in R0.

ARGUMENTS

tcc

VMS usage: **record**
type: **xtcc**
access: **modify**
mechanism: **by reference**
Address of the XTCC for this connection.

nullarg

VMS usage: **null_arg**
type: **longword (unsigned)**
access: **read**
mechanism: **value**
Place-holding argument reserved for historical reasons. It is never referred to by XTFT$A_EXECUTE_FREE and its contents are unpredictable.

type

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**
Type of XTCB. Valid types are DECW$C_XPORT_BUFFER_LRP or DECW$C_XPORT_BUFFER_SRP.

free_queue

VMS usage: **array**
type: **longword**
access: **modify**
mechanism: **reference**
Pointer to the free queue.

DESCRIPTION

The XTFT$A_EXECUTE_FREE routine logically returns an XTCB to a logical link. The common transport DECW$XPORT_FREE_INPUT_BUFFER invokes XTFT$A_EXECUTE_FREE after it returns an input XTCB to a previously empty free queue.

XTFT$A_EXECUTE_FREE checks to see if input_free operations are enabled for this connection and tries to remove the first XTCB. If it successfully removes the XTCB, XTFT$A_EXECUTE_FREE initiates a $QIO read into it by invoking DECW$$XPORT_FREE_INPUT (which invokes XTFT$A_FREE_INPUT_BUFFER).

The **nullarg** argument is a placeholder; XTFT$A_EXECUTE_FREE does a REMQHI to get the XTCB from the head of the queue.

XTFT$A_EXECUTE_FREE is invoked in user mode.

See Section 8.3.6 for a sample implementation of the XTFT$A_EXECUTE_FREE routine.

XTFT$A_EXECUTE_WRITE

Writes an XTCB to a transport-specific connection.

FORMAT

XTFT$A_EXECUTE_WRITE *xtcc, nullarg, mode*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write**
mechanism: **value**

Returns a longword condition value in R0.

ARGUMENTS

xtcc

VMS usage: **record**
type: **xtcc**
access: **modify**
mechanism: **reference**
Address of the XTCC for this connection.

nullarg

VMS usage: **null_arg**
type: **longword (unsigned)**
access: **read**
mechanism: **value**
Place-holding argument reserved for historical reasons.

mode

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**
Modifying flags for the write operation. The valid field is:

Constant	Description
DECW$M_MODE_NOBLOCK	Nonblocking write. If no buffer is available when an attempt is made to allocate one, do not block but return the status value DECW$_BUFNOTAVL.

DESCRIPTION

XTFT$A_EXECUTE_WRITE is invoked to write an XTCB to a transport connection. XTFT$A_EXECUTE_WRITE is expected to remove the head XTCB from the output work queue and, if the remove was successful, initiate a write operation for the XTCB by invoking the common transport DECW$$XPORT_WRITE routine. DECW$$XPORT_WRITE then calls XTFT$A_WRITE to send the buffer.

The **nullarg** argument is a placeholder retained for historical reasons; its value is never accessed or relied upon.

XTFT$A_EXECUTE_WRITE is invoked in user mode.

See Section 8.3.2 for a sample implementation of the XTFT$A_EXECUTE_WRITE routine.

XTFT$A_FREE_INPUT_BUFFER

Starts a read operation on a freed input buffer.

FORMAT

XTFT$A_FREE_INPUT_BUFFER *itcc, tcb*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value in R0.

ARGUMENTS

itcc

VMS usage:	**record**
type:	**ixtcc**
access:	**modify**
mechanism:	**by reference**

Address of the connection where communication takes place.

tcb

VMS usage:	**record**
type:	**xtcb**
access:	**modify**
mechanism:	**by reference**

Address of the XTCB to free.

DESCRIPTION

The XTFT$A_FREE_INPUT_BUFFER routine does a $QIO read operation for a connection into the provided buffer. If the read $QIO fails, XTFT$A_FREE_INPUT_BUFFER inserts the XTCB on the free queue and sets the connection state to dying.

The transport-common DECW$$XPORT_FREE_INPUT routine calls the XTFT$A_FREE_INPUT_BUFFER routine to perform a $QIO read operation for the connection.

Unlike the XTFT$A_EXECUTE_FREE routine, the **xtcb** argument is a "real" XTCB and it is assumed that any enable/disable checks, performed with the XPORT_IN_FREE_DISABLE macro, have already been performed.

XTFT$A_FREE_INPUT_BUFFER is invoked in executive mode.

See Section 8.3.7 for a sample implementation of the XTFT$A_FREE_INPUT_BUFFER routine.

XTFT$A_OPEN

Attempts to establish a connection to a server.

FORMAT

XTFT$A_OPEN *workstation, server, itcc*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value in R0.

ARGUMENTS

workstation

VMS usage:	**char string**
type:	**descriptor**
access:	**read**
mechanism:	**reference**

Name of the server object passed by descriptor. Contains network address and authentication information. The **workstation** argument is usually a node name.

server

VMS usage:	**longword**
type:	**longword**
access:	**read**
mechanism:	**value**

The number of the server to be connected. The **server** argument is usually zero because most workstations run only one server.

itcc

VMS usage:	**record**
type:	**ixtcc**
access:	**modify**
mechanism:	**reference**

Location of a preallocated IXTCC. This IXTCC has an XTPB (IXTCC$A_TPB and XTCC$A_TPB) that has been initialized.

DESCRIPTION

The XTFT$A_OPEN routine tries to connect a client to a server.

The transport-common DECW$XPORT_OPEN routine attempts to locate a transport with a name matching that passed in its **xportnam** argument (for example "DECNET"). If a matching transport is found, the XTFT$A_OPEN routine is called with the server and workstation arguments and an IXTCC that has been partially initialized.

XTFT$A_OPEN

XTFT$A_OPEN is responsible for allocating and initializing the XTCC and all necessary XTCBs, populating the XTCQ with the XTCBs, and initiating I/O on the connection. Parameters that would affect these operations are found in the XTPB attached to the IXTCC by means of the IXTCC$A_TPB field.

XTFT$A_OPEN is invoked in executive mode.

See Section 8.3.11 for a sample implementation of the XTFT$A_OPEN routine.

XTFT$A_RUNDOWN

Performs the transport-specific rundown functions required during image rundown.

FORMAT

XTFT$A_RUNDOWN *tdb*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value in R0.

ARGUMENT

tdb

VMS usage:	**record**
type:	**xtdb**
access:	**modify**
mechanism:	**reference**

XTDB structure. The XTDB is initialized by the common transport before XTFT$A_RUNDOWN is called.

DESCRIPTION

XTFT$A_RUNDOWN is invoked by the common transport when the image in which the transport is running exits. The transport's rundown procedure must release any resources that might survive the image exit. ASTs are disabled while XTFT$A_RUNDOWN is executing.

XTFT$A_RUNDOWN is invoked in executive mode.

See Section 8.3.18 for a sample implementation of the XTFT$A_RUNDOWN routine.

XTFT$A_WRITE

Writes an XTCB buffer from the common transport to a transport-specific connection.

FORMAT

XTFT$A_WRITE *itcc, tcb, mode*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write**
mechanism: **value**

Returns a longword condition value in R0.

ARGUMENTS

itcc

VMS usage: **record**
type: **xtcc**
access: **modify**
mechanism: **reference**
Address of the IXTCC for this connection.

tcb

VMS usage: **record**
type: **xtcb**
access: **modify**
mechanism: **reference**
Address of the XTCB to write to the transport-specific connection.

mode

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**
Modifying flags for the write operation. The valid field is:

Constant	Description
DECW$M_MODE_NOBLOCK	Nonblocking write. If no buffer is available when an attempt is made to allocate one, do not block but return the status value DECW$_BUFNOTAVL.

DESCRIPTION

The XTFT$A_WRITE routine is invoked to write the data in an XTCB, possibly by means of $QIO, to a transport connection associated with the XTCC. If there is nothing to write, that is, the XTCB is empty, XTFT$A_WRITE inserts the XTCB on the appropriate (small or large) output free queue.

If the $QIO write operation fails, XTFT$A_WRITE puts the XTCB back at the head of the output work queue and sets the connection status to dying.

Unlike XTFT$A_EXECUTE_WRITE, the XTCB parameter in the argument list is significant and is the address of an XTCB, not an element of any queue, whose data is to be written to a connection.

The XTFT$A_WRITE routine is called in executive mode.

See Section 8.3.3 for a sample implementation of the XTFT$A_WRITE routine.

XTFT$A_WRITE_USER

Attempts to write a buffer in the user's address space to a transport-specific connection.

FORMAT

XTFT$A_WRITE_USER *itcc, buffer, mode*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write**
mechanism: **value**

Returns a longword condition value in R0.

ARGUMENTS

itcc

VMS usage: **record**
type: **ixtcc**
access: **modify**
mechanism: **reference**
Address of the IXTCC data structure that identifies the connection.

buffer

VMS usage: **char string**
type: **char string**
access: **read**
mechanism: **descriptor**
A buffer in the user's address space that contains data to write to the connection.

mode

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**
Modifying flags for the write operation. The valid field is:

Constant	Description
DECW$M_MODE_NOBLOCK	Nonblocking write. If no buffer is available when an attempt is made to allocate one, do not block but return the status value DECW$_BUFNOTAVL.

DESCRIPTION

The XTFT$A_WRITE_USER routine attempts to write a buffer in the user's address space to a transport connection. The purpose of this interface is to avoid a data copy into XTCBs when the caller has a large, contiguous block of data to be written to a connection, such as when sending image data between client and server.

There are two methods for implementing this routine. The first is to wait for the output work queue to become empty and then perform the I/O operation on the user's buffer, typically by means of a $QIO. The other method is to invoke the common transport's DECW$XPORT_COPY_AND_WRITE routine for the buffer arguments. It is *strongly recommended* that transports use the DECW$XPORT_COPY_AND_WRITE routine.

XTFT$A_WRITE_USER is called in user mode.

See Section 8.3.5 for a sample implementation of the XTFT$A_WRITE_USER routine.

7 Transport Support Macros

This chapter describes the support macros that you can use if you write your own transport-specific component. These routines are provided for your convenience; there is no requirement that you use them, but you must implement similar functions.

The transport support macros are located in the file SYS$LIBRARY:DECW$XPORTMAC.R32.

The transport support macros are listed in Table 7–1.

Table 7–1 Transport Support Macros

Routine	Function
XPORT_IN_NOTIFY_SET	Requests input notification.
XPORT_IN_NOTIFY_CLEAR	Clears a request-for-input notification.
XPORT_IN_NOTIFY_WAIT	Initiates a wait-for-input notification.
XPORT_IN_NOTIFY_SEND	Sends notice that an input operation has completed.
XPORT_OUT_NOTIFY_SET	Sends notice that output notification is required.
XPORT_OUT_NOTIFY_CLEAR	Sends notice that output notification is no longer required.
XPORT_OUT_NOTIFY_WAIT	Waits for output notification.
XPORT_OUT_NOTIFY_SEND	Sends notice that an output operation has completed.
XPORT_XTCB_FILLED	Returns the number of data bytes in an XTCB.
XPORT_XTCB_TOTAL	Determines the total number of bytes in the data area of an XTCB.
XPORT_XTCB_FREE	Determines the number of unused bytes in the data area of an XTCB.
XPORT_WRITE_WAIT	Waits for the output work queue to empty.
XPORT_WRITE_UNWAIT	Reenables output work queue write operations.
XPORT_ABORT_SEND	Declares a user-mode AST to the process indicating that the connection has died.
XPORT_OUT_WRITE_ENABLE	Clears the write disable flag.
XPORT_OUT_WRITE_DISABLE	Sets the write disable flag.
XPORT_OUT_STATE_SRP	Marks a switch to the use of SRPs for output and returns true (1) if an LRP was being used.
XPORT_OUT_STATE_LRP	Marks a switch to the use of LRPs for output and returns true (1) if an SRP was being used.

(continued on next page)

Table 7–1 (Cont.) Transport Support Macros

Routine	Function
XPORT_IN_STATE_SRP	Marks a switch to the use of SRPs for input and returns true (1) if an LRP was being used.
XPORT_IN_STATE_LRP	Marks a switch to the use of LRPs for input and returns true (1) if an SRP was being used.
XPORT_IN_FREE_ENABLE	Clears the free disable flag for this connection type of queue and returns true (1) if it was clear or false (0) if it was set.
XPORT_IN_FREE_DISABLE	Sets the free disable flag for this connection and type of queue and returns true (1) if it was set or false (0) if it was clear.
VALIDATE_XTCC	Validates an XTCC and returns the IXTCC.
VALIDATE_USERW	Checks user buffer for write access.
VALIDATE_USER	Checks user buffer for read access.

XPORT_IN_NOTIFY_SET

Requests input notification.

FORMAT

XPORT_IN_NOTIFY_SET *xtcc*

ARGUMENTS

xtcc

VMS usage:	**record**
type:	**xtcc**
access:	**modify**
mechanism:	**value**

The XTCC of the connection for which input notification is requested.

DESCRIPTION

The XPORT_IN_NOTIFY_SET macro clears the waiting-for-input I/O status block (IOSB) field of the XTCC and sets the XTCC$L_IWQ_FLAG bit to indicate that the transport user wants to be notified when input is delivered to the input work queue.

XPORT_IN_NOTIFY_CLEAR

Clears a request-for-input notification.

FORMAT

XPORT_IN_NOTIFY_CLEAR *xtcc*

ARGUMENTS

xtcc

VMS usage: **record**
type: **xtcc**
access: **modify**
mechanism: **reference**

The XTCC of the connection for which to cancel input notification.

DESCRIPTION

The XPORT_IN_NOTIFY_CLEAR macro clears the XTCC$L_IWQ_FLAG bit to indicate that the transport user no longer wants to be notified when input is delivered to the input work queue.

XPORT_IN_NOTIFY_WAIT

Initiates a wait-for-input notification

FORMAT

XPORT_IN_NOTIFY_WAIT *xtcc, xtpb*

ARGUMENTS

xtcc

VMS usage: **record**
type: **xtcc**
access: **modify**
mechanism: **reference**
The XTCC of the connection for which to wait for input notification.

xtpb

VMS usage: **record**
type: **xtpb**
access: **modify**
mechanism: **reference**
The XTPB of the connection for which to wait for input notification.

DESCRIPTION

The XPORT_IN_NOTIFY_WAIT macro calls the $SYNCH system service to suspend a process until input notification is set. The service is satisfied when the XTPB$W_IN_EFN flag is set and the lower word of the XTCC$W_IN_IOSB is made nonzero, as performed by the XPORT_IN_NOTIFY_SEND macro.

XPORT_IN_NOTIFY_SEND

Sends notice that an input operation has completed.

FORMAT

XPORT_IN_NOTIFY_SEND *xtcc, xtpb*

ARGUMENTS

xtcc

VMS usage: **record**
type: **xtcc**
access: **modify**
mechanism: **reference**
The XTCC of the connection for which to send input notification.

xtpb

VMS usage: **record**
type: **xtpb**
access: **modify**
mechanism: **reference**
The XTPB of the connection for which to send input notification.

DESCRIPTION

The XPORT_IN_NOTIFY_SEND macro conditionally performs the operations that inform a process that an input operation has completed. These operations consist of sending a user-mode AST to the process in the case where the XTPB$A_I_NOTIFY_RTNADR field points to a procedure to call for input notification, or of code that completes the $SYNCH system service call performed by the XPORT_IN_NOTIFY_WAIT macro.

XPORT_IN_NOTIFY_SEND sends the AST only if the previous AST (identified by the XTCC$V_IN_AST_IN_PROG field) has been delivered, that is, the field was clear. This prevents EXQUOTA errors due to excessive use of ASTs.

XPORT_OUT_NOTIFY_SET

Sends notice that output notification is required.

FORMAT

XPORT_OUT_NOTIFY_SET *xtcc, type*

ARGUMENTS

xtcc

VMS usage: **record**
type: **xtcc**
access: **modify**
mechanism: **reference**

The XTCC of the connection for which you want to receive output notification.

type

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**

The type of output free queue you are interested in. Valid types are DECW$C_XPORT_BUFFER_SRP and DECW$C_XPORT_BUFFER_LRP.

DESCRIPTION

The XPORT_OUT_NOTIFY_SET macro clears the waiting-for-output I/O status block (IOSB) field of the XTCC and sets the XTCC$L_OFSQ_FLAG or XTCC$L_OFLQ_FLAG bit to indicate that you are waiting for the output free queue to become empty.

XPORT_OUT_NOTIFY_CLEAR

Sends notice that output notification is no longer required.

FORMAT

XPORT_OUT_NOTIFY_CLEAR *xtcc, type*

ARGUMENTS

xtcc

VMS usage: **record**
type: **xtcc**
access: **modify**
mechanism: **reference**

The XTCC of the connection for which you want to cancel output notification.

type

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**

The type of output free queue you are no longer interested in. Valid types are DECW$C_XPORT_BUFFER_SRP and DECW$C_XPORT_BUFFER_LRP.

DESCRIPTION

The XPORT_OUT_NOTIFY_CLEAR macro clears the XTCC$L_OFSQ_FLAG or XTCC$L_OFLQ_FLAG bit to indicate that you do not want to receive output notification. The XPORT_OUT_NOTIFY_CLEAR macro reverses the effect of the XPORT_OUT_NOTIFY_SET macro.

XPORT_OUT_NOTIFY_WAIT

Waits for output notification.

FORMAT

XPORT_OUT_NOTIFY_WAIT *xtcc, xtpb*

ARGUMENTS

xtcc

VMS usage:	**record**
type:	**xtcc**
access:	**modify**
mechanism:	**reference**

The XTCC of the connection for which you want to wait for output notification.

xtpb

VMS usage:	**record**
type:	**xtpb**
access:	**modify**
mechanism:	**reference**

The XTPB of the connection for which you want to wait for output notification.

DESCRIPTION

The XPORT_IN_NOTIFY_WAIT macro calls the $SYNCH system service to suspend a process until output notification is set. The service is satisfied when the XTPB$W_ON_EFN flag is set and the lower word of the XTCC$W_ON_IOSB field is made nonzero, as performed by the XPORT_OUT_NOTIFY_SEND macro.

XPORT_OUT_NOTIFY_SEND

Sends notice that an output operation has completed.

FORMAT

XPORT_OUT_NOTIFY_SEND *xtcc, xtpb, type*

ARGUMENTS

xtcc

VMS usage:	**record**
type:	**xtcc**
access:	**modify**
mechanism:	**reference**

The XTCC of the connection for which you want to send output notification.

xtpb

VMS usage:	**record**
type:	**xtpb**
access:	**modify**
mechanism:	**reference**

The XTPB of the connection for which you want to send output notification.

type

VMS usage:	**longword**
type:	**longword**
access:	**read**
mechanism:	**value**

The type of output free queue you are interested in. Valid types are DECW$C_XPORT_BUFFER_SRP and DECW$C_XPORT_BUFFER_LRP.

DESCRIPTION

The XPORT_OUT_NOTIFY_SEND macro conditionally performs the operations that inform a process that an output operation has completed. These operations consist of sending a user-mode AST to the process in the case where the XTPB$A_O_NOTIFY_RTNADR field points to a procedure to call for output notification, or of code that completes the $SYNCH system service call performed by the XPORT_OUT_NOTIFY_WAIT macro.

XPORT_XTCB_FILLED

Returns the number of data bytes in an XTCB.

FORMAT

XPORT_XTCB_FILLED *xtcb*

ARGUMENTS

xtcb

VMS usage: **record**
type: **xtcb**
access: **modify**
mechanism: **reference**

The XTCB for which you want to return the number of data bytes in an XTCB.

DESCRIPTION

The XPORT_XTCB_FILLED macro returns the number of data bytes in an XTCB.

XPORT_XTCB_TOTAL

Determines the total number of bytes in the data area of an XTCB.

FORMAT

XPORT_XTCB_TOTAL *xtcb*

ARGUMENTS

xtcb

VMS usage:	**record**
type:	**xtcb**
access:	**modify**
mechanism:	**reference**

The XTCB for which you want to determine the total number of bytes in the data area of an XTCB.

DESCRIPTION

The XPORT_XTCB_TOTAL macro determines the total number of bytes in the data area of an XTCB.

XPORT_XTCB_FREE

Determines the number of unused bytes in the data area of an XTCB.

FORMAT

XPORT_XTCB_FREE *xtcb*

ARGUMENTS

xtcb

VMS usage: **record**
type: **xtcb**
access: **modify**
mechanism: **reference**
The XTCB for which you want to determine the number of unused bytes in the data area of an XTCB.

DESCRIPTION

The XPORT_XTCB_FREE macro determines the number of unused bytes in the data area of an XTCB.

XPORT_WRITE_WAIT

Waits for the output work queue to empty.

FORMAT

XPORT_WRITE_WAIT *xtcc, xtpb*

ARGUMENTS

xtcc

VMS usage: **record**
type: **xtcc**
access: **modify**
mechanism: **reference**
The XTCC for the connection on which you want to wait.

xtpb

VMS usage: **record**
type: **xtpb**
access: **modify**
mechanism: **reference**
The XTPB for the connection on which you want to wait.

DESCRIPTION

The XPORT_WRITE_WAIT macro performs a $SYNCH system service to wait for the output work queue to empty. XPORT_WRITE_WAIT is similar to XPORT_OUT_NOTIFY_WAIT, but uses the XTCC$W_OW_IOSB field. Also, because there are no equivalents to the XPORT_OUT_NOTIFY_SET and XPORT_OUT_NOTIFY_CLEAR macros, you must manually clear the XTCC$W_OW_IOSB field, or set or clear the XTCC$V_WAIT_ON_WRITE bit.

XPORT_WRITE_UNWAIT

Reenables output work queue write operations.

FORMAT

XPORT_WRITE_UNWAIT *xtcc, xtpb*

ARGUMENTS

xtcc

VMS usage: **record**
type: **xtcc**
access: **modify**
mechanism: **reference**
The XTCC for which you no longer want to wait.

xtpb

VMS usage: **record**
type: **xtpb**
access: **modify**
mechanism: **reference**
The XTPB for which you no longer want to wait.

DESCRIPTION

The XPORT_WRITE_UNWAIT macro cancels the wait on the output work queue initiated by XPORT_WRITE_WAIT.

XPORT_ABORT_SEND

Declares a user-mode AST to the process indicating that the connection has aborted.

FORMAT

XPORT_ABORT_SEND *xtdb, xtcc*

ARGUMENTS

xtdb

VMS usage: **record**
type: **xtdb**
access: **modify**
mechanism: **reference**

The XTDB for which you want to abort the connection.

xtcc

VMS usage: **record**
type: **xtcc**
access: **modify**
mechanism: **reference**

The XTCC for which you want to abort the connection.

DESCRIPTION

The XPORT_ABORT_SEND macro declares a user-mode AST to the user-provided connection-abort notification process, identified by the XTDB$A_CONNECT_ABORT field, indicating that the connection has died.

XPORT_ABORT_SEND is called as part of the abort notification.

XPORT_OUT_WRITE_ENABLE

Clears the write disable flag.

FORMAT

XPORT_OUT_WRITE_ENABLE *xtcc*

RETURNS

VMS usage:	**longword**
type:	**longword**
access:	**write**
mechanism:	**value**

Returns true (1) if the XTCC$L_OWQ_FLAG field was clear, that is, if write operations were already enabled, or false (0) if it was set.

ARGUMENTS

xtcc

VMS usage:	**record**
type:	**xtcc**
access:	**modify**
mechanism:	**reference**

The XTCC for which you want to enable write operations.

DESCRIPTION

The XPORT_OUT_WRITE_ENABLE macro clears the write disable flag for this connection and returns true (1) if it was clear or false (0) if it was set.

XPORT_OUT_WRITE_DISABLE

Sets the write disable flag.

FORMAT

XPORT_OUT_WRITE_DISABLE *xtcc*

RETURNS

VMS usage:	**longword**
type:	**longword**
access:	**write**
mechanism:	**value**

Returns true (1) if the XTCC$L_OWQ_FLAG was set, that is, if write operations were already disabled, or false (0) if it was clear.

ARGUMENTS

xtcc

VMS usage:	**record**
type:	**xtcc**
access:	**modify**
mechanism:	**reference**

The XTCC for which you want to disable write operations.

DESCRIPTION

The XPORT_OUT_WRITE_DISABLE macro sets the write disable flag for this connection and returns true (1) if it was set or false (0) if it was clear.

XPORT_OUT_STATE_SRP

Marks a switch to the use of SRPs for output and returns true (1) if an LRP was being used.

FORMAT

XPORT_OUT_STATE_SRP *xtcc*

RETURNS

VMS usage: **longword**
type: **longword**
access: **write**
mechanism: **value**

Returns true (1) if an LRP was being used.

ARGUMENTS

xtcc
VMS usage: **record**
type: **xtcc**
access: **modify**
mechanism: **reference**
The XTCC for which you want to test and set the state.

DESCRIPTION

The XPORT_OUT_STATE_SRP macro tests to see if the XTCC$V_LRP_ON_OUTPUT bit was set, and then clears it. XPORT_OUT_STATE_SRP marks a switch to the use of SRPs for output and returns true (1) if an LRP was being used.

XPORT_OUT_STATE_LRP

Marks a switch to the use of LRPs for output and returns true (1) if an SRP was being used.

FORMAT

XPORT_OUT_STATE_LRP *xtcc*

RETURNS

VMS usage: **longword**
type: **longword**
access: **write**
mechanism: **value**

Returns true (1) if an SRP was being used.

ARGUMENTS

xtcc
VMS usage: **record**
type: **xtcc**
access: **modify**
mechanism: **reference**
The XTCC for which you want to test and set the state.

DESCRIPTION

The XPORT_OUT_STATE_LRP macro tests to see if the XTCC$V_LRP_ON_OUTPUT bit was clear, and then sets it. XPORT_OUT_STATE_SRP marks a switch to the use of LRPs for output and returns true (1) if an SRP was being used.

XPORT_IN_STATE_SRP

Marks a switch to the use of SRPs for input and returns true (1) if an LRP was being used.

FORMAT

XPORT_IN_STATE_SRP *xtcc*

RETURNS

VMS usage:	**longword**
type:	**longword**
access:	**write**
mechanism:	**value**

Returns true (1) if an LRP was being used.

ARGUMENTS

xtcc

VMS usage:	**record**
type:	**xtcc**
access:	**modify**
mechanism:	**reference**

The XTCC for which you want to test and set the state.

DESCRIPTION

The XPORT_IN_STATE_SRP macro tests to see if the XTCC$V_LRP_ON_INPUT bit was set, and then clears it. XPORT_IN_STATE_SRP marks a switch to the use of SRPs for input and returns true (1) if an LRP was being used.

XPORT_IN_STATE_LRP

Marks a switch to the use of LRPs for input and returns true (1) if an SRP was being used.

FORMAT

XPORT_IN_STATE_LRP *xtcc*

RETURNS

VMS usage: **longword**
type: **longword**
access: **write**
mechanism: **value**

Returns true (1) if an SRP was being used.

ARGUMENTS

xtcc
VMS usage: **record**
type: **xtcc**
access: **modify**
mechanism: **reference**
The XTCC for which you want to test and set the state.

DESCRIPTION

The XPORT_IN_STATE_LRP macro tests to see if the XTCC$V_LRP_ON_INPUT bit was clear, and then sets it. XPORT_IN_STATE_SRP marks a switch to the use of LRPs for input and returns true (1) if an SRP was being used.

XPORT_IN_FREE_ENABLE

Clears the free disable flag for this connection type of queue and returns (1) if it was clear or (0) if it was set.

FORMAT

XPORT_IN_FREE_ENABLE *xtcc, type*

RETURNS

VMS usage:	**longword**
type:	**longword**
access:	**write**
mechanism:	**value**

Returns true (1) if the free disable flag was clear, that is, input operations were already enabled, or false (0) if it was set.

ARGUMENTS

xtcc

VMS usage:	**record**
type:	**xtcc**
access:	**modify**
mechanism:	**reference**

The XTCC for which you want to clear the disable flag.

type

VMS usage:	**longword**
type:	**longword**
access:	**read**
mechanism:	**value**

The type of buffer for which to clear the disable flag. Valid types are DECW$XPORT_BUFFER_SRP and DECW$XPORT_BUFFER_LRP.

DESCRIPTION

The XPORT_IN_FREE_ENABLE macro tests the XTCC$L_IFSQ_FLAG flag or the XTCC$L_IFLQ_FLAG flag (depending on the **type** argument) to see if it is clear, and then clears it. XPORT_IN_FREE_ENABLE returns true (1) if the free disable flag was clear or false (0) if it was set.

XPORT_IN_FREE_DISABLE

Sets the free disable flag for this connection and type of queue and returns (1) if it was set or (0) if clear.

FORMAT

XPORT_IN_FREE_DISABLE *xtcc, type*

RETURNS

VMS usage:	**longword**
type:	**longword**
access:	**write**
mechanism:	**value**

Returns true (1) if the free disable flag was set, that is, free input operations were already disabled, or false (0) if it was clear.

ARGUMENTS

xtcc

VMS usage:	**record**
type:	**xtcc**
access:	**modify**
mechanism:	**reference**

The XTCC for which you want to set the disable flag.

type

VMS usage:	**longword**
type:	**longword**
access:	**read**
mechanism:	**value**

The type of buffer for which to set the disable flag. Valid types are DECW$XPORT_BUFFER_SRP and DECW$XPORT_BUFFER_LRP.

DESCRIPTION

The XPORT_IN_FREE_ENABLE macro tests the XTCC$L_IFSQ_FLAG flag or the XTCC$L_IFLQ_FLAG flag (depending on the **type** argument) to see if it is set, and then sets it. XPORT_IN_FREE_ENABLE returns true (1) if the free disable flag was set or false (0) if it was clear.

VALIDATE_XTCC

Validates an XTCC and returns the IXTCC.

FORMAT

status_return=**VALIDATE_XTCC** *xtcc, ixtcc*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value to R0. Possible condition values are listed under Condition Values Returned.

ARGUMENTS

xtcc

VMS usage:	**record**
type:	**xtcc**
access:	**modify**
mechanism:	**reference**

The XTCC that you want to validate.

ixtcc

VMS usage:	**record**
type:	**ixtcc**
access:	**modify**
mechanism:	**reference**

The previously registered IXTCC address is returned to this argument. Valid only if SS$_NORMAL is returned.

DESCRIPTION

The VALIDATE_XTCC macro calls the transport-common DECW$XPORT_VALIDATE_STRUCT_JSB routine to validate an XTCC. If the XTCC ID is known and is valid, VALIDATE_XTCC returns the previously registered address of the IXTCC data structure in the **ixtcc** argument.

VALIDATE_XTCC is called only by the transport-common component.

CONDITION VALUES RETURNED

SS$_ACCVIO	The XTCC is not user-readable.
SS$_NORMAL	Routine successfully completed.
SS$_BADPARAM	Bad parameter. Either the XTCC$B_SUBTYPE field is not equal to the constant DECW$C_DYN_XTCC, or the IXTCC$A_TCC field does not point to this XTCC.

Any DECW$XPORT_VALIDATE_STRUCT_JSB condition value.

VALIDATE_USERW

Checks a user buffer for write access.

FORMAT

*status_return=***VALIDATE_USERW** *bufadr, buflen*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write**
mechanism: **value**

Returns a longword condition value to R0. Possible condition values are listed under Condition Values Returned.

ARGUMENTS

bufadr

VMS usage: **record**
type: **buffer**
access: **modify**
mechanism: **reference**
The address of the buffer that you want to check for write access.

buflen

VMS usage: **longword**
type: **longword**
access: **read**
mechanism: **value**
The length of the buffer that you want to check for write access.

DESCRIPTION

The VALIDATE_USERW macro checks the user-supplied buffer for write access and returns a status.

VALIDATE_USERW is called only by the transport-common component.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed. The buffer is user-writable.
SS$_ACCVIO	The buffer is not user-writable.

VALIDATE_USER

Checks a user buffer for read access.

FORMAT

status_return=**VALIDATE_USER** *bufadr, buflen*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write**
mechanism:	**value**

Returns a longword condition value to R0. Possible condition values are listed under Condition Values Returned.

ARGUMENTS

bufadr

VMS usage:	**record**
type:	**buffer**
access:	**modify**
mechanism:	**reference**

The address of the buffer that you want to check for read access.

buflen

VMS usage:	**longword**
type:	**longword**
access:	**read**
mechanism:	**value**

The length of the buffer that you want to check for read access.

DESCRIPTION

The VALIDATE_USERW macro checks the user-supplied buffer for read access and returns a status.

VALIDATE_USER is called only by the transport-common component.

CONDITION VALUES RETURNED

SS$_NORMAL	Routine successfully completed. The buffer is user-readable.
SS$_ACCVIO	The buffer is not user-readable.

8 Writing Your Own Transport

This chapter describes the sample transport layer provided in DECW$EXAMPLES:XPORT_EXAMPLE.B32. The chapter includes a sample BLISS-32 code example for each of the transport-specific routines that you must write. This chapter also includes an example of how to compile and link the routines as a shareable image in the DECwindows environment.

The code examples collectively describe a TCP/IP transport for DECwindows layered on the ULTRIX Connection product (UCX), the implementation of TCP/IP for VMS. Your own implementation of the transport-specific routines may be different depending on the lower-level transport on which you are building.

Note: **This chapter describes how to write the transport-specific routines only. Modifications to the existing transport-common routines are not recommended or supported.**

8.1 Where to Begin

Depending upon your implementation, you will probably find that the routines that initialize a transport and establish a connection require the most modification. The following routines are included in this group:

- XTFT$A_OPEN (client side)
- XTFT$A_ATTACH_TRANSPORT (server side)
- TRANSPORT_READ_AST (server side)

You may also find that routines that primarily insert and remove buffers from the queues, such as XTFT$A_EXECUTE_FREE, can be used with minimal changes.

The sample transport layer described in this chapter uses the $QIO /AST completion interface. If the lower-level transport on which you are building uses the $QIO/AST completion interface, you should check the $QIO function codes to make sure that they are applicable in your implementation.

If the lower-level transport on which you are building does not use the $QIO/AST completion interface and instead waits for read operations to complete, the server may spend a substantial amount of time waiting.

8.1.1 Identifying the Transport-Specific Shareable Image

After you write your own transport-specific layer, you must make it known to the transport-common layer as follows:

- Make sure that the transfer vector to DECW$TRANSPORT_INIT is in the first cluster of the transport-specific shareable image, as described in Section 8.2 and Section 8.3.19.
- Make sure your transport-specific shareable image is in the form SYS$SHARE:DECW_TRANSPORT_**transport_name**.EXE. When called by either Xlib or the server, DECW$XPORT_ATTACH_TRANSPORT attempts to locate and activate an image with a name in this form.

Note: Because the common transport uses only executive-mode logical names, the logical name SYS$SHARE cannot be redefined.

As described in Section 3.3.1.2, transport names that do not contain a "$" character are reserved for third-party and customer transport images. These transport names must be in the form SYS$SHARE:DECW_TRANSPORT_transport_name.EXE.

- Copy SYS$MANAGER:DECW$PRIVATE_SERVER_SETUP.TEMPLATE to *.COM and modify the DECW$SERVER_TRANSPORTS symbolic name to include your transport. For example, DECW$SERVER_TRANSPORTS could translate to "DECNET,LOCAL,TCPIP,FOO".

  The server uses the logical name DECW$SERVER_TRANSPORTS to determine which transports to attach and initialize and calls the DECW$XPORT_ATTACH_TRANSPORT routine for each transport identified by the logical name. The **transport_name** argument specifies the transport, such as "FOO".

8.2 Compiling and Linking Options for the Transport

The DECW$EXAMPLES:DEMO_BUILD.COM procedure supplied with the DECwindows kit builds the VMS DECwindows example programs, including the sample TCP/IP transport layer described in this chapter. You can refer to this procedure for suggestions on compiling and linking a transport layer in the DECwindows environment.

Example 8–1 shows the transport-specific portion of the DEMO_BUILD.COM procedure.

Example 8–1 DEMO_BUILD.COM Procedure

```
       .
       .
       .
  $ ! If Bliss and UCX are installed on the system and DECwindows is installed
  $ ! with the common transport shareable image and Bliss programming support,
  $ ! then compile and link the example transport.
  $ !
  $ if f$search("sys$library:ucx$inetdef.r32") .eqs. "" then goto do_fortran
  $ if f$search("sys$share:decw$transport_common.exe") .eqs. "" then goto do_fortran
  $ define/nolog src$ decw$examples
  $ bliss decw$examples:xport_example
❶ $ macro decw$examples:xport_example_queue
  $ macro decw$examples:xport_example_xfer
❷ $ link/share=decw$transport_example.exe -
   xport_example,-
   xport_example_queue,-
   xport_example_xfer,-
   sys$input/opt
❸ gsmatch=lequal,12,12

❹ cluster=transfer_cluster,,,
  collect=transfer_cluster,$transfer$

❺ protect=yes
  cluster=own_cluster,,,
  collect=own_cluster,$own$
  protect=no

  cluster=code_cluster,,,
  collect=code_cluster,$code$
  sys$share:decw$transport_common/shareable
       .
       .
       .
```

❶ The DECW$EXAMPLES:XPORT_EXAMPLE_QUEUE.MAR module provides emulations of the REMQxI and INSQxI instructions that probe the memory occupied by a queue entry to see if it has user-mode write access. See Section 3.2.4 for more information.

The DECW$EXAMPLES:XPORT_EXAMPLE_XFER.MAR module generates transfer vectors for the sample transport.

❷ Link the transport-specific code as a VMS shareable image that can be accessed by the transport-common component.

❸ Prevent image activation of incompatible transport versions.

❹ Create a cluster for the transfer vector and place the named program section, in this case $transfer$, in it. The $transfer$ program section points to the DECW$TRANSPORT_INIT routine, as described in Section 8.3.19. The transfer vector must be at the beginning of the image (in the first cluster) or the transport common will not activate it.

❺ OWN variables are protected from user-mode writing.

8.3 Sample TCP/IP Transport Layer Implementation

The code examples in this section implement a sample TCP/IP DECwindows transport layer.

8.3.1 TCP/IP Transport Layer Setup

Example 8–2 shows module and data structure declarations, macro and literal definitions, and external definitions that are used in the TCP/IP implementation.

Example 8–2 TCP/IP Transport Layer Setup

```
    .
    .
    .
%TITLE   'XPORT_EXAMPLE   - Example TCP/IP Communication Library'
MODULE XPORT_EXAMPLE     (
        IDENT = 'V1.0',
        ADDRESSING_MODE(EXTERNAL = GENERAL,
                        NONEXTERNAL = WORD_RELATIVE )
        ) =
BEGIN
❶
LIBRARY 'SYS$LIBRARY:STARLET' ;
REQUIRE 'SYS$LIBRARY:UCX$INETDEF.R32' ;
REQUIRE 'SRC$:XPORTEXAMPLEDEF.R32' ;
REQUIRE 'SYS$LIBRARY:DECW$XPORTMAC.R32' ;
REQUIRE 'SYS$LIBRARY:DECW$XPORTMSG.R32' ;

❷
PSECT
    PLIT        = $CODE$                    (PIC,SHARE) ,
    CODE        = $CODE$                    (PIC,SHARE) ,
    OWN         = $OWN$                     (PIC,NOSHARE) ,
    GLOBAL      = $OWN$                     (PIC,NOSHARE) ;

❸ FORWARD ROUTINE
    DECW$$TCPIP_EXECUTE_WRITE,
    DECW$$TCPIP_WRITE,
    write_ast : NOVALUE,
    DECW$$TCPIP_WRITE_USER,
    DECW$$TCPIP_EXECUTE_FREE,
    DECW$$TCPIP_FREE_INPUT_BUFFER,
    free_input_ast : NOVALUE,
    DECW$$TCPIP_ATTACH_TRANSPORT,
    DECW$$TCPIP_CLOSE,
    close_and_deallocate_ast : NOVALUE,
    DECW$$TCPIP_OPEN,
    transport_read_queue,
    transport_read_ast : NOVALUE,
    transport_open_callback : NOVALUE,
    detach_and_poll : NOVALUE,
    reattach_ast : NOVALUE,
    DECW$$TCPIP_RUNDOWN : NOVALUE ,
    DECW$TRANSPORT_INIT ;
```

(continued on next page)

Example 8–2 (Cont.) TCP/IP Transport Layer Setup

```
❹ MACRO
    inet_dev_str    = ’UCX$DEVICE’ %,
    inet_local_node = ’UCX$INET_HOST’ %,
    swap_long( val ) = ( ( (val ^ 24) AND %X’FF000000’) OR ( (val ^ 8 )
        AND %X’FF0000’) OR ( (val ^ -8) AND %X’FF00’) OR ( (val ^ -24 )
        AND %X’FF’) ) %,
    swap_short( val ) = ( ( (val ^ 8) AND %X’FF00’) OR ( (val ^ -8) AND %X’FF’) ) %,

    xtcc_status( xtcc, status ) =
    IF NOT .xtcc [xtcc$v_err_sts_valid]
    THEN
         BEGIN
         xtcc [xtcc$l_err_status] = status ;
         xtcc [xtcc$v_err_sts_valid] = 1 ;
         END %,

    load_desc( desc, string ) = BEGIN desc [0] = %CHARCOUNT( string ) ;
        desc [1] = UPLIT( string ) ; END % ;

❺ LITERAL
        REATTACH_INTERVAL_SECS = 60,
        USER_WRITE_BY_COPY = 1,
        ASYNC_EFN = 31,
        WRITE_MAXIMUM_LENGTH = 32768,
        INET_NODE_NAME_LEN = 256
        BASE_TCP_PORT = 5000 ;

❻ OWN
    reattach_timer_id : INITIAL( 0 ),
    reattach_timer_delta : VECTOR[2] INITIAL( 0, 0 ),
    inet_dev_desc : VECTOR[2],
    tcpip_tft : $BBLOCK [xtft$c_length],
    tcpip_tdb : REF $BBLOCK,
    local_node : $BBLOCK [INET_NODE_NAME_LEN],
    lnn_desc : $BBLOCK [DSC$S_DSCDEF1] ;

❼ EXTERNAL ROUTINE
    DECW$$XPORT_FREE_INPUT,
    DECW$$XPORT_WRITE,
    DECW$XPORT_CLOSE,
    DECW$XPORT_FREE_INPUT_BUFFER,
    DECW$XPORT_COPY_AND_WRITE,
    DECW$XPORT_ALLOC_INIT_QUEUES,
    DECW$XPORT_DEALLOC_QUEUES,
    DECW$XPORT_ALLOC_PMEM,
    DECW$XPORT_DEALLOC_PMEM : NOVALUE,
    DECW$XPORT_VALIDATE_STRUCT,
    DECW$XPORT_VALIDATE_STRUCT_JSB : L_VALIDATE_STRUCT,
    DECW$XPORT_ACCEPT_FAILED,
    DECW$XPORT_ATTACHED,
    DECW$XPORT_ATTACH_LOST,
    DECW$XPORT_REATTACH_FAILED,
    DECW$XPORT_REFUSED_BY_SERVER ;

     .
     .
     .
```

❶ The included files are as follows:

- SYS$LIBRARY:STARLET includes the VAX/VMS System Service definitions.
- SYS$LIBRARY:UCX$INETDEF.R32 includes the definitions for the UCX product, the TCP/IP implementation upon which this example is layered.
- SRC$:XPORTEXAMPLEDEF.R32 includes the TCP/IP transport-specific structure definitions.
- SYS$LIBRARY:DECW$XPORTMAC.R32 includes the transport layer support macros.
- SYS$LIBRARY:DECW$XPORTMSG.R32 includes the transport layer message symbols.

❷ Define the program section (PSECT) names for the four PSECTs that BLISS implicitly uses. Code and static data should be position independent and shareable, while OWN and global data should be position independent and nonshareable. You may choose the PSECT names, but the PSECT attributes should obey these rules and be used in the link command.

❸ Forward declarations for the procedures and routines that are defined in this module.

❹ Define the following macros to be used in this module:

- **inet_dev_str** is the logical name that identifies UCX's controlling device.
- **inet_local_node** is the logical name that contains the name of the local host.
- **swap_long** converts between little-endian and big-endian unsigned longword integer formats.
- **swap_short** converts between little-endian and big-endian unsigned word integer formats.
- **xtcc_status** stores a condition code in an XTCC's error status field (XTCC$L_ERR_STATUS) exactly once.
- **load_desc** builds a 2-longword string descriptor suitable for system services and internal descriptors.

❺ Define the following constants used in this module:

- **REATTACH_INTERVAL_SECS** controls the number of seconds between transport restart attempts.
- **USER_WRITE_BY_COPY** controls whether the DECW$$TCPIP_WRITE_USER routine implements its function as a call to the common transport DECW$XPORT_COPY_AND_WRITE routine.
- **ASYNC_EFN** is the number of the event flag to be used in asynchronous network I/O operations. Event flag 31 is used for such operations by convention.

- **WRITE_MAXIMUM_LENGTH** has meaning only if **USER_WRITE_BY_COPY** tests false. Then, the DECW$$TCPIP_WRITE_USER routine implements a "true" write-from-user's-buffer function and the value of this literal is the maximum size of any one $QIO. It should be a multiple of 4 and representable in 15 bits.
- **INET_NODE_NAME_LEN** is the maximum length of any Internet node name.
- **BASE_TCP_PORT** is the TCP/IP port used by server number 0. The convention for TCP/IP is that server number 0 listens on port 6000. Port 5000 is used in this example to prevent collision with a "real" TCP/IP transport.

❻ Allocate the following data structures that are private to this transport:

- **reattach_timer_id** is the identifier of this timer.
- **reattach_timer_delta** is the time to wait between polling attempts.
- **inet_dev_desc** is a 2-longword descriptor of the name of UCX's controlling device (macro inet_dev_str). It is initialized in the DECW$$TCPIP_ATTACH_TRANSPORT routine.
- **tcpip_tft** is the XTFT structure for the TCP/IP transport. It is initialized in the DECW$TRANSPORT_INIT routine.
- **tcpip_tdb** is the address of the XTDB allocated by the common transport on behalf of the TCP/IP transport. It is initialized in the DECW$$TCPIP_ATTACH_TRANSPORT routine.
- **local_node** is where the translation of the local node name logical name (UCX$DEVICE) is stored. It is initialized in the DECW$$TCPIP_ATTACH_TRANSPORT routine.
- **lnn_desc** is a 2-longword descriptor for the logical name that represents the local node name. It is initialized in the DECW$$TCPIP_ATTACH_TRANSPORT routine.

❼ Declare all references to external procedures. These are all resident in the common transport shareable image.

8.3.2 Sample XTFT$A_EXECUTE_WRITE Routine

XTFT$A_EXECUTE_WRITE is invoked to write an XTCB to a transport connection.

The procedure is expected to remove the head XTCB from the output work queue and, if the remove was successful, initiate I/O on the data in the XTCB.

Note that the xtcb parameter in the call sequence is merely a placeholder retained for historical reasons; its value is never accessed or relied upon. Example 8–3 shows a sample implementation of the XTFT$A_EXECUTE_WRITE routine.

Example 8–3 Sample XTFT$A_EXECUTE_WRITE Routine

```
    .
    .
    .
ROUTINE DECW$$TCPIP_EXECUTE_WRITE(
        xtcc:           REF $BBLOCK,
        xtcb:           REF $BBLOCK,
        mode
) =

BEGIN
BUILTIN
    REMQHI ;
LOCAL
    tcb : REF $BBLOCK,
    status ;

❶ IF NOT xport_out_write_disable( xtcc )
THEN
    BEGIN
  ❷ WHILE ( status = REMQHI( .xtcc [xtcc$a_ow_queue], tcb ) )
          EQL xport$k_queue_locked DO
        WHILE ..xtcc [xtcc$a_ow_queue] DO ;
    IF .status EQL xport$k_queue_no_entry
    THEN
        xport_out_write_enable( xtcc )
    ELSE
      ❸ RETURN DECW$$XPORT_WRITE( .xtcc, .tcb, .mode ) ;
    END ;

SS$_NORMAL
END ;
    .
    .
    .
```

❶ If write operations are enabled, then disable them and execute the following block. Otherwise, leave them disabled and return.

❷ Attempt to remove an XTCB from the head of the output work queue. If there was no XTCB on the work queue, enable future write operations and return.

❸ If the queue was not empty, initiate a write operation for the XTCB by invoking the common transport routine (DECW$$XPORT_WRITE). This form of write operation expects a “real” XTCB in the argument list, not just a placeholder.

8.3.3 Sample XTFT$A_WRITE Routine

The XTFT$A_WRITE routine is invoked to write an XTCB to a transport connection. Unlike XTFT$A_EXECUTE_WRITE, the XTCB parameter in the argument list is significant and is the address of an XTCB, not an element of any queue, whose data is to be written to a connection.

Example 8–4 shows a sample implementation of the XTFT$A_WRITE routine.

Example 8–4 Sample XTFT$A_WRITE Routine

```
      .
      .
      .
  GLOBAL ROUTINE DECW$$TCPIP_WRITE( itcc : REF $BBLOCK VOLATILE, tcb : REF
  $BBLOCK, mode ) =

  BEGIN
  BUILTIN
      TESTBITCS,
      TESTBITCC ;

  BIND
     xtpb = .itcc [ixtcc$a_tpb] : $BBLOCK,
     xtcc = .itcc [ixtcc$a_tcc] : $BBLOCK,
     xtcb = .tcb : $BBLOCK ;

  LOCAL
      status ;

❶ IF .xtcc [xtcc$v_dying]
  THEN
      BEGIN
      $INSQHI( xtcb, .itcc [ixtcc$a_ow_queue] ) ;
      RETURN DECW$_CNXABORT ;
      END ;
❷ IF .xtcb [xtcb$l_length] GTRU xport_xtcb_total( xtcb )
  THEN
      RETURN SS$_IVBUFLEN ;

❸ IF .xtcb [xtcb$l_length] EQLU 0
  THEN
      BEGIN
      IF .xtcb [xtcb$b_subtype] EQLU decw$c_dyn_xtcb_srp
      THEN
          status = $INSQHI( xtcb, .itcc [ixtcc$a_ofs_queue] )
      ELSE
          status = $INSQHI( xtcb, .itcc [ixtcc$a_ofl_queue] ) ;

    ❹ IF .status EQL xport$k_queue_corrupted
      THEN
         BEGIN
         xtcc [xtcc$v_dying] = 1 ;
         xtcc_status( xtcc, DECW$_BADQUEUE ) ;
         RETURN DECW$_CNXABORT ;
         END ;

      RETURN SS$_NORMAL ;
      END ;
```

(continued on next page)

Example 8–4 (Cont.) Sample XTFT$A_WRITE Routine

```
❺ xtcb [xtcb$l_rflink] = xtcc ;
  IF NOT (status = $QIO(  EFN =.itcc [ixtcc$w_efn],
                          FUNC = IO$_WRITEVBLK,
                          CHAN = .itcc [ixtcc$w_chan],
                          IOSB = xtcb [xtcb$w_iosb],
                          ASTADR = write_ast,
                          ASTPRM = xtcb,
                          P1 = xtcb [xtcb$t_data],
                          P2 = .xtcb [xtcb$l_length] ) )
❻ THEN
      BEGIN
      $INSQHI( xtcb, .itcc [ixtcc$a_ow_queue] ) ;
      xtcc [xtcc$v_dying] = 1 ;
      xtcc_status( xtcc, .status ) ;
      END ;
  .status
  END ;
      .
      .
      .
```

❶ See if the connection is marked dying. If so, return the XTCB to the head of the output work queue and return with a status indicating that the connection has died.

❷ This is a consistency check on the information provided in the XTCB. If it is bad, return a fatal status.

❸ If this XTCB is empty, return it to either the large or small output free queue, as appropriate for the XTCB type in the XTCB$B_SUBTYPE field, and return with a successful status.

❹ If the queue was found to be corrupted, close the connection and return with a fatal status.

❺ The XTCB has a valid data length. Initiate a write $QIO on the data in the XTCB. **P1** is the address of the first byte of user data in the XTCB; **P2** is the length of the data in the buffer. The **ASTADR** argument specifies a WRITE_AST routine, as described in Section 8.3.4. **ASTPRM** is the address of the XTCB being operated on.

❻ If the $QIO service failed, return the XTCB to the head of the output work queue and mark the connection as dying. Use the XTCC_STATUS macro to save the failure code in the XTCC if it has not yet been set.

8.3.4 Sample WRITE_AST Routine

WRITE_AST is an AST completion routine for TCP/IP write operations. WRITE_AST returns the XTCB to the appropriate free queue. If the $QIO failed or the connection is dying, failure processing is performed and the transport prohibits further operations on this connection.

Failure processing is dependent upon the type of transport being used. In the case of the TCP/IP implemented by UCX, a failed I/O attempt to the connection indicates that a connection has aborted. When an I/O operation completes and the status indicates failure, the code must perform all logical-link rundown operations including setting the dying bit and error status, completing any process waits for input or output, and sending notification to the process that the connection has died.

Other transports such as DECnet provide a separate mechanism for receiving notice that a connection has aborted. For such a transport, these logical-link rundown operations need only be performed by the code that receives this notification.

If the I/O completed successfully, the procedure attempts to remove another XTCB from the head of the output work queue and initiate a write operation on this XTCB. If the queue was empty, the procedure must enable write operations on the connection to cause the common transport to call the specific transport when an XTCB is next inserted on the output work queue. If the queue was not empty, the I/O operation is performed and the usual error processing is performed on the result. Example 8–5 shows a sample implementation of the WRITE_AST routine.

Example 8–5 Sample WRITE_AST Routine

```
    .
    .
    .
ROUTINE write_ast( xtcb : REF $BBLOCK ) : NOVALUE =

BEGIN
BIND
    tcc = .xtcb [xtcb$l_rflink] : $BBLOCK,
    iosb = xtcb [xtcb$w_iosb] : VECTOR [4,WORD,UNSIGNED] ;

BUILTIN
    TESTBITCS ;

LOCAL
    itcc : REF $BBLOCK,
    tcb : REF $BBLOCK,
    status,
    type ;

➊
VALIDATE_XTCC( tcc, itcc ) ;
IF .xtcb [xtcb$b_subtype] EQLU decw$c_dyn_xtcb_srp
THEN
    BEGIN
    BIND
        xtpb = .itcc [ixtcc$a_tpb] : $BBLOCK ;
```

(continued on next page)

Example 8–5 (Cont.) Sample WRITE_AST Routine

```
    status = $INSQHI( .xtcb, .itcc [ixtcc$a_ofs_queue] ) ;
    xport_out_notify_send( tcc, xtpb, decw$c_xport_buffer_srp ) ;
    END
ELSE
    BEGIN
    BIND
        xtpb = .itcc [ixtcc$a_tpb] : $BBLOCK ;

    status = $INSQHI( .xtcb, .itcc [ixtcc$a_ofl_queue] ) ;
    xport_out_notify_send( tcc, xtpb, decw$c_xport_buffer_lrp ) ;
    END ;

IF .status EQL xport$k_queue_corrupted
THEN
    BEGIN
    BIND
        xtpb = .itcc [ixtcc$a_tpb] : $BBLOCK ;

    IF TESTBITCS( tcc [xtcc$v_dying] )
    THEN
        BEGIN
        xport_abort_send( tcpip_tdb, tcc ) ;
        IF .tcc [xtcc$v_lrp_on_output]
        THEN xport_out_notify_send( tcc, xtpb, decw$c_xport_buffer_lrp )
        ELSE xport_out_notify_send( tcc, xtpb, decw$c_xport_buffer_srp ) ;
        xport_in_notify_send( tcc, xtpb ) ;
        END ;

    xport_out_write_enable( tcc ) ;
    xport_write_unwait( tcc, xtpb ) ;
    xtcc_status( tcc, DECW$_BADQUEUE ) ;
    RETURN ;
    END ;

❷ IF .tcc [xtcc$v_dying] OR NOT .iosb [0]
THEN
    BEGIN
    BIND
        xtpb = .itcc [ixtcc$a_tpb] : $BBLOCK ;

    IF TESTBITCS( tcc [xtcc$v_dying] )
    THEN
        BEGIN

        xport_abort_send( tcpip_tdb, tcc ) ;
        IF .tcc [xtcc$v_lrp_on_output]
        THEN xport_out_notify_send( tcc, xtpb, decw$c_xport_buffer_lrp )
        ELSE xport_out_notify_send( tcc, xtpb, decw$c_xport_buffer_srp ) ;
        xport_in_notify_send( tcc, xtpb ) ;
        END ;

  ❸ xport_out_write_enable( tcc ) ;
    xport_write_unwait( tcc, xtpb ) ;
    xtcc_status( tcc, .iosb [0] ) ;
    RETURN ;
    END ;
```

(continued on next page)

Example 8–5 (Cont.) Sample WRITE_AST Routine

```
❹status = $REMQHI( .itcc [ixtcc$a_ow_queue], tcb ) ;
IF .status EQL xport$k_queue_corrupted
THEN
    BEGIN
    BIND
         xtpb = .itcc [ixtcc$a_tpb] : $BBLOCK ;

    IF TESTBITCS( tcc [xtcc$v_dying] )
    THEN
        BEGIN
        xport_abort_send( tcpip_tdb, tcc ) ;
        IF .tcc [xtcc$v_lrp_on_output]
        THEN xport_out_notify_send( tcc, xtpb, decw$c_xport_buffer_lrp )
        ELSE xport_out_notify_send( tcc, xtpb, decw$c_xport_buffer_srp ) ;
        xport_in_notify_send( tcc, xtpb ) ;
        END ;

    xport_out_write_enable( tcc ) ;
    xport_write_unwait( tcc, xtpb ) ;
    xtcc_status( tcc, DECW$_BADQUEUE ) ;
    RETURN ;
    END ;

IF .status EQL xport$k_queue_no_entry
THEN
    BEGIN
    BIND
        xtpb = .itcc [ixtcc$a_tpb] : $BBLOCK ;
    xport_out_write_enable( tcc ) ;
    xport_write_unwait( tcc, xtpb ) ;
    RETURN ;
    END ;

❺tcb [xtcb$l_rflink] = tcc ;
IF NOT ( status = $qio( EFN = .itcc [ixtcc$w_efn],
                        FUNC = IO$_WRITEVBLK,
                        CHAN = .itcc [ixtcc$w_chan],
                        IOSB = tcb [xtcb$w_iosb],
                        ASTADR = write_ast,
                        ASTPRM = .tcb,
                        P1 = tcb [xtcb$t_data],
                        P2 = .tcb [xtcb$l_length] ) )
THEN
    BEGIN
    BIND
        xtpb = .itcc [ixtcc$a_tpb] : $BBLOCK ;
```

(continued on next page)

Example 8–5 (Cont.) Sample WRITE_AST Routine

```
 ❻ $INSQHI( .tcb, .itcc [ixtcc$a_ow_queue] ) ;
    IF TESTBITCS( tcc [xtcc$v_dying] )
    THEN
        BEGIN
        xport_abort_send( tcpip_tdb, tcc ) ;
        xport_in_notify_send( tcc, xtpb ) ;
        xtcc_status( tcc, .status ) ;
        END ;
    xport_write_unwait( tcc, xtpb ) ;
    END ;
END ;
    .
    .
    .
```

❶ First, assume that the write operation worked and put the buffer back on either the small or large OutputFreeQueue. The XPORT_OUT_NOTIFY_SEND macro informs the process that a write operation has completed. This operation may consist of sending a user-mode AST to the process, or of code that completes the $SYNCH system service call performed by the XPORT_OUT_WAIT macro.

If the output free queue was corrupted, perform failure processing to close this connection.

❷ In the case of TCP/IP under UCX, this procedure must detect and respond to any problem on the connection/socket. If the $QIO failed, and the connection is not yet marked as dying, then mark it. Additionally, invoke the XPORT_ABORT_SEND macro to declare a user-mode AST to the process indicating that the connection has died. Invoke the XPORT_OUT_NOTIFY_SEND and XPORT_IN_NOTIFY_SEND macros to complete any $SYNCH service used to wait for this connection.

❸ Allow write operations. The XPORT_WRITE_UNWAIT macro tests the XTCC$V_WAIT_ON_WRITE flag to see if it is set. If set, the common transport is waiting for the specific transport to empty the output work queue so that a write-user operation can be initiated. XPORT_WRITE_UNWAIT clears the XTPB$W_ON_EFN flag.

Note: **Most transports should implement the write-user function as a call to DECW$COPY_AND_WRITE and need not invoke the XPORT_WRITE_UNWAIT macro.**

Save the reason for the link abort and return.

❹ The I/O operation was successful, so attempt to get another XTCB from the output work queue. If it is empty, enable write operations on this connection and call XPORT_WRITE_UNWAIT. If the queue was corrupted, close the connection and perform abort processing.

❺ There was an XTCB on the output work queue. Initiate a write $QIO on the data.

❻ If the $QIO does not return successfully, insert the XTCB back on the output work queue. Test and set the dying flag and perform abort processing if it was clear.

8.3.5 Sample XTFT$A_WRITE_USER Routine

The XTFT$A_WRITE_USER routine attempts to write a buffer in the user's address space to a TCP/IP connection. The purpose of this interface is to avoid a data copy into XTCBs when the caller has a large, contiguous block of data to be written to a connection, such as when sending image data between client and server.

There are two methods for implementing the XTFT$A_WRITE_USER routine. The first is to wait for the output work queue to become empty and then perform the I/O operation on the user's buffer, typically by means of a $QIO.

The other method is to invoke the common transport's DECW$XPORT_COPY_AND_WRITE routine with the user's buffer as an argument. Due to the way this feature is being used in the DECwindows software, it is *strongly recommended* that transports use the DECW$XPORT_COPY_AND_WRITE routine.

Example 8–6 shows a sample implementation of the XTFT$A_WRITE_USER routine.

Example 8–6 Sample XTFT$A_WRITE_USER Routine

```
    .
    .
    .
GLOBAL ROUTINE DECW$$TCPIP_WRITE_USER( itcc : REF $BBLOCK VOLATILE, buffer :
        REF $BBLOCK, mode ) =

BEGIN
BUILTIN
    TESTBITCS,
    TESTBITCC,
    INSQTI,
    INSQHI,
    REMQHI ;

BIND
    xtcc = .itcc [ixtcc$a_tcc] : $BBLOCK,
    xtpb = .itcc [ixtcc$a_tpb] : $BBLOCK ;

LOCAL
    status,
    data_adr,
    data_len,
    size,
    lcl_iosb : VECTOR[4,WORD,UNSIGNED] ;

❶ IF .xtcc [xtcc$v_dying]
THEN
    BEGIN
    RETURN DECW$_CNXABORT ;
    END ;
```

(continued on next page)

Example 8–6 (Cont.) Sample XTFT$A_WRITE_USER Routine

```
❷ IF (data_adr = .buffer [dsc$a_pointer]) EQLA 0
      OR
     (data_len = .buffer [dsc$w_length]) EQLU 0
  THEN
      RETURN SS$_NORMAL ;

❸ %IF USER_WRITE_BY_COPY
  %THEN

❹ status = DECW$XPORT_COPY_AND_WRITE( xtcc, 0, .data_adr,
              .data_len, size ) ;

  %ELSE

❺
  (xtcc [xtcc$w_ow_iosb])< 0,32, 0> = 0 ;
  (xtcc [xtcc$w_ow_iosb])<32,32, 0> = 0 ;
  TESTBITCS( (xtcc [xtcc$l_flags])<$BITPOSITION( xtcc$v_wait_on_write ), 1> ) ;
❻ WHILE (xport_out_write_disable( xtcc )) DO
      BEGIN
      xport_write_wait( xtcc, xtpb ) ;
      END ;

❼ xport_out_write_enable( xtcc ) ;
  TESTBITCC( (xtcc [xtcc$l_flags])<$BITPOSITION( xtcc$v_wait_on_write ), 1> ) ;
❽ IF .xtcc [xtcc$v_dying]
  THEN
      BEGIN
      RETURN DECW$_CNXABORT ;
      END ;
❾ DO
      BEGIN
      size = MINU( WRITE_MAXIMUM_LENGTH, .data_len ) ;
    ❿ IF (status = $QIOW(EFN = .itcc [ixtcc$w_efn],
                           FUNC = IO$_WRITEVBLK,
                           CHAN = .itcc [ixtcc$w_chan],
                           IOSB = lcl_iosb,
                           P1 = .data_adr,
                           P2 = .size ) )
      THEN
          status = .lcl_iosb [0] ;
      IF NOT .status
      THEN
          RETURN .status ;
      data_len = .data_len - .size ;
      data_adr = .data_adr + .size ;
      END
  WHILE .data_len NEQU 0 ;

  %FI

⓫ .status
  END ;
      .
      .
      .
```

❶ If the dying field is set, return an abort status.

❷ If the address or length of the data equals zero, we are done.

❸ In the code developed by Digital, the BLISS literal USER_WRITE_BY_COPY is always set to the value 1 and the code that invokes the DECW$XPORT_COPY_AND_WRITE procedure is compiled. For completeness, the uncompiled code is also described.

❹ Invoke the common transport DECW$XPORT_COPY_AND_WRITE routine to copy the data in the user's buffer into XTCBs and make them available for transmission to the connection.

❺ Clear the waiting-for-output IOSB field of the XTCC and set the XTCC$V_WAIT_ON_WRITE bit to indicate that it is waiting for the output work queue to become empty. This operation is similar to the operation performed by the XPORT_IN_NOTIFY_SET and XPORT_OUT_NOTIFY_SET macros.

❻ Begin waiting for the output work queue to empty. The wait is required to ensure that the data sent to the connection is not reordered. When the output work queue is emptied, writing is enabled on the connection, hence the WHILE loop. Invoke the XPORT_WRITE_WAIT macro inside the loop to perform the $SYNCH service.

❼ The wait has been satisfied, so continue processing the write-user request. The XPORT_OUT_WRITE_DISABLE macro tests and sets the disable flag, so enable write operations (this is a branch-on-bit-clear-and-clear-interlocked (BBCCI) instruction). Clear the XTCC$V_WAIT_ON_WRITE bit.

❽ Check if the connection is dying. Wakeup may be due to connection abort.

❾ Begin a loop that writes the user's data. For very large user data blocks, it may be necessary to break the data into smaller pieces for transmission. The optimal size of these pieces may be different for each type of transport. In the example, this size is given by the literal WRITE_MAXIMUM_LENGTH, which is a value that can be represented by a 15-bit integer.

❿ Write the data to the connection using the $QIOW system service. The wait form of the service is required because the user's buffer is not safely copied until the I/O request completes.

⓫ Return the status.

8.3.6 Sample XTFT$A_EXECUTE_FREE Routine

The XTFT$A_EXECUTE_FREE routine logically returns an XTCB to a local logical link. The common transport invokes XTFT$A_EXECUTE_FREE after it returns an input XTCB to a previously empty free queue. XTFT$A_EXECUTE_FREE checks to see if input-free operations are enabled for this connection, and if so, attempts to start a read operation into an XTCB after removing it from the head of the queue.

The **tcb** argument is a placeholder; it is never referred to by the routine and its contents are unpredictable.

Example 8–7 shows a sample implementation of the XTFT$A_EXECUTE_FREE routine.

Example 8–7 Sample XTFT$A_EXECUTE_FREE Routine

```
   .
   .
   .
ROUTINE DECW$$TCPIP_EXECUTE_FREE(
 tcc:  REF $BBLOCK,
 tcb:  REF $BBLOCK,
 type,
 free_queue
) =

BEGIN
BUILTIN
    REMQHI ;
LOCAL
    newtcb : REF $BBLOCK,
    status ;

❶ IF NOT xport_in_free_disable( tcc, .type )
THEN
     BEGIN
   ❷ WHILE (status = REMQHI( .free_queue, newtcb )) EQL xport$k_queue_locked DO
         WHILE ..free_queue DO ;
   ❸ IF .status EQL xport$k_queue_no_entry
     THEN
         BEGIN
         xport_in_free_enable( tcc, .type ) ;
         status = SS$_NORMAL ;
         END
   ❹ ELSE
         RETURN DECW$$XPORT_FREE_INPUT( .tcc, .newtcb ) ;
     END ;

SS$_NORMAL
END ;
   .
   .
   .
```

❶ If free-input operations are disabled, leave them disabled and return a successful status. No additional work is necessary.

❷ Free-input operations that were enabled are now disabled. Attempt to remove an XTCB from the appropriate free queue. If the queue is locked, test the interlock bit of the free queue until it is no longer set, then go back and do the remove operation again.

❸ If the queue was empty this was a false start. Enable free-input operations and return a successful status.

❹ Otherwise, invoke XTFT$A_FREE_INPUT_BUFFER by means of the common transport DECW$$XPORT_FREE_INPUT routine to start a read operation on the XTCB removed from the queue.

8.3.7 Sample XTFT$A_FREE_INPUT_BUFFER Routine

The XTFT$A_FREE_INPUT_BUFFER does an asynchronous read, by means of a $QIO read, for a connection into the provided buffer. The common transport invokes XTFT$A_FREE_INPUT_BUFFER when an input XTCB has been returned by the caller to a transport and the specific transport needs to receive control in order to initiate a read operation.

Unlike the XTFT$A_EXECUTE_FREE routine, the **xtcb** argument is a "real" XTCB. It is assumed that any enable/disable checks, performed with the XPORT_IN_FREE_DISABLE macro, have already been performed.

Example 8–8 shows a sample implementation of the XTFT$A_FREE_INPUT_BUFFER routine.

Example 8–8 Sample XTFT$A_FREE_INPUT_BUFFER Routine

```
     .
     .
     .
  GLOBAL ROUTINE DECW$$TCPIP_FREE_INPUT_BUFFER( itcc : REF $BBLOCK VOLATILE, tcb
  : REF $BBLOCK ) =

  BEGIN
  BIND
      xtcb = .tcb : $BBLOCK,
      xtcc = .itcc [ixtcc$a_tcc] : $BBLOCK,
      xtpb = .itcc [ixtcc$a_tpb] : $BBLOCK ;

  LOCAL
      status,
      size,
      free_queue ;

❶ IF .xtcb [xtcb$b_subtype] EQLU decw$c_dyn_xtcb_srp
  THEN
      free_queue = .itcc [ixtcc$a_ifs_queue]
  ELSE
      free_queue = .itcc [ixtcc$a_ifl_queue] ;

❷ xtcb [xtcb$l_rflink]     = xtcc ;
  IF NOT (status = $QIO( EFN = .itcc [ixtcc$w_efn],
                         CHAN = .itcc [ixtcc$w_chan],
                         FUNC = IO$_READVBLK,
                         IOSB = xtcb [xtcb$w_iosb],
                         ASTADR = free_input_ast,
                         ASTPRM = xtcb,
                         P1 = xtcb [xtcb$t_data],
                         P2 = xport_xtcb_total( xtcb ) ) )
❸ THEN
      BEGIN
      $INSQHI( xtcb, .free_queue ) ;
      xtcc [xtcc$v_dying] = 1 ;
      xtcc_status( xtcc, .status ) ;
      END ;
  .status
  END ;
     .
     .
     .
```

❶ Determine which free queue this XTCB came from, based on the XTCB$B_SUBTYPE field of the XTCB.

❷ Store the address of the connection structure in the XTCB and then initiate an asynchronous read $QIO into the XTCB. The FREE_INPUT_AST read-completion AST routine is called with the address of the XTCB when the I/O operation completes. The FREE_INPUT_AST routine is described in Section 8.3.8.

❸ If the $QIO service failed, perform failure recovery. Return the XTCB to the correct free queue, mark the connection as dying, and store the failure status code in the XTCC with the XTCC_STATUS macro.

In either case, return the result of the $QIO service as the return value.

8.3.8 Sample FREE_INPUT_AST Routine

The FREE_INPUT_AST routine is the $QIO read-completion AST routine. It performs a number of steps based on the type of transport being used. Example 8–9 shows a sample implementation of the FREE_INPUT_AST routine.

Example 8–9 Sample FREE_INPUT_AST Routine

```
    .
    .
    .
ROUTINE free_input_ast( xtcb : REF $BBLOCK ) : NOVALUE =

BEGIN
BUILTIN
    TESTBITCS ;

BIND
   tcc = .xtcb [xtcb$l_rflink] : $BBLOCK ;

LOCAL
    itcc : REF $BBLOCK,
    tpb : REF $BBLOCK,
    status,
    tcb : REF $BBLOCK,
    type,
    free_queue ;


VALIDATE_XTCC( tcc, itcc ) ;
tpb = .itcc [ixtcc$a_tpb] ;

❶ status = $INSQTI( .xtcb, .itcc [ixtcc$a_iw_queue] ) ;
❷ IF .status EQL xport$k_queue_corrupted
THEN
    BEGIN
    BIND
        xtpb = .itcc [ixtcc$a_tpb] : $BBLOCK;
```

(continued on next page)

Example 8–9 (Cont.) Sample FREE_INPUT_AST Routine

```
  IF TESTBITCS( tcc [xtcc$v_dying] )
  THEN
     BEGIN
     xport_abort_send( tcpip_tdb, tcc ) ;
     IF .tcc [xtcc$v_lrp_on_output]
     THEN xport_out_notify_send( tcc, xtpb, decw$c_xport_buffer_lrp )
     ELSE xport_out_notify_send( tcc, xtpb, decw$c_xport_buffer_srp ) ;
     xport_in_notify_send( tcc, xtpb ) ;
     END ;

❸ xport_out_write_enable( tcc ) ;
  xport_write_unwait( tcc, xtpb ) ;
  xtcc_status( tcc, DECW$_BADQUEUE ) ;
  RETURN ;
  END ;

  BEGIN
  BIND
      xtpb = .itcc [ixtcc$a_tpb] : $BBLOCK,
      iosb = xtcb [xtcb$w_iosb] : VECTOR [4,WORD,UNSIGNED] ;

❹ IF .tcc [xtcc$v_dying] OR NOT .iosb [0]
  THEN
      BEGIN
      IF TESTBITCS( tcc [xtcc$v_dying] )
      THEN
         BEGIN
         xport_abort_send( tcpip_tdb, tcc ) ;
         IF .tcc [xtcc$v_lrp_on_output]
         THEN xport_out_notify_send( tcc, xtpb, decw$c_xport_buffer_lrp )
         ELSE xport_out_notify_send( tcc, xtpb, decw$c_xport_buffer_srp ) ;
         xport_in_notify_send( tcc, xtpb ) ;
         END ;

      xport_out_write_enable( tcc ) ;
      xport_write_unwait( tcc, xtpb ) ;
      xtcc_status( tcc, .iosb [0] ) ;
      RETURN ;
      END ;
```

(continued on next page)

Example 8–9 (Cont.) Sample FREE_INPUT_AST Routine

```
❺ xtcb [xtcb$l_length] = .iosb [1] ;
  xtcb [xtcb$a_pointer] = xtcb [xtcb$t_data] + .iosb [1] ;
❻ IF .iosb [1] LSSU .tpb [xtpb$w_srp_size]
  THEN
      BEGIN
      IF ( xport_in_state_srp( tcc ) )
      THEN
         BEGIN
         xport_in_free_disable( tcc, decw$c_xport_buffer_lrp ) ;
         END ;
      type = decw$c_xport_buffer_srp ;
      free_queue = .itcc [ixtcc$a_ifs_queue] ;
      END
  ELSE
      BEGIN
      IF ( xport_in_state_lrp( tcc ) )
      THEN
          BEGIN
          xport_in_free_disable( tcc, decw$c_xport_buffer_srp ) ;
          END ;
      type = decw$c_xport_buffer_lrp ;
      free_queue = .itcc [ixtcc$a_ifl_queue] ;
      END ;

❼ status = $REMQHI( .free_queue, tcb )  ;

  IF .status EQL xport$k_queue_corrupted
  THEN
     BEGIN
     BIND
          xtpb = .itcc [ixtcc$a_tpb] : $BBLOCK ;

     IF TESTBITCS( tcc [xtcc$v_dying] )
     THEN
         BEGIN
         xport_abort_send( tcpip_tdb, tcc ) ;
         IF .tcc [xtcc$v_lrp_on_output]
         THEN xport_out_notify_send( tcc, xtpb, decw$c_xport_buffer_lrp )
         ELSE xport_out_notify_send( tcc, xtpb, decw$c_xport_buffer_srp ) ;
         xport_in_notify_send( tcc, xtpb ) ;
         END ;

     xport_out_write_enable( tcc ) ;
     xport_write_unwait( tcc, xtpb ) ;
     xtcc_status( tcc, DECW$_BADQUEUE ) ;
     RETURN ;

     END ;
```

(continued on next page)

Example 8–9 (Cont.) Sample FREE_INPUT_AST Routine

```
    IF .status NEQ xport$k_queue_no_entry
    THEN
        BEGIN
      ❽ tcb [xtcb$l_rflink]    = tcc ;
      ❾ IF NOT status = $QIO( EFN = .itcc [ixtcc$w_efn],
                           CHAN = .itcc [ixtcc$w_chan],
                           FUNC = IO$_READVBLK,
                           IOSB = tcb [xtcb$w_iosb],
                           ASTADR = free_input_ast,
                           ASTPRM = .tcb,
                           P1 = tcb [xtcb$t_data],
                           P2 = xport_xtcb_total( tcb ) )
      ❿ THEN
           $INSQHI( .tcb, .free_queue ) ;
        END
      ⓫ ELSE
            BEGIN
            IF .type EQLU decw$c_xport_buffer_srp
            THEN xport_in_free_enable( tcc, decw$c_xport_buffer_srp )
            ELSE xport_in_free_enable( tcc, decw$c_xport_buffer_lrp ) ;
            status = SS$_NORMAL ;
            END ;
        END ;
⓬ IF NOT .status
THEN
    BEGIN
    BIND
        xtpb = .itcc [ixtcc$a_tpb] : $BBLOCK ;

    IF TESTBITCS( tcc [xtcc$v_dying] )
    THEN
        BEGIN
        xport_abort_send( tcpip_tdb, tcc ) ;
        xport_in_notify_send( tcc, xtpb ) ;
        xtcc_status( tcc, .status ) ;
        END ;
    END ;
xport_in_notify_send( tcc, tpb ) ;
END ;
    .
    .
    .
```

❶ Insert the XTCB returned by the read operation on the tail of the input work queue.

❷ If the queue was corrupted, this connection must be run down. Abort notification must be performed if this is the first attempt at rundown. This consists of conditionally declaring a user-mode AST for the link abort (by means of XPORT_ABORT_SEND), sending notification that a write operation has completed (to complete any output wait condition), and sending notification that a read operation has completed (to complete any input wait condition).

❸ Failure processing continues by enabling write operations, clearing any wait-for-empty-output-work-queue condition, saving the reason for the connection abort, and returning.

❹ Check the result of the read $QIO. If it failed, perform failure processing as done in the case of the corrupted queue.

❺ The read operation completed successfully, so initialize the XTCB$L_LENGTH and XTCB$A_POINTER fields of the completed XTCB.

❻ Determine whether the large or small XTCBs should be used in subsequent I/O operations. If the read operation that just completed filled a small XTCB, then shift up to the large XTCBs. Otherwise, if large XTCBs were being used and the last read operation would have fit in a small XTCB, then shift down to the small XTCBs. Otherwise, no change. At the end of this decision making, **type** contains the type of XTCB to be used and **free_queue** is the address of the free queue header from which these XTCBs are removed.

❼ Attempt to remove an XTCB from the free queue. If the queue was corrupted, perform failure processing.

❽ An XTCB was successfully removed from the queue. Save the connection structure address in the XTCB for use on reentry to this procedure.

❾ Initiate an asynchronous read $QIO on this XTCB, specifying this routine as the read-completion AST routine.

❿ The I/O request failed, so return the XTCB to the free queue.

⓫ There were no XTCBs on the free queue, so no more work can be done here. Enable free-input operations on the particular free queue being used and set status to normal completion.

⓬ The $QIO system service failed. Mark the connection as dying and, if this is the first time that it has been marked, perform connection-abort processing. Send input notification to complete any wait operation on the process.

8.3.9 Sample XTFT$A_CLOSE Routine

The XTFT$A_CLOSE routine initiates a series of operations that disconnect a connection and release the data structures associated with the link. Example 8–10 shows a sample implementation of the XTFT$A_CLOSE routine.

Example 8–10 Sample XTFT$A_CLOSE Routine

```
    .
    .
    .
GLOBAL ROUTINE DECW$$TCPIP_CLOSE( itcc : REF $BBLOCK VOLATILE) =

BEGIN

LOCAL
    tcc : REF $BBLOCK INITIAL( .itcc [ixtcc$a_tcc] ),
    status ;

❶tcc [xtcc$v_dying] = 1 ;

❷$CANCEL(           CHAN = .itcc [ixtcc$w_chan] ) ;
 $DASSGN(           CHAN = .itcc [ixtcc$w_chan] ) ;
 itcc [ixtcc$w_chan]  = 0 ;

❸status = $DCLAST(  ASTADR = close_and_deallocate_ast,
                    ASTPRM = .itcc ) ;
.status
END ;
    .
    .
    .
```

❶ Mark the connection as dying, both to prevent the caller from requesting operations on this connection and to prevent the various completion AST routines from attempting to perform additional work.

❷ Cancel I/O and deassign the channel to the connection. This action completes all outstanding I/O operations to the connection and queues all completion ASTs. Further references to the channel are not permitted.

❸ Declare an AST to the CLOSE_AND_DEALLOCATE_AST routine that is executed after the completion ASTs. This routine performs the final cleanup operations such as structure invalidation and deallocation.

8.3.10 Sample CLOSE_AND_DEALLOCATE_AST Routine

The CLOSE_AND_DEALLOCATE_AST routine completes the connection close initiated by XTFT$A_CLOSE. Once this procedure executes, it is assumed that neither the transport caller nor any part of the transport will refer to this connection again. It is also assumed that *all* XTCBs have been returned to the communication queue structure (XTCQ).

Example 8–11 shows a sample implementation of the CLOSE_AND_DEALLOCATE_AST routine.

Example 8–11 Sample CLOSE_AND_DEALLOCATE_AST Routine

```
    .
    .
    .
ROUTINE close_and_deallocate_ast( itcc : REF $BBLOCK ) : NOVALUE =

BEGIN
BUILTIN
    REMQUE ;

LOCAL
    tdb : REF $BBLOCK INITIAL( .itcc [ixtcc$a_tdb] ),
    tpb : REF $BBLOCK INITIAL( .itcc [ixtcc$a_tpb] ),
    status ;

❶ REMQUE( .itcc, itcc ) ;
tdb [xtdb$l_ref_count] = .tdb [xtdb$l_ref_count] - 1 ;
❷ DECW$XPORT_DEALLOC_QUEUES( .itcc ) ;

❸ itcc [ixtcc$a_xport_table] = 0 ;

❹ DECW$XPORT_DEALLOC_PMEM( .itcc ) ;
DECW$XPORT_DEALLOC_PMEM( .tpb ) ;
END ;
    .
    .
    .
```

❶ Remove the IXTCC from the IXTCC queue in the XTDB and decrement the reference count in the XTDB.

❷ Deallocate the storage previously allocated for the connection queues.

❸ Zero the IXTCC$A_XPORT_TABLE field in the IXTCC to catch any subsequent references to the connection.

❹ Deallocate the IXTCC and XTPB connection structures. At this point, the connection is completely run down.

8.3.11 Sample XTFT$A_OPEN Routine

The XTFT$A_OPEN routine is invoked by a client to establish a connection to an X server.

Example 8–12 shows a sample implementation of the XTFT$A_OPEN routine.

Example 8–12 Sample DECW$$TCPIP_OPEN Routine

```
    .
    .
    .
GLOBAL ROUTINE DECW$$TCPIP_OPEN( workstation : REF $BBLOCK, server, itcc : REF
$BBLOCK ) =

BEGIN
BUILTIN
    REMQUE,
    INSQUE ;

BIND
    tpb = .itcc [ixtcc$a_tpb] : $BBLOCK ;

LOCAL
    socktype : INITIAL( (UCX$C_STREAM ^ 16) + UCX$C_TCP ),
    sockaddrin : $BBLOCK [SIN$S_SOCKADDRIN] PRESET(
        [SIN$W_FAMILY] = INET$C_AF_INET,
        [SIN$W_PORT] = 0,
        [SIN$L_ADDR]  = swap_long( INET$C_INADDR_ANY ) ),
    sin_desc : VECTOR [2] INITIAL( %ALLOCATION( sockaddrin ), sockaddrin ),
    server_addr : VECTOR [16,BYTE,UNSIGNED],
    server_desc : VECTOR [2] INITIAL( %ALLOCATION( server_addr ) - 1, server_addr ),
    server_len : INITIAL( 0 ),
    stradr : REF VECTOR [,BYTE,UNSIGNED],
    sinadr : REF VECTOR [,BYTE,UNSIGNED],
    strlen,
    iosb : VECTOR [4,WORD,UNSIGNED],
    func_code : INITIAL( INETACP_FUNC$C_GETHOSTBYNAME ),
    func_code_desc : VECTOR [2] INITIAL( %ALLOCATION( func_code ), func_code ),
    status,
    success : INITIAL ( 0 ),
    tcb,
    tcc : REF $BBLOCK,
    free_queue ;

LABEL
    connect ;

❶ connect:
BEGIN
❷ status = DECW$XPORT_ALLOC_INIT_QUEUES( .itcc,
    .tcpip_tft[xtft$l_xtcc_length],
    .tpb [xtpb$w_srp_size],
    .tpb [xtpb$w_lrp_size],
    .tpb [xtpb$w_i_srp_count],
    .tpb [xtpb$w_i_lrp_count],
    .tpb [xtpb$w_o_srp_count],
    .tpb [xtpb$w_o_lrp_count],
    0,
    0) ;

IF NOT .status
THEN
    RETURN .status ;
tcc = .itcc[ixtcc$a_tcc] ;

❸ INSQUE( .itcc, tcpip_tdb [xtdb$a_itcc_flink] ) ;
tcpip_tdb [xtdb$l_ref_count] = .tcpip_tdb [xtdb$l_ref_count] + 1 ;
```

(continued on next page)

Example 8–12 (Cont.) Sample DECW$$TCPIP_OPEN Routine

```
❹  IF NOT (status = $assign(  DEVNAM = inet_dev_desc,
                              CHAN = itcc [ixtcc$w_chan],
                              ACMODE = psl$c_user ) )
   THEN
      LEAVE connect ;

❺ sockaddrin [SIN$W_PORT] = 0 ;
  IF (status = $qiow( EFN = .tcpip_tdb [xtdb$w_efn],
                      CHAN = .itcc [ixtcc$w_chan],
                      FUNC = IO$_SETMODE,
                      IOSB = iosb,
                      P1 = socktype,
                      P2 = ( %X'01000000' OR INET$M_LINGER ),
                      P3 = sin_desc ) )
  THEN
      status = .iosb [0] ;
  IF NOT .status
  THEN
      LEAVE connect ;

❻ IF .workstation [DSC$W_LENGTH] EQL 1
        AND .(.workstation [DSC$A_POINTER])<0,8,0> EQL %C'0'
  THEN
      workstation = lnn_desc ;

❼ IF (status = $qiow( EFN = .tcpip_tdb [xtdb$w_efn],
                          CHAN = .itcc [ixtcc$w_chan],
                          FUNC = IO$_ACPCONTROL,
                          IOSB = iosb,
                          P1 = func_code_desc,
                          P2 = workstation [0,0,0,0],
                          P3 = server_len,
                          P4 = server_desc ) )
❽ THEN
      status = .iosb [0] ;
  IF NOT .status
  THEN
      IF .status NEQU SS$_ENDOFFILE
      THEN
          LEAVE connect ;
    ❾ ELSE
          BEGIN
          IF .workstation [DSC$W_LENGTH] GEQU %ALLOCATION( server_addr )
          THEN
               BEGIN
               status = DECW$_INVSRVNAM ;
               LEAVE connect ;
               END ;
          CH$MOVE( .workstation [DSC$W_LENGTH], .workstation [DSC$A_POINTER],
                   server_addr ) ;
          server_len = .workstation [DSC$W_LENGTH] ;
          END ;
```

(continued on next page)

Example 8–12 (Cont.) Sample DECW$$TCPIP_OPEN Routine

```
❿ server_addr [.server_len] = %C'.' ;
  server_desc [0] = .server_len + 1 ;
  sinadr = sockaddrin [SIN$L_ADDR] ;
  sockaddrin [SIN$W_PORT] = swap_short( ( BASE_TCP_PORT + .server ) ) ;
  INCR i FROM 0 TO 3
⓫ DO
      BEGIN
      sinadr [.i] = 0;
      stradr = .server_desc [1] ;
      IF CH$FAIL( strlen = CH$FIND_CH( .server_desc [0], .server_desc [1], %C'.' ) )
      THEN
          BEGIN
          status = SS$_BADPARAM ;
          LEAVE connect ;
          END ;
      strlen = .strlen - .stradr ;
      INCR j FROM 0 TO ( .strlen - 1 )
      DO
          BEGIN
          IF .stradr [.j] LSSU %C'0' OR .stradr [.j] GTRU %C'9'
          THEN
              BEGIN
              status = DECW$_INVSRVNAM ;
              LEAVE connect ;
              END ;
          sinadr [.i] = .sinadr [.i] * 10 + ( .stradr [.j] - %C'0' ) ;
          END ;
      server_desc [1] = .server_desc [1] + .strlen + 1 ;
      server_desc [0] = ( .server_desc [0] - .strlen ) - 1 ;
      IF ( .server_desc [0] LSS 0 )
      THEN
          BEGIN
          status = SS$_BADPARAM ;
          LEAVE connect ;
          END ;
      END ;

⓬ IF (status = $qiow( EFN = .tcpip_tdb [xtdb$w_efn],
                       CHAN = .itcc [ixtcc$w_chan],
                       FUNC = IO$_ACCESS,
                       IOSB = iosb,
                       P3 = sin_desc ) )
  THEN
      status = .iosb [0] ;
  IF NOT .status
  THEN
      LEAVE connect ;

⓭ tcc [xtcc$l_flags]           = xtcc$m_active ;
  tcc [xtcc$v_mode]            = DECW$K_XPORT_REMOTE_CLIENT ;
  itcc [ixtcc$a_tdb]           = .tcpip_tdb ;
  itcc [ixtcc$w_efn]           = .tcpip_tdb [xtdb$w_efn] ;
  itcc [ixtcc$a_xport_table]   = .tcpip_tdb [xtdb$a_xport_table] ;

⓮ xport_in_state_srp( tcc ) ;
  xport_out_state_srp( tcc ) ;
  xport_in_free_disable( tcc, decw$c_xport_buffer_lrp ) ;
```

(continued on next page)

Example 8–12 (Cont.) Sample DECW$$TCPIP_OPEN Routine

```
⓯status = $REMQTI( .itcc [ixtcc$a_ifs_queue], tcb ) ;
  IF (.status EQL xport$k_queue_corrupted) OR
     (.status EQL xport$k_queue_no_entry)
  THEN
      BEGIN
      status = DECW$_BADQUEUE ;
      LEAVE connect ;
      END ;

⓰xport_in_free_disable( tcc, decw$c_xport_buffer_srp ) ;
  status = DECW$$TCPIP_FREE_INPUT_BUFFER( .itcc, .tcb ) ;
  RETURN .status ;
  END;

⓱IF .itcc [ixtcc$w_chan] NEQU 0
  THEN
      $DASSGN( CHAN = .itcc [ixtcc$w_chan] ) ;
  REMQUE( .itcc, itcc ) ;
  tcpip_tdb [xtdb$l_ref_count] = .tcpip_tdb [xtdb$l_ref_count] - 1 ;
  RETURN .status ;
  END ;
  .
  .
  .
```

❶ Start a named block of code that attempts to allocate and initialize the resources needed to maintain a connection. If any part of this setup fails, the code block exits and failure processing recovers any allocated resources.

❷ Call the transport-common DECW$XPORT_ALLOC_INIT_QUEUES routine to allocate and initialize the communication queues. DECW$XPORT_ALLOC_INIT_QUEUES allocates a block of storage for an XTCC, XTCQ, and all of the XTCBs for a connection. DECW$XPORT_ALLOC_INIT_QUEUES must allocate at least an XTCC; the other structures are optional. DECW$XPORT_ALLOC_INIT_QUEUES places all of the XTCBs on the appropriate free queues.

❸ Insert the IXTCC on the XTDB's queue of active connections so that the structure can be found if the rundown routine is invoked before the connection is fully started. Increment the XTDB$L_REF_COUNT field that tracks the number of connections using this transport.

❹ Assign a channel and create a socket to the Internet networking service.

❺ Initialize the sockaddrin structure to request an Internet-protocol socket on any available port. Perform a SETMODE $QIO system service to establish the desired characteristics on the socket.

The **P1** argument specifies a stream-mode, TCP/IP socket.

The **P2** argument enables the "linger" option on the TCP/IP socket.

The **P3** argument provides port, address, and address-family information for the socket.

❻ If the transport caller requested a connection with a node-identifying string of “0”, use the Internet name of the host as the node string.

❼ Attempt to perform a name-to-address conversion with an IO$_ACPCONTROL $QIO system service that queries the UCX host database. The IO$_ACPCONTROL arguments are as follows:

- **P1** specifies the function to be performed by the ACPCONTROL $QIO, which in this case requests a get-host-by-name conversion.
- **P2** is the address of a descriptor of the host name to search for in the host database.
- **P3** is the address of a word to receive the length of the returned address string.
- **P4** is the address of the descriptor of the storage to receive the address found by the search.

❽ If the $QIO failed and the reason was other than SS$_ENDOFFILE, cease processing.

❾ If the $QIO failed and the reason was SS$_ENDOFFILE, then the conversion request could not find the host name in the UCX host database. In this case, assume that the caller supplied the name of the node in Internet Standard Format (for example, 130.180.40.44).

This is also the format of the result string returned by the GETHOSTBYNAME ACPCONTROL function, so continue as if the $QIO was successful and use the caller’s node argument as the result of the ACPCONTROL $QIO system service.

❿ Determine which TCP port to attach to on the server side and place this port number, in Network Standard Format, in the sockaddrin structure. The port number is the result of adding the server number to 5000. (5000 is used in this example to prevent collision with a “real” TCP/IP transport. In a “real” transport, this number would be 6000.)

⓫ Attempt to convert the ASCII Internet Standard Format address into the binary, 32-bit Network Standard Format address. This entails converting numeric fields separated by periods into 1-byte values and then packing these bytes into a longword with the first-encountered field converted and inserted into the rightmost byte of the longword. For example, the address 130.180.40.44 would be converted to the longword value 2C28B482$_{16}$.

If there are too few fields or any nonnumeric characters seen (other than the field separator), then fail and indicate a bad parameter or invalid server name.

If successful, the Network Standard Format address is built in the sockaddrin structure.

⓬ Address conversion was successful, so attempt to attach to the server. This is done by invoking the $QIOW system service with an IO$_ACCESS function code. The **P3** argument is the address of a descriptor of the sockaddrin structure that contains the address and port number of the server.

⓭ Mark the connection as active by setting the flags longword of the XTCC to the constant XTCC$M_ACTIVE and the connection mode as a remote client. These two operations should be performed in this order because the mode field is a subfield of the flags longword.

Initialize the pointers to the XTDB in the IXTCC and the pointer to the transport function table in the IXTCC. Inherit the event flag for I/O operations from the XTDB.

⓮ Force the input and output operations to use SRPs for startup and disable free-input operations on the input LRP free queue.

⓯ Remove an XTCB from the tail of the small input free queue. If the queue was empty or corrupted, return a bad queue status and leave the code path.

⓰ Disable free-input operations on the input SRP free queue. Call DECW$XPORT_FREE_INPUT_BUFFER to start the first read operation on the connection, and return with the status. This is the successful end of the CONNECT code path.

⓱ This code is entered as a result of leaving the CONNECT code path due to some problem. Deassign the channel, remove the IXTCC from the IXTCC queue header, and decrement the XTDB$L_REF_COUNT field that tracks the number of connections using this transport.

The allocated memory will be deallocated by DECW$XPORT_OPEN.

8.3.12 Sample XTFT$A_ATTACH_TRANSPORT Routine

The XTFT$A_ATTACH_TRANSPORT routine acts as an initialization procedure to perform any client- or server-specific operations required prior to establishing connections.

Example 8–13 shows a sample implementation of the XTFT$A_ATTACH_TRANSPORT routine.

Example 8–13 Sample XTFT$A_ATTACH_TRANSPORT Routine

```
    .
    .
    .
GLOBAL ROUTINE DECW$$TCPIP_ATTACH_TRANSPORT(   tdb : REF $BBLOCK ) =

BEGIN
BIND
    iosb = tdb [xtdb$w_iosb] : VECTOR [4,WORD],
    xtpb = .tdb [xtdb$a_tpb] : $BBLOCK ;

LABEL
    attach ;

OWN
    socktype : INITIAL( (UCX$C_STREAM ^ 16) + UCX$C_TCP ),
    sockaddrin : $BBLOCK [SIN$S_SOCKADDRIN] PRESET(
        [SIN$W_FAMILY] = INET$C_AF_INET,
        [SIN$W_PORT]   = 0,
        [SIN$L_ADDR]   = swap_long( INET$C_INADDR_ANY ) ) ;

LOCAL
    sin_desc : VECTOR [2] INITIAL( SIN$S_SOCKADDRIN, sockaddrin ),
    log_desc : $BBLOCK [DSC$S_DSCDEF1],
    tab_desc : $BBLOCK [DSC$S_DSCDEF1],
    items : BLOCKVECTOR [2, ITM$S_ITEM, 1],
    host_addr : VECTOR [16,BYTE,UNSIGNED],
    host_desc : VECTOR [2] INITIAL( %ALLOCATION( host_addr ) - 1, host_addr ),
    host_len : INITIAL( 0 ),
    stradr : REF VECTOR [,BYTE,UNSIGNED],
    sinadr : REF VECTOR [,BYTE,UNSIGNED],
    strlen,
    func_code : INITIAL( INETACP_FUNC$C_GETHOSTBYNAME ),
    func_code_desc : VECTOR [2] INITIAL( %ALLOCATION( func_code ), func_code ),
    retlen,
    status ;

❶ inet_dev_desc [0] = %CHARCOUNT( inet_dev_str ) ;
  inet_dev_desc [1] = UPLIT( inet_dev_str ) ;

❷ tcpip_tdb = .tdb ;
  tdb [xtdb$w_efn] = ASYNC_EFN ;

❸ lnn_desc [DSC$A_POINTER] = local_node ;
  items [0,ITM$W_ITMCOD] = LNM$_STRING ;
  items [0,ITM$W_BUFSIZ] = %ALLOCATION( local_node ) ;
  items [0,ITM$L_BUFADR] = local_node ;
  items [0,ITM$L_RETLEN] = lnn_desc [DSC$W_LENGTH] ;
  items [1,ITM$W_ITMCOD] = 0 ;
  items [1,ITM$W_BUFSIZ] = 0 ;
  items [1,ITM$L_BUFADR] = 0 ;
  items [1,ITM$L_RETLEN] = 0 ;
  log_desc [DSC$W_LENGTH] = %CHARCOUNT( inet_local_node ) ;
  log_desc [DSC$B_DTYPE] = DSC$K_DTYPE_T ;
  log_desc [DSC$B_CLASS] = DSC$K_CLASS_S ;
  log_desc [DSC$A_POINTER] = UPLIT( inet_local_node ) ;
  tab_desc [DSC$W_LENGTH] = %CHARCOUNT( 'LNM$FILE_DEV' ) ;
  tab_desc [DSC$B_DTYPE] = DSC$K_DTYPE_T ;
  tab_desc [DSC$B_CLASS] = DSC$K_CLASS_S ;
  tab_desc [DSC$A_POINTER] = UPLIT( 'LNM$FILE_DEV' ) ;
```

(continued on next page)

Example 8–13 (Cont.) Sample XTFT$A_ATTACH_TRANSPORT Routine

```
status = $TRNLNM( TABNAM = tab_desc,
                  LOGNAM = log_desc,
                  ITMLST = items ) ;

❹ IF ( .tdb [xtdb$v_mode] AND DECW$M_XPORT_CLIENT ) NEQ 0
THEN
    RETURN .status ;

attach:
     BEGIN
     IF NOT .status
     THEN
         LEAVE attach;

❺ IF NOT (status = $ASSIGN(   DEVNAM = inet_dev_desc,
                            CHAN = tdb [xtdb$w_chan],
                            ACMODE = psl$c_user ) )
THEN
    LEAVE attach ;

❻ IF (status = $qiow(         EFN = .tdb [xtdb$w_efn],
                            CHAN = .tdb [xtdb$w_chan],
                            FUNC = IO$_ACPCONTROL,
                            IOSB = iosb,
                            P1 = func_code_desc,
                            P2 = lnn_desc,
                            P3 = host_len,
                            P4 = host_desc ) )
THEN
    status = .iosb [0] ;
IF NOT .status
THEN
    LEAVE attach ;

❼ host_addr [.host_len] = %C'.' ;
host_desc [0] = .host_len + 1 ;
sinadr = sockaddrin [SIN$L_ADDR] ;
INCR i FROM 0 TO 3
DO
    BEGIN
    sinadr [.i] = 0 ;
    stradr = .host_desc [1] ;
    IF CH$FAIL( strlen = CH$FIND_CH( .host_desc [0], .host_desc [1], %C'.' ) )
    THEN
       BEGIN
       status = SS$_BADPARAM ;
       LEAVE attach ;
       END ;
```

(continued on next page)

Example 8–13 (Cont.) Sample XTFT$A_ATTACH_TRANSPORT Routine

```
      strlen = .strlen - .stradr ;
      INCR j FROM 0 TO ( .strlen - 1 )
      DO
          BEGIN
          sinadr [.i] = .sinadr [.i] * 10 + ( .stradr [.j] - %C'0' ) ;
          END ;
      host_desc [1] = .host_desc [1] + .strlen + 1 ;
      host_desc [0] = ( .host_desc [0] - .strlen ) - 1 ;
      IF ( .host_desc [0] LSS 0 )
      THEN
          BEGIN
          status = SS$_BADPARAM ;
          LEAVE attach ;
          END ;
      END ;

  sockaddrin [SIN$W_PORT] = swap_short( ( BASE_TCP_PORT +
                                           .xtpb [xtpb$w_display_num] ) ) ;

❽ IF (status = $QIOW(      EFN = .tdb [xtdb$w_efn],
                           CHAN = .tdb [xtdb$w_chan],
                           FUNC = IO$_SETMODE,
                           IOSB = iosb,
                           P1 = socktype,
                           P2 = ( %X'01000000' OR INET$M_LINGER OR INET$M_KEEPALIVE ),
                           P3 = sin_desc,
                           P4 = 5 ) )
  THEN
      status = .iosb [0] ;
  IF NOT .status
  THEN
      LEAVE attach ;

❾ IF NOT ( status = transport_read_queue( .tdb ) )
  THEN
      LEAVE attach ;

❿ DECW$XPORT_ATTACHED( .tdb ) ;
  reattach_timer_id = 0 ;
  RETURN SS$_NORMAL ;
  END ;

⓫ detach_and_poll( .tdb ) ;
  RETURN .status ;
  END ;
      .
      .
      .
```

❶ Create a suitable descriptor for the Internet device logical name string. The .address directive that would result from doing this at compile time cannot be fixed up by the image activator, so the descriptor is created at run time.

❷ Store the address of the XTDB that the common transport allocated for this transport in tcpip_tdb. References to this XTDB by the TCP/IP transport are usually made through this variable. Use event flag 31 (ASYNC_EFN) in asynchronous transport operations.

❸ Get the string that represents the local system name in the Internet name space by translating the logical name "UCX$INET_HOST" that is generated by the local_node macro.

❹ If the transport is being attached by a client, no additional work is needed. A server continues processing to create a listener socket, among other things.

❺ Create a socket and assign a channel to the Internet networking service. This socket is owned by the specific transport and becomes the listener socket for all connection requests received from clients.

❻ Attempt to perform a name-to-address conversion for the local node with an IO$_ACPCONTROL $QIO system service that queries the UCX host database. The $QIO arguments are as follows:

- The **P1** argument specifies the control function that, in this case, requests a get-host-by-name conversion. P1 is the function to be performed by the IO$_ACPCONTROL $QIO.
- **P2** is the address of the descriptor of the local node name acquired at the beginning of this procedure.
- **P3** is the address of a word to receive the length of the returned address string.
- **P4** is the address of the descriptor of the storage to receive the address found by the search.

❼ Attempt to convert the ASCII Internet Standard Format address returned by the ACPCONTROL $QIO into the binary, 32-bit Network Standard Format address. (See Example 8–12.)

To do this, convert numeric fields separated by periods into 1-byte values and then pack these bytes into a longword with the first-encountered field converted and inserted into the rightmost byte of the longword. For example, the address 130.180.40.44 would be converted to the longword value $2C28B482_{16}$.

If there are too few fields, or any nonnumeric characters seen (other than the field separator), fail and indicate a bad parameter or invalid server name.

If the conversion is successful, the Network Standard Format address is built in the sockaddrin structure. Determine which TCP port number to use to listen for client connection requests. Place this port number, in Network Standard Format, in the sockaddrin structure. (The port number is the result of adding the server number to 5000. In a "real" transport, this number would be 6000.)

❽ Establish the local address, port number, address family, and socket options on the listener socket. The $QIO arguments are as follows:

- The **P1** argument of the IO$_SETMODE $QIO system service is the address of a longword containing the protocol family and type to use (stream-mode, TCP socket).
- The **P2** argument sets various options for the socket (in this case, the linger and keep-alive TCP options).

- The **P3** argument is the address of a descriptor of the sockaddrin structure that contains the address information for the socket.
- The **P5** argument is a nonzero value (specifically, 5) that marks the socket as a listener and sets the size of the connect queue. The connect queue limits the number of unacknowledged client connect requests that the Internet networking service is willing to retain.

❾ Invoke the TRANSPORT_READ_QUEUE routine to start an asynchronous socket-accept operation. At this point, the server is capable of receiving connection requests from clients.

❿ Call DECW$XPORT_ATTACHED to report that the transport is attached.

⓫ This code is entered only if something went wrong. Call the DETACH_AND_POLL routine to deassign any channels and attempt to reattach to the network.

8.3.13 Sample TRANSPORT_READ_QUEUE Routine

The TRANSPORT_READ_QUEUE routine initiates an asynchronous connect-accept operation on the listener socket. Example 8–14 shows a sample implementation of the TRANSPORT_READ_QUEUE routine.

Example 8–14 Sample TRANSPORT_READ_QUEUE Routine

```
    .
    .
    .
ROUTINE transport_read_queue( tdb : REF $BBLOCK ) =

BEGIN

LOCAL
    item3 : VECTOR [3],
    status ;

❶ IF NOT .tdb [xtdb$v_dying]
THEN
    BEGIN

  ❷ IF NOT (status = $ASSIGN(      DEVNAM = inet_dev_desc,
                                   CHAN = tdb [xtdb$w_acc_chan],
                                   ACMODE = psl$c_user ) )

    THEN
        BEGIN
        tdb [xtdb$v_dying] = 1 ;
        RETURN .status
        END ;
```

(continued on next page)

Example 8–14 (Cont.) Sample TRANSPORT_READ_QUEUE Routine

```
  ❸ item3 [0] = xtdb$s_acc_inaddr ;
    item3 [1] = tdb [xtdb$t_acc_inaddr] ;
    item3 [2] = tdb [xtdb$l_acc_inaddr_len] ;
  ❹ status = $qio(  EFN = .tdb [xtdb$w_efn],
                    CHAN = .tdb [xtdb$w_chan],
                    FUNC = IO$_ACCESS OR IO$M_ACCEPT,
                    IOSB = tdb [xtdb$w_acc_iosb],
                    ASTADR = transport_read_ast,
                    ASTPRM = .tdb,
                    P3 = item3,
                    P4 = tdb [xtdb$w_acc_chan] ) ;
    IF NOT .status
    THEN
        BEGIN
        $DASSGN(   CHAN = .tdb [xtdb$w_acc_chan] ) ;
        tdb [xtdb$v_dying] = 1 ;
        END
    END
ELSE
    status = DECW$_CNXABORT ;

RETURN .status ;

END ;
    .
    .
    .
```

❶ If the transport is dying, do not try to start another accept operation.

❷ Create a socket and assign a channel to the Internet networking service. If $ASSIGN fails, mark the transport as dying and quit. If successful, the channel number is stored in the XTDB$W_ACC_CHAN field of the XTDB, which is a field specific to this transport.

❸ Create an item-list structure describing the area that is to receive information about the client that attempts to connect to the server. The first longword is the size of the area in bytes, the second is the address of the first byte, and the third is the address of a longword to receive the length of the data actually placed in the area.

The item-list structure is allocated as a 16-byte field, XTDB$T_ACC_INADDR, in the XTDB. XTDB$S_ACC_INADDR is the symbolic name for the length.

❹ Initiate an asychronous accept operation on the listener socket by means of the $QIO system service. The $QIO arguments are as follows:

- The **CHAN** argument is the channel associated with the listener socket created in the XTFT$A_ATTACH_TRANSPORT routine. The specific transport uses this channel for connection requests from all clients.
- The **FUNC** argument code is IO$_ACCESS with the IO$M_ACCEPT modifier specified.

- The **ASTADR** argument is the AST completion routine to invoke on completion. TRANSPORT_READ_AST, which is described in Section 8.3.14, expects the address of the TCP/IP XTDB as its argument.
- The **P3** argument is the address of the 3-longword item list previously described. The item list is used to store information about the connecting client's node.
- The **P4** argument is the channel associated with the socket created on entry to TRANSPORT_READ_QUEUE. The server will use this channel for communication with the client. Each client connection is assigned its own channel.

If the $QIO service failed, mark the transport as dying and release the channel.

8.3.14 Sample TRANSPORT_READ_AST Routine

TRANSPORT_READ_AST is invoked when the IO$_ACCESS $QIO issued by the TRANSPORT_READ_QUEUE routine completes and continues processing to fully establish the client connection.

Example 8–15 shows a sample implementation of the TRANSPORT_READ_AST routine.

Example 8–15 Sample TRANSPORT_READ_AST Routine

```
    .
    .
    .
ROUTINE transport_read_ast( tdb : REF $BBLOCK ) : NOVALUE =

BEGIN
BUILTIN
    MOVPSL,
    REMQUE,
    INSQTI,
    INSQUE ;
BIND
    xtpb = .tcpip_tdb [xtdb$a_tpb] : $BBLOCK,
    acc_iosb = tcpip_tdb [xtdb$w_acc_iosb] : VECTOR [4,WORD,UNSIGNED] ;

LOCAL
    psl : $BBLOCK [4],
    found : INITIAL( 0 ),
    iosb : VECTOR [4,WORD,UNSIGNED] ;

❶ IF .acc_iosb [0] EQL SS$_SHUT
THEN
    BEGIN
    DECW$XPORT_ATTACH_LOST( .tdb , 0 ) ;
    detach_and_poll( .tdb ) ;
    RETURN ;
    END ;

IF .acc_iosb [0]
THEN
```

(continued on next page)

Example 8–15 (Cont.) Sample TRANSPORT_READ_AST Routine

```
  BEGIN
  MACRO
      ctrstr = '!UB.!UB.!UB.!UB' % ;

  LOCAL
      tcq : REF $BBLOCK INITIAL( 0 ),
      tcc : REF $BBLOCK INITIAL( 0 ),
      itcc : REF $BBLOCK INITIAL( 0 ),
      tcc_id : INITIAL( 0 ),
      tpb : REF $BBLOCK INITIAL( 0 ),
      fail : INITIAL( 1 ),
      status,
      il_count : INITIAL( .xtpb [xtpb$w_i_lrp_count] ),
      is_count : INITIAL( .xtpb [xtpb$w_i_srp_count] ),
      ol_count : INITIAL( .xtpb [xtpb$w_o_lrp_count] ),
      os_count : INITIAL( .xtpb [xtpb$w_o_srp_count] ),
      at_tcb,
      tcb_count,
      tcb_array : REF VECTOR [] INITIAL( 0 ),
      func_code : INITIAL( INETACP_FUNC$C_GETHOSTBYADDR ),
      func_code_desc : VECTOR [2] INITIAL( %ALLOCATION( func_code ),
                                            func_code ),
      inaddr : $BBLOCK [16],
      inaddr_len,
      inaddr_desc : VECTOR [2] INITIAL( %ALLOCATION( inaddr ), inaddr ),
      client_desc : VECTOR [2],
      ctr_desc : VECTOR [2] INITIAL( %CHARCOUNT( ctrstr ),
                                                     UPLIT( ctrstr ) ),
      info_size,
      info_ptr,
      client_len ;

  LABEL
      connect ;

❷ connect:    BEGIN

      info_size = INET_NODE_NAME_LEN ;
    ❸ IF (itcc = DECW$XPORT_ALLOC_PMEM( ixtcc$c_tcpip_length,
                                            DECW$C_DYN_IXTCC )) EQLA 0
      THEN
          BEGIN
          status = SS$_INSFMEM ;
          LEAVE connect ;
          END

      IF (tpb = DECW$XPORT_ALLOC_PMEM( xtpb$c_tcpip_length,
                                       DECW$C_DYN_XTPB )) EQLA 0
      THEN
          BEGIN
          status = SS$_INSFMEM ;
          LEAVE connect ;
          END

       itcc [ixtcc$a_tpb] = .tpb ;
```

(continued on next page)

Example 8–15 (Cont.) Sample TRANSPORT_READ_AST Routine

```
❹ status = DECW$XPORT_ALLOC_INIT_QUEUES( .itcc,
           .tcpip_tft[xtft$l_xtcc_length],
           .tpb [xtpb$w_srp_size],
           .tpb [xtpb$w_lrp_size],
           .tpb [xtpb$w_i_srp_count],
           .tpb [xtpb$w_i_lrp_count],
           .tpb [xtpb$w_o_srp_count],
           .tpb [xtpb$w_o_lrp_count],
           .info_size,
           info_ptr) ;

  IF NOT .status
  THEN
      LEAVE connect ;

  tcc = .itcc[ixtcc$a_tcc] ;
  inaddr_len = 0 ;
❺ IF NOT (status = $FAO(  ctr_desc,
                          inaddr_len,
                          inaddr_desc,
                          .(tdb [xtdb$t_acc_inaddr])<32,8,0>,
                          .(tdb [xtdb$t_acc_inaddr])<40,8,0>,
                          .(tdb [xtdb$t_acc_inaddr])<48,8,0>,
                          .(tdb [xtdb$t_acc_inaddr])<56,8,0> ) )
  THEN
      BEGIN
      LEAVE connect ;
      END ;

  inaddr_desc [0] = .inaddr_len ;
  client_desc [0] = INET_NODE_NAME_LEN - 1 ;
  client_desc [1] = .info_ptr + 1 ;
  client_len = 0 ;

❻ IF (status = $QIOW(    EFN = .tdb [xtdb$w_efn],
                         CHAN = .tdb [xtdb$w_chan],
                         FUNC = IO$_ACPCONTROL,
                         IOSB = iosb,
                         P1 = func_code_desc,
                         P2 = inaddr_desc,
                         P3 = client_len,
                         P4 = client_desc ) )
  THEN
      status = .iosb [0] ;
  IF NOT .status
  THEN
      BEGIN
      IF .status NEQU SS$_ENDOFFILE
      THEN
          BEGIN
          LEAVE connect ;
          END
    ❼ ELSE
          BEGIN
          CH$MOVE( .inaddr_len, .inaddr_desc [1], .client_desc [1] ) ;
          client_desc [0] = .inaddr_len ;
          client_len = .inaddr_len ;
          END ;
      END
```

(continued on next page)

Example 8–15 (Cont.) Sample TRANSPORT_READ_AST Routine

```
      ❽ ELSE
            BEGIN
            IF CH$EQL( .client_len, .client_desc [1], .lnn_desc [DSC$W_LENGTH],
                       .lnn_desc [DSC$A_POINTER], %C' ' )
            THEN
                BEGIN
                (.client_desc [1])<0,8,0> = %C'0' ;
                client_desc [0] = 1 ;
                client_len = 1 ;
                END ;
            END ;
  ❾
        (.info_ptr)<0,8,0> = %C'?' ;

      ❿ IF .info_size GTR 0
        THEN
            BEGIN
            tcc [xtcc$a_rem_user]         = .info_ptr ;
            tcc [xtcc$l_rem_user_len]     = 1 ;
            tcc [xtcc$a_rem_node]         = .info_ptr + 1 ;
            tcc [xtcc$l_rem_node_len]     = .client_len ;
            END ;

      ⓫ tcpip_tdb [xtdb$l_ref_count]   = .tcpip_tdb [xtdb$l_ref_count] + 1 ;
        INSQUE( .itcc, tcpip_tdb [xtdb$a_itcc_flink] ) ;

      ⓬ itcc [ixtcc$w_chan]           = .tcpip_tdb [xtdb$w_acc_chan] ;
        tcpip_tdb [xtdb$w_acc_chan]   = 0 ;
      ⓭ tcc [xtcc$l_flags]            = xtcc$m_active ;
        tcc [xtcc$v_mode]             = DECW$K_XPORT_REMOTE_SERVER ;
        itcc [ixtcc$w_efn]            = .tcpip_tdb [xtdb$w_efn] ;
        itcc [ixtcc$a_tdb]            = .tcpip_tdb ;
        itcc [ixtcc$a_xport_table]    = .tcpip_tdb [xtdb$a_xport_table] ;

      ⓮ IF NOT (status = $DCLAST(   ASTADR = transport_open_callback,
                                    ASTPRM = .itcc,
                                    ACMODE = psl$c_user ) )
        THEN
            LEAVE connect ;
      ⓯ fail = 0 ;
        END ;

 ⓰ IF .fail
   THEN
     ⓱ BEGIN
       IF .itcc NEQA 0
       THEN
           BEGIN
           IF .itcc [ixtcc$w_chan] NEQU 0
           THEN
               $DASSGN( CHAN = .itcc [ixtcc$w_chan] ) ;
           REMQUE( .itcc, itcc ) ;
           tcpip_tdb [xtdb$l_ref_count] = .tcpip_tdb [xtdb$l_ref_count] - 1 ;
           DECW$XPORT_DEALLOC_QUEUES( .itcc ) ;
           DECW$XPORT_DEALLOC_PMEM( .itcc ) ;
           END ;
       IF .tpb NEQA 0
       THEN
           DECW$XPORT_DEALLOC_PMEM( .tpb ) ;
```

(continued on next page)

Example 8–15 (Cont.) Sample TRANSPORT_READ_AST Routine

```
        DECW$XPORT_ACCEPT_FAILED (   .tcpip_tdb [xtdb$l_acc_inaddr_len],
                                    tcpip_tdb [xtdb$t_acc_inaddr],
                                    .status ) ;
        END ;
    END ;
⓲transport_read_queue( .tcpip_tdb ) ;
END ;
    .
    .
    .
```

❶ Test the status from the IO$_ACCESS $QIO. If the network is shutting down, call DECW$XPORT_ATTACH_LOST to report that the transport shut down and then call DETACH_AND_POLL to poll for its return. If it was successful, continue processing to construct a full connection context.

❷ Start a named block of code that attempts to allocate and initialize the resources needed to maintain a connection. If any part of this setup fails, the code block is exited and failure processing recovers any allocated resources.

❸ Allocate an IXTCC and XTPB for this connection. Store the address of the XTPB in the IXTCC.

❹ Call the transport-common DECW$XPORT_ALLOC_INIT_QUEUES routine to allocate and initialize the communication queues. DECW$XPORT_ALLOC_INIT_QUEUES allocates a block of storage for an XTCC, XTCQ, and all of the XTCBs for a connection. DECW$XPORT_ALLOC_INIT_QUEUES must allocate at least an XTCC; the other structures are optional. DECW$XPORT_ALLOC_INIT_QUEUES places all of the XTCBs on the appropriate free queues.

❺ Convert the Network Standard Format Internet address of the connection client, which is returned by the IO$_ACCESS $QIO in the XTDB$T_ACC_INADDR field of the XTDB, into a dot-format Internet address. For example, the longword value 2C28B482$_{16}$ would be converted to the ASCII string 130.180.40.44.

❻ Attempt to perform address-to-nodename translation of the dot-format Internet address to get the textual name of the client node. The arguments of the IO$_ACPCONTROL $QIO system service are as follows:

- **P1** is the address of a 2-longword descriptor of a longword containing the function code INETACP_FUNC$C_GETHOSTBYADDR.
- **P2** is the address of the descriptor of the dot-format Internet address to be translated.
- **P3** is the address of a word to receive the length of the resultant string.

- **P4** is the address of a descriptor of an area of memory to receive the result of the translation.

❼ The address-to-nodename translation failed with the status of SS$_ENDOFFILE. This indicates that the dot-address could not be found in the local host database. The dot-address form of the client's node address will be used as the node name of the client.

❽ The address-to-nodename translation succeeded. Check if the returned node name matches the local node name. If it does, use the string "0" as the client's node name; otherwise use the string returned by the translation operation.

❾ The Internet protocols do not support the concept of a remote user name. Synthesize this information by identifying the remote user with the string "?".

❿ Point the XTCC to the remote-user name and node fields.

⓫ Increment the reference count and insert the IXTCC on the XTDB so that the connection can be located during image rundown.

⓬ Copy the channel assigned to the client connection into the IXTCC and zero this field in the XTDB.

⓭ Mark the connection as active by setting the flags longword of the XTCC to the constant XTCC$M_ACTIVE and the connection mode as a remote server (these two operations should be performed in this order because the mode field is a subfield of the flags longword).

Initialize the pointer to the XTDB in the IXTCC and the pointer to the transport function table in the IXTCC. Inherit the event flag for I/O operations from the XTDB.

⓮ Deliver a user-mode AST to the TRANSPORT_OPEN_CALLBACK routine (with the IXTCC as the argument) to complete the connection acceptance in user mode.

TRANSPORT_OPEN_CALLBACK performs the callback to the transport caller and starts I/O on the connection.

⓯ Executive-mode connection setup completed.

⓰ If any step of the connection setup in the connect block failed, resource recovery operations are performed here: channels are deassigned, memory is deallocated, connection structures are disassociated, reference counts are decremented, and so on.

⓱ Call the transport common DECW$XPORT_ACCEPT_FAILED routine to generate a report describing the cause of the failure.

⓲ This is the common exit point. Invoke TRANSPORT_READ_QUEUE to issue another IO$_ACCESS $QIO to receive another client connection request.

8.3.15 Sample TRANSPORT_OPEN_CALLBACK Routine

TRANSPORT_OPEN_CALLBACK is invoked as a user-mode AST procedure by TRANSPORT_READ_AST to complete the creation of a connection. It performs a callback to the server's connect-request routine and, depending on the value returned by that routine, completes the connection by starting the initial read operation.

Example 8–16 shows a sample implementation of the TRANSPORT_OPEN_CALLBACK routine.

Example 8–16 Sample TRANSPORT_OPEN_CALLBACK Routine

```
    .
    .
    .
ROUTINE transport_open_callback( itcc : REF $BBLOCK ) : NOVALUE =
BEGIN
BUILTIN
     REMQTI ;
LOCAL
    tcc:        REF $BBLOCK INITIAL ( .itcc [ixtcc$a_tcc] ),
    free_queue: REF VECTOR[2],
    status ;

LABEL
    start_reading ;

start_reading:
        BEGIN
      ❶ IF NOT (status = (.tcpip_tdb [xtdb$a_connect_request])( .tcc ))
        THEN
            LEAVE start_reading ;
      ❷ xport_in_state_srp( tcc ) ;
        xport_out_state_srp( tcc ) ;
        xport_in_free_disable( tcc, decw$c_xport_buffer_lrp ) ;
      ❸ free_queue = .tcc [xtcc$a_ifs_queue] ;
        IF .free_queue[0] EQLA 0
        THEN
            BEGIN
            free_queue = .tcc [xtcc$a_ifl_queue] ;
            IF .free_queue[0] EQLA 0
            THEN
                BEGIN
                status = DECW$_BADQUEUE ;
                LEAVE start_reading ;
                END ;
            END ;

      ❹ status = DECW$$TCPIP_EXECUTE_FREE
                     ( .tcc, 0, decw$c_xport_buffer_srp, .free_queue ) ;
        IF .status
        THEN
            RETURN ;
        END ;

      ❺ DECW$XPORT_CLOSE( .tcc ) ;
```

(continued on next page)

Example 8–16 (Cont.) Sample TRANSPORT_OPEN_CALLBACK Routine

```
     ❻ DECW$XPORT_REFUSED_BY_SERVER ( .status ) ;
 RETURN ;
 END ;

     .
     .
     .
```

❶ Invoke the server's connect-request routine with the address of the XTCC for the connection.

❷ The transport user accepted the connection. Force the input and output operations to use small request packets for startup and disable free-input operations on the input large request packet free queue.

❸ Check the small input free queue and, if it is empty, the large input free queue. If both are empty, the transport cannot perform a read operation, so return a bad queue status.

❹ Call the XTFT$A_EXECUTE_FREE routine to put the buffers on the free queue and return.

❺ This code is entered only if the callback procedure returned a failure status. Invoke the transport-common DECW$XPORT_CLOSE routine to complete the destruction of the connection.

❻ Call DECW$XPORT_REFUSED_BY_SERVER to format and deliver a message vector to describe the reason for the connection refusal. This is constructed on the status code returned by the callback procedure.

8.3.16 Sample DETACH_AND_POLL Routine

DETACH_AND_POLL starts polling for a transport restart.

Example 8–17 shows a sample implementation of the DETACH_AND_POLL routine.

Example 8–17 Sample DETACH_AND_POLL Routine

```
    .
    .
    .
ROUTINE detach_and_poll( tdb : REF $BBLOCK ) : NOVALUE =

BEGIN
BUILTIN
    EMUL ;

LOCAL
    status ;

❶ IF .tdb [xtdb$v_dying]
THEN
    RETURN ;

❷ IF .tdb [xtdb$w_chan] NEQ 0
THEN
    BEGIN
    $CANCEL( CHAN = .tdb [xtdb$w_chan] ) ;
    $DASSGN( CHAN = .tdb [xtdb$w_chan] ) ;
    tdb [xtdb$w_chan] = 0 ;
    END ;

❸ IF .tdb [xtdb$w_acc_chan] NEQ 0
THEN
    BEGIN
    $CANCEL( CHAN = .tdb [xtdb$w_acc_chan] ) ;
    $DASSGN( CHAN = .tdb [xtdb$w_acc_chan] ) ;
    tdb [xtdb$w_acc_chan] = 0 ;
    END ;

❹ IF .reattach_timer_id EQL 0
THEN
    BEGIN
    reattach_timer_id = .tdb ;
    EMUL( %REF( REATTACH_INTERVAL_SECS ), %REF( -10000000 ), %REF( 0 ),
              reattach_timer_delta ) ;
    END ;

❺ IF .tcpip_tdb [xtdb$v_dying]
THEN
    RETURN ;

❻ status = $SETIMR(
             EFN = 31,
             DAYTIM = reattach_timer_delta,
             ASTADR = reattach_ast,
             REQIDT = .reattach_timer_id ) ;

❼ IF NOT .status
THEN
   DECW$XPORT_REATTACH_FAILED( .tdb, .status ) ;
RETURN ;
END ;
    .
    .
    .
```

❶ If the XTDB$V_DYING bit is set, there is no need to continue.

❷ Release the channel to the Internet device.

❸ Release the connection-accept channel.

❹ If the timer is not identified, associate the timer with the address of the XTDB.

❺ Make sure that the transport is still alive.

❻ Start polling for a network restart at **reattach_timer_delta** intervals. (The REATTACH_AST routine is described in Section 8.3.17.)

❼ If unable to poll for the restart, report that the reattach failed.

8.3.17 Sample REATTACH_AST Routine

REATTACH_AST calls DECW$$TCPIP_ATTACH_TRANSPORT to reattach the transport when the **reattach_timer_delta** interval has expired.

Example 8–18 shows a sample implementation of the REATTACH_AST routine.

Example 8–18 Sample REATTACH_AST Routine

```
    .
    .
    .
ROUTINE reattach_ast( tdb : REF $BBLOCK ) : NOVALUE =

BEGIN

LOCAL
    status;

❶ status = DECW$$TCPIP_ATTACH_TRANSPORT( .tdb ) ;

RETURN ;
END ;
    .
    .
    .
```

❶ Call DECW$$TCPIP_ATTACH_TRANSPORT to reattach the transport.

8.3.18 Sample XTFT$A_RUNDOWN Routine

XTFT$A_RUNDOWN is invoked by the common transport when the image in which the transport is running exits. It is the responsibility of each transport's rundown procedure to release any resources that might survive the image exit. ASTs are disabled while XTFT$A_RUNDOWN is executing.

Example 8–19 shows a sample implementation of the XTFT$A_RUNDOWN routine.

Example 8–19 Sample XTFT$A_RUNDOWN Routine

```
   .
   .
   .
GLOBAL ROUTINE DECW$$TCPIP_RUNDOWN(   tdb : REF $BBLOCK ) : NOVALUE =

BEGIN
BIND
    iosb = tdb [xtdb$w_iosb] : VECTOR [4,WORD],
    xtpb = .tdb [xtdb$a_tpb] : $BBLOCK ;

LOCAL
    itcc : REF $BBLOCK INITIAL( .tdb [xtdb$a_itcc_flink] ),
    status ;

❶tdb [xtdb$v_dying] = 1 ;

❷WHILE .itcc NEQA tdb [xtdb$a_itcc_flink] DO
    BEGIN
    BIND
       xtcc = .itcc [ixtcc$a_tcc] : $BBLOCK ;

    xtcc [xtcc$v_dying] = 1 ;
    $CANCEL(    CHAN = .itcc [ixtcc$w_chan] ) ;
    $DASSGN(    CHAN = .itcc [ixtcc$w_chan] ) ;
    itcc [ixtcc$w_chan] = 0 ;
    itcc = .itcc [xtcc$a_flink] ;
    END ;

❸IF ( .tdb [xtdb$v_mode] AND DECW$M_XPORT_CLIENT ) NEQ 0
THEN
    RETURN ;

❹$CANCEL(   CHAN = .tdb [xtdb$w_chan] ) ;
$CANCEL(   CHAN = .tdb [xtdb$w_acc_chan] ) ;

$DASSGN(   CHAN = .tdb [xtdb$w_acc_chan] ) ;
tdb [xtdb$w_acc_chan] = 0 ;
$DASSGN(   CHAN = .tdb [xtdb$w_chan] ) ;
tdb [xtdb$w_chan] = 0 ;

IF .reattach_timer_id NEQ 0
THEN
    $CANTIM( REQIDT = .reattach_timer_id ) ;

RETURN ;

END ;
   .
   .
   .
```

❶ Mark this transport as dying to prevent any of the routines, such as TRANSPORT_READ_QUEUE, from performing additional operations.

❷ Locate each connection known to this transport and perform any rundown operations necessary. For TCP/IP, the connection is marked as dying to prevent any per-connection routine from operating on the connection. Any I/O operation in progress on the connection is canceled and the communication channel is deassigned.

❸ If the caller is a client, no more processing is needed for rundown.

❹ For servers, cancel any I/O operation on the listening and accepting channels and deassign the channels.

8.3.19 Sample DECW$TRANSPORT_INIT Routine

The DECW$TRANSPORT_INIT routine initializes and returns the address of the XTFT. Each specific transport has a global symbol named DECW$TRANSPORT_INIT. This is the address of the first procedure invoked in the specific transport during the common transport DECW$XPORT_ATTACH_TRANSPORT routine and it is responsible for initializing the fields in the XTFT structure.

The DECW$EXAMPLES:XPORT_EXAMPLE_XFER.MAR module and DECW$EXAMPLES:DEMO_BUILD.COM procedure ensure that the transfer vector to the DECW$TRANSPORT_INIT routine is found at the beginning of the transport-specific shareable image.

Unlike the other transport-specific functions, DECW$TRANSPORT_INIT returns the address of the XTFT structure as its return value instead of a VMS condition code.

Example 8–20 shows a sample implementation of the DECW$TRANSPORT_INIT routine.

Example 8–20 Sample DECW$TRANSPORT_INIT Routine

```
    .
    .
    .
GLOBAL ROUTINE DECW$TRANSPORT_INIT =

BEGIN
LOCAL
    tft:        REF $BBLOCK ;

tft = tcpip_tft ;
tft[xtft$l_required0] = xtft$k_required0 ;
tft[xtft$l_reserved0] = 0 ;
tft[xtft$a_execute_write] = decw$$tcpip_execute_write ;
tft[xtft$a_write] = decw$$tcpip_write ;
tft[xtft$a_write_user] = decw$$tcpip_write_user ;
tft[xtft$a_execute_free] = decw$$tcpip_execute_free ;
tft[xtft$a_free_input_buffer] = decw$$tcpip_free_input_buffer ;
tft[xtft$a_close] = decw$$tcpip_close ;
tft[xtft$a_open] = decw$$tcpip_open ;
tft[xtft$a_attach_transport] = decw$$tcpip_attach_transport ;
tft[xtft$a_rundown] = decw$$tcpip_rundown ;
tft[xtft$l_xtcc_length] = xtcc$c_tcpip_length ;
tft[xtft$l_xtpb_length] = xtpb$c_tcpip_length ;
tft[xtft$l_xtdb_length] = xtdb$c_tcpip_length ;
tft[xtft$l_ixtcc_length] = ixtcc$c_tcpip_length ;
tft[xtft$l_required1] = xtft$k_required1 ;
.tft
END ;
    .
    .
    .
```

DECW$EXAMPLES:XPORT_EXAMPLE_XFER.MAR generates transfer vectors for the sample transport. A portion of the XPORT_EXAMPLE_XFER.MAR code follows:

```
    .
    .
    .
.PSECT $TRANSFER$     PIC,USR,CON,REL,LCL,SHR,EXE,RD,NOWRT,QUAD

TRANSFER DECW$TRANSPORT_INIT

.END
    .
    .
    .
```

The $TRANSFER$ program section points to a program unit, in this case DECW$TRANSPORT_INIT. The DEMO_BUILD.COM procedure then creates a cluster and collects the $TRANSFER$ program section in it, as described in Example 8–1.

Index

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal[1]	————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1] For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

VMS DECwindows
Transport Manual
AA–PABWA–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	□	□	□	□
Completeness (enough information)	□	□	□	□
Clarity (easy to understand)	□	□	□	□
Organization (structure of subject matter)	□	□	□	□
Figures (useful)	□	□	□	□
Examples (useful)	□	□	□	□
Index (ability to find topic)	□	□	□	□
Page layout (easy to find information)	□	□	□	□

I would like to see more/less ______________________________

____

____

What I like best about this manual is ______________________________

____

____

What I like least about this manual is ______________________________

____

____

I found the following errors in this manual:

Page Description

_____ ______________________________

_____ ______________________________

_____ ______________________________

_____ ______________________________

_____ ______________________________

Additional comments or suggestions to improve this manual:

____

____

____

____

I am using **Version** _____ of the software this manual describes.

Name/Title ______________________________ Dept. __________

Company ______________________________ Date __________

Mailing Address ______________________________

______________________________ Phone __________

- Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

– Do Not Tear - Fold Here

Cut Along Dotted Line