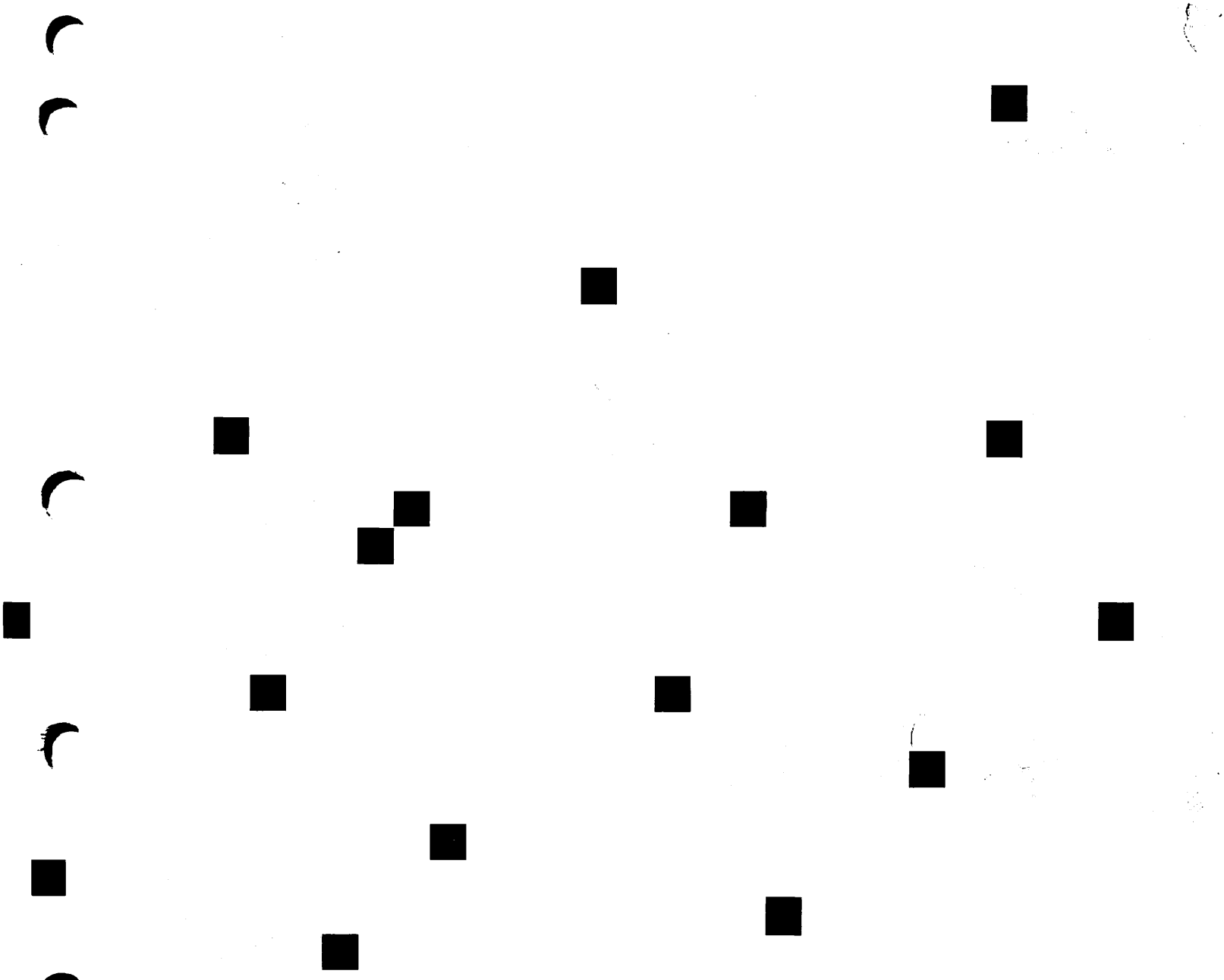

Prime™

Programmer's Guide to Prime Networks

Revision 21.0

DOC10113-1LA

Programmer's Guide to Prime Networks

First Edition

Robert Canavello
Emily Stone

Updated for Revision 22.0 By

Bernard Gilman

This document reflects the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision 21.0 (Rev. 21.0).

Prime Computer, Inc., Prime Park, Natick, MA 01760

COPYRIGHT INFORMATION

PRINTING HISTORY

First Edition (DOC10113-1LA) July 1987 for Revision 21.0

Update 1 (UPD10113-11A) September 1988 for Revision 22.0

CREDITS

Editorial: Barbara Fowlkes, Joyce Haines, Michael McNulty, Kathe Rhoades

Project Support: Bertil Lindblad, Peter Lynch, Graeme Williams, Nancy Webb Leblang

Illustration: Anna Spoerri

Design: Carol Smith

Document Preparation: Mary Mixon, Margaret Theriault

Production: Judy Gordon

Composition: Julie Cyphers, Anne Marie Fantasia

HOW TO ORDER TECHNICAL DOCUMENTS

Follow the instructions below to obtain a catalog, a price list, and information on placing orders.

United States Only: Call Prime Telemarketing, toll free, at 800-343-2533, Monday through Friday, 8:30 a.m. to 5:00 p.m. (EST).

International: Contact your local Prime subsidiary or distributor.

CUSTOMER SUPPORT CENTER

Prime provides the following toll-free numbers for customers in the United States needing service:

1-800-322-2838 (Massachusetts)
1-800-541-8888 (Alaska and Hawaii)
1-800-343-2320 (within other states)

For other locations, contact your Prime representative.

SURVEYS AND CORRESPONDENCE

Please comment on this manual using the Reader Response Form provided in the back of this book. Address any additional comments on this or other Prime documents to:

Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, MA 01701

Contents

About This Book

The *Programmer's Guide to Prime Networks* provides tutorial and reference information about

- Interprocess Communications Facility (IPCF) subroutines
- The programming interface to the File Transfer Service (FTS)

There are six chapters and four appendices:

- Chapter 1 is an introduction intended for programmers who have never written applications on a network before.
- Chapter 2 provides background information on virtual circuits.
- Chapter 3 describes IPCF programming principles.
- Chapter 4 is the keystone of the book. Each IPCF subroutine is described in detail.
- Chapter 5 provides IPCF programming examples.
- Chapter 6 explains how to incorporate the FT$SUB subroutine into a file transfer application that uses FTS. Examples are included.
- Appendix A contains information on how PRIMENET™ software supports the CCITT X.25 standards of 1980 and 1984.
- Appendix B describes the error messages that may be generated while writing an FTS application.
- Appendix C lists clearing causes and diagnostic codes for virtual circuits.
- Prime Network Glossary describes Prime networking terms. Key terms are printed in boldface type the first time they appear in the text.

Related Documentation

This guide is one of a series of books about Prime networks. The other books in the series may be of interest to programmers. They are listed below.

- *The User's Guide to Prime Network Services (DOC10115-1LA)*
- *Operator's Guide to Prime Networks (DOC10114-1LA)* and the update package for Rev. 22.0 (UPD10114-11A)
- *NTS User's Guide (DOC10117-2LA)*
- *PRIMENET Planning and Configuration Guide (DOC7532-4LA)*

Prime Documentation Conventions

The following conventions are used in command formats, statement formats, and in examples throughout this document. Examples illustrate the uses of these commands and statements in typical applications.

Convention	*Explanation*	*Example*
UPPERCASE	In command formats, words in uppercase indicate the names of commands, options, statements, and keywords. Enter them in either uppercase or lowercase.	**SLIST**
lowercase	In command formats, words in lowercase indicate variables for which you must substitute a suitable value.	**LOGIN user-id**
Abbreviations in format statements	If an uppercase word in a command format has an abbreviation, either the abbreviation is underscored or the name and abbreviation are placed within braces.	**LOGOUT** **{SET_QUOTA / SQ}**
Brackets [ ]	Brackets enclose a list of one or more optional items. Choose none, one, or more of these items.	**LD [-BRIEF / -SIZE]**
Braces { }	Braces enclose a list of items. Choose one and only one of these items.	**CLOSE {filename / ALL}**
Parentheses ( )	In command or statement formats, you must enter parentheses exactly as shown.	**DIM array (row, col)**
Hyphen –	Wherever a hyphen appears as the first character of an option, it is a required part of that option.	**SPOOL -LIST**

Bold italics in examples	In examples, user input is in bold italics but system prompts and output are not.	`OK, `***`RESUME MY_PROG`***
		`This is the output of MY_PROG.CPL`
		`OK,`

Angle brackets in messages < >	In messages, a word or words enclosed within angle brackets indicates a variable for which the program substitutes the appropriate value.	**Disk <diskname>**

1
Introduction to Network Programming

This chapter provides a context for programmers who write distributed applications on a Prime system. The following topics are discussed:

- The architecture of PRIMENET
- The underlying structure of applications that use the programming interface to PRIMENET
- The programming interface to the File Transfer Service (FTS)

PRIMENET Architecture

The 50 Series™ network services are based on **PRIMENET**, the Prime distributed network facility. PRIMENET software provides standardized network services over all physical media on which Prime systems operate.

PRIMENET is a packet-switching network, that is, a network in which each message is broken up into one or more **packets**, each of which the network transfers as a unit. PRIMENET transfers packets on a logical channel known as a **virtual circuit**.

PRIMENET consists of several levels, as shown in Figure 1-1. These levels are based on the **International Organization for Standardization's** Open Systems Interconnection (ISO OSI) model. The 50 Series systems are compatible with all other systems that support the same protocols. In particular, they are compatible with systems that support X.25 protocols at Layer 3 of the ISO OSI model.

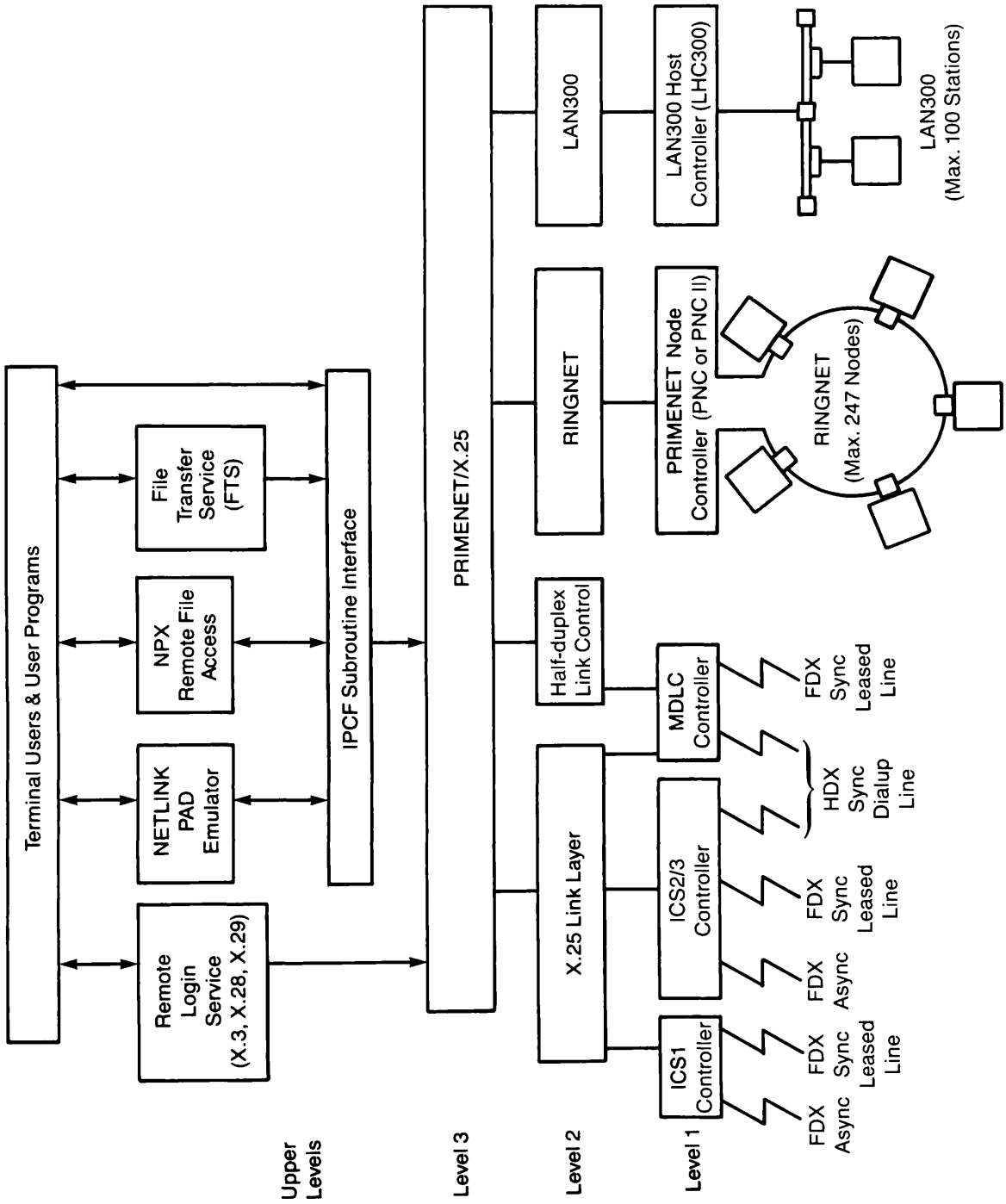

Figure 1-1
Levels of PRIMENET Architecture

PRIMENET Levels

PRIMENET's architecture consists of several functional levels, as shown in Figure 1-1. Each level contains a standard interface to the adjacent levels, and each level implements a **protocol** (a set of rules) to communicate with peer entities on different systems. PRIMENET has the following levels:

- Upper levels
- Level 3, the packet level
- Level 2, the link protocol level
- Level 1, the hardware interface

The upper levels of PRIMENET contain applications that perform actions directly specified by users or actions that facilitate completion of user tasks. These levels use the services of the lower levels. PRIMENET's upper levels correspond to the upper layers of the ISO OSI model; its lower levels, which contain the PRIMENET internals, correspond to the lower layers of the model.

Level 3, the packet level, corresponds to the ISO OSI network layer. The packet level implements the **CCITT (Consultative Committee for International Telephony and Telegraphy) X.25** protocol to manage the transmission of packets in the network. This level creates and controls virtual circuits across the network, handles error recovery, and controls the flow of information. It also keeps track of the destination process of each packet. User applications call the **Interprocess Communications Facility (IPCF)** subroutines to use the services of the packet level.

Level 2, the link protocol level, corresponds to the ISO OSI data link layer. It specifies the protocol that two linked systems must adhere to when transferring information between them. This protocol dictates the format of the data, how the systems should request, transfer, receive, and acknowledge the data, and how the systems signal faulty transmissions.

Level 1, the hardware interface level, corresponds to the ISO OSI physical layer. This layer acts as an intermediary between the physical transmission medium and the rest of PRIMENET.

IPCF Programming

With Interprocess Communications Facility (IPCF) subroutines, you can write multiple-process applications that can run on one or more PRIMENET systems. To exchange data, programs must follow a definite sequence of steps. For this reason, all applications that use the IPCF subroutines share a common underlying structure.

An application that uses IPCF subroutines to communicate with a remote system must have some way of identifying both the remote system and the destination process on the remote system. A **port** is a logical address within a given system. You can assign a port to a process, which permits that process to receive calls addressed to that port. The calling module initiates the call request. Once the virtual circuit is established, either module can transmit data messages. The sequence of tasks for each application is outlined below. Each task is followed by the name (or names) of the IPCF subroutine(s) that perform the task.

Note

This example is for an application with just two modules.

Tasks for the Calling Module

- Issue a call request.
- Check the virtual circuit status array to see if the connection is completed (X$CONN/ X$FCON/X$SCON/XLCONN).
- Transmit data (X$TRAN).
- Receive data (X$RCV).
- Clear the circuit (XCLR/XFCLR/XLCLR).

Tasks for the Called Module

- Assign the proper PRIMENET port (X$ASGN/XLASGN).
- Wait (X$WAIT) for an incoming call.
- When a call occurs, obtain data about it (X$GCON/X$FGCN/XLGCON/XLGC$).
- Accept the connection (X$ACPT/X$FACP/X$SACP/XLACPT).
- Transmit data (X$TRAN) and receive data (X$RCV).
- Clear the circuit (XCLR/XFCLR/XLCLR).

Note

Either the calling or the called application can terminate the connection by clearing the virtual circuit. The application program that receives the last message should be the one to issue the clear request.

While the above general steps are common to all IPCF applications, the details of each application will be unique, and will depend on the application's specific requirements for data transfer. In addition to the subroutines named in the above task lists, other routines can be used to

- Deassign ports (X$UASN/XLUASN)
- Clear all active virtual circuits (X$CLRA)
- Transfer control of virtual circuits to another process (X$GVVC/XLGVVC)
- Return status information (X$STAT)
- Return the contents of various packets (XLGA$, XLGC$, XLGI$)
- Reset the virtual circuit (X$RSET)

You can design a sequence of messages between the calling and called programs to ensure both programs' proper operation and to handle potential errors and malfunctions. For example, you can send interrupts (X$TRAN) and specify diagnostics when clearing a virtual circuit (XCLR/ XFCLR/XLCLR). You can use the Fast Select facility to transfer data and clear the circuit with one call. Chapter 5, IPCF Programming Examples, shows how to design an application using the Fast Select facility.

The calling application can function as a front-end program. The front-end program performs line control, message handling, code conversion, and error control. The front-end program also initiates the request to establish a virtual circuit to a program already running on a remote system. That remote program, the called application, is a server, and is likely to run as a phantom user process.

FTS Programming

As more applications involving distributed processing develop, corresponding programming interfaces are being created. One such interface exists to Prime's **File Transfer Service (FTS)**. The FT$SUB subroutine allows an application program to perform any function that a user can perform by using the FTR command. A program can use FT$SUB to submit, modify, cancel, abort, hold, release, and check the status of file transfer requests. FT$SUB cannot be called from R-mode or S-mode programs.

The FT$SUB subroutine makes use of keys and error codes specific to FTS, all of which have names beginning with F$ or Q$. These keys and codes are defined in an insert file in SYSCOM. In addition, the FT$SUB subroutine is installed as a shared subroutine library. A library file is used to satisfy the references to this subroutine during your program load sequence.

To use the FT$SUB subroutine, you must

- Declare the FT$SUB subroutine in your program.
- Use an %INCLUDE or $INSERT statement in your program to define the keys and error codes related to FT$SUB.
- Use the LIBRARY VFTSLB command in your program load sequence to load the FTS library.

Chapter 6, FTS Programming, explains in detail how to use the FT$SUB subroutine.

2

Ports and Virtual Circuits

This chapter discusses the following topics:

- Assigning ports
- Establishing a virtual circuit
- Polling virtual circuits
- Clearing causes and diagnostic codes for virtual circuits

Assigning Ports

To ensure that incoming calls reach their proper destination, PRIMENET passes calls to target processes based on port(s) that you have assigned to a target process. You assign a port to a process with a call to the X$ASGN or XLASGN routine. The port mechanism is depicted in Figure 2-1.

Normally you assign a port by number in the range of 0 through 99. Besides assigning a port by number, a privileged process can assign a port by matching the fields of an incoming call against a mask you have specified in the call to XLASGN. In this case you can use ports numbered 100 through 256. Refer to the description of X$ASGN/XLASGN in Chapter 4, IPCF Subroutines, for further information.

You can assign a port either permanently or with the provision that it be automatically deassigned after a certain number of connections. If you assign a port to a process where the port has been previously assigned, PRIMENET places the process on a queue for that port.

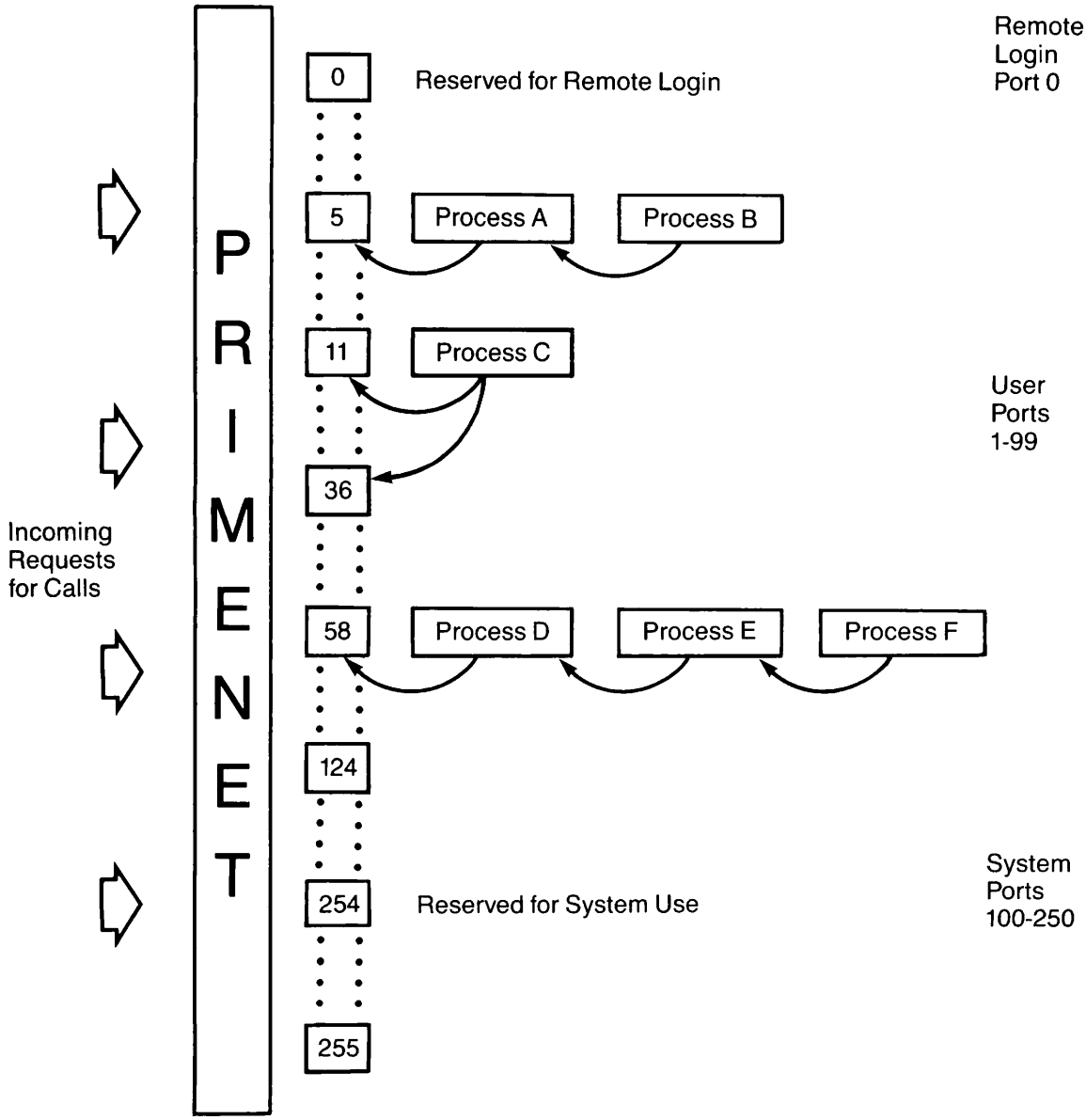

FIGURE 2-1
The Port Mechanism

You pass a parameter in the assign statement to specify the number of calls permitted to that port for that process. A negative value keeps that process last in the queue, ensuring that every request to that port gets processed before this request. A value of zero means that the process will handle all calls to the port. A process can assign more than one port. There is no relationship between the process number and the ID number of the assigned port.

When you call X$CONN in the calling module, you specify a port number to direct the call to the destination process that has assigned that port. If you do not specify a port number in the calling module, the call is sent to the default port on the target system, which is PRIMENET's remote login port (Port 0).

Note

A process that only makes calls, and does not receive them, need not assign a port.

Establishing a Virtual Circuit

When one process specifies the system and port of another process, PRIMENET establishes a bidirectional link between the two through a virtual circuit (VC). A **virtual circuit** is a logical path across the network from one process to another. Virtual circuits do not necessarily correspond exactly to physical communication lines. For example, one physical line may carry many virtual circuits, just as one telephone trunk line may carry many voices at the same time. Moreover, a virtual circuit between two given systems might use a different physical path each time it is established (if different physical paths exist). PRIMENET currently allows as many as 255 virtual circuits at a time on each system.

Note

The virtual circuit ID is what users and PRIMENET use to communicate about a particular virtual circuit. This is the *vcid* argument passed to the IPCF subroutines. The **Logical Channel Number (LCN)** is what PRIMENET and the network use to communicate about a particular virtual circuit. This is not the same value as the virtual circuit ID.

In addition to being able to create virtual circuits to ports on remote systems, you can create a virtual circuit to ports on the local system. You do this by specifying the local system as the destination system in the call connect subroutine. A virtual circuit created this way behaves exactly the same as one that is created to a remote system. This is known as a loopback connection. This feature is useful for developing and testing network applications because it permits the testing of several different pieces of a distributed application on the same system. Also, it permits local users to access a system the same way that remote users do.

PRIMENET permits you to switch a virtual circuit over to another application, or to remote login. Refer to the description of X$GVVC/XLGVVC in Chapter 4, IPCF Subroutines, for more information on transferring virtual circuits to another process.

Polling Virtual Circuits

Each process that holds virtual circuit connections must specify an array for each virtual circuit's status. The purpose of the virtual circuit status array is to provide processes with an easy way to poll the state of their virtual circuits. When needed, PRIMENET reports status changes for the virtual circuit into that array.

A process that is managing many virtual circuits, each with several data-moving operations in progress, need only poll the virtual circuit status arrays for each virtual circuit for the completed status (XS$CMP) to see which circuit(s) are ready for more traffic.

The caller specifies this virtual circuit status array in the call to the connect routine (X$CONN/ X$FCON/X$SCON/XLCONN). The call receiver specifies the array in a call to the accept routine (X$ACPT/X$FACP/X$SACP/ XLACPT). The virtual circuit status array may be written into by PRIMENET until the virtual circuit is cleared. The clear confirmation is written into the array as well.

The first of the two array elements is continually updated by PRIMENET to reflect the latest status of the circuit. When a data transmit or data receive completes, this virtual circuit status word is set to XS$CMP.

If the circuit is reset because of errors in the communications media or through network congestion, the virtual circuit status word is set to XS$RST. In addition, the status arguments of any subsequent data transmits or data receives are also set to the same value.

When a user-held network connection is disconnected (cleared), the first word of the virtual circuit status array is set to the "circuit cleared" status code (XS$CLR). At this time, the second word of the array is also valid and indicates the reason for the clearing.

Clearing Causes and Diagnostic Codes

A virtual circuit may be cleared by a **Packet Switched Data Network (PSDN)**, by PRIMENET, or by either of the two processes controlling it. The reason for clearing appears in the second word of the virtual circuit status array. The high-order byte of this word is the clearing cause and the low-order byte is the diagnostic code.

X.25 defines several network-generated clearing causes. These are listed below.

0	DTE Originated (Standard Diagnostics used)
1-127	Public Network Originated
128	DTE Originated (Nonstandard Diagnostics used)
129-191	Private Network Originated
192-255	Gateway Originated (gateway between public networks)

The clearing cause is most likely one of the CC$xxx codes listed Appendix C. If the clearing cause is CC$CLR, the circuit was cleared by either PRIMENET or a user process (DTE-originated). PRIMENET currently clears all calls with a clearing cause of zero (CC$CLR), except in the following three situations:

- When a reverse charge call is not allowed (CC$GRN)
- When there is insufficient memory for the **Route-through Server** to execute its protocol (CC$GCN)
- When the Route-through Server detects an error it cannot recover from without clearing the circuit (CC$GPE)

If you are writing applications that do not conform to ISO standards, use a clearing cause of 128 and nonstandard diagnostics if your PSDN allows. If you want to use a clearing cause of zero, then use diagnostic codes from 128 through 143.

When a call is cleared explicitly with XCLR/XFCLR/XLCLR, the clearing user process may specify the value of the diagnostic code in the *why* argument. Communicating processes may use this facility to describe fatal error conditions or to supply final status information. The list of status codes that may be written into the first word of the virtual circuit status array appears in the sections on X$CONN/X$FCON/X$SCON/XLCONN and X$ACPT/X$FACP/X$SACP/XLACPT in Chapter 4, IPCF Subroutines.

3

IPCF Programming Principles

This chapter elaborates some of the techniques and principles of IPCF programming. The following topics are discussed:

- Front-end principles
- Server principles
- Storage of variables
- Performance issues
- The Fast Select facility
- Window and packet sizes in virtual circuits (throughput)
- Network event waiting
- Checking return codes
- The virtual circuit status array
- Defining message protocols
- Virtual circuit clearing
- Program closedown
- The effect of START_NET and STOP_NET on IPCF programs

Front-end Principles

A good front-end program should

- Always keep the user who runs the program updated on communication progress
- Ensure that anyone running the program cannot cause malfunction of the server

Thus, the front-end program should recognize and handle situations such as the unavailability of a server, downtime on the remote system, and unexpected resets and clear requests during message exchanges. In addition, the front-end program should use a condition handler to clear unterminated virtual circuits left alive when the program has been aborted.

Server Principles

There are two models for the design of network **servers**: single-threaded and multi-threaded. The single-threaded server assigns its port for one call request each time and reassigns the port after completing a transaction. You can run a single instance of the single-threaded server when the call will not be open for a long time. Or, you can run multiple instances of a single-threaded server. Multiple single-threaded servers all assign the same port, and they handle incoming call requests in rotation. This is described in detail in the section on X$ASGN/XLASGN in Chapter 4, IPCF Subroutines. In contrast, a multi-threaded server assigns one or several ports for all calls, and regularly scans for new incoming calls.

Whichever design you use, there is a limit to the number of active requests, limited by either the parallel capacity of the multi-threaded server, or by the number of running single-threaded servers. Each server design must take into account what to do with additional calls when all servers, or all slots within a multi-threaded server, are in use. Either additional calls can be cleared, or they can be ignored until a server becomes free or the incoming call times out.

A convenient solution to this problem is to run a special server that assigns the common server port for an infinite number of calls *at the end of the assign queue*. The description of X$ASGN/ XLASGN in Chapter 4, IPCF Subroutines, explains how to make this assignment. This special server is called only if no other server is available. Its sole action is to transmit the message, **`No server available`** and to clear the call.

Single-threaded servers are easier to design and maintain because they do not need any internal task-switching code and require only one set of state information. However, for synchronization or because resources are scarce on your machine, you may prefer to use a multi-threaded server.

A well-designed server should be stable if it encounters errors. Preferably it should not crash but rather reinitialize itself and reenter service. The condition-handling mechanism of PRIMOS® should be used to catch and properly handle forced logouts and similar events. The server also should create a log file so that problems can be analysed later.

An application based on multiple servers must be designed to be independent of user-specific properties, such as the local user number of the server. The reason is that one can never know which server is used for a specific transaction.

Storage of Variables

Most IPCF routines have status return arguments that are updated to indicate a completed network operation (for example, transmit completed). PRIMENET updates status arguments both at return to the immediate caller and at later times. Such variables must remain stable during program execution. In particular, a status variable should not reside in the stack of a subroutine that returns to a caller *before* the status update is completed. If the stack area is used by another routine, the first return variable will overwrite the stackframe. Such errors are extremely difficult to trap because they are difficult to reproduce.

Thus, you should store an updated variable in an area of memory that is relatively stable — that is, an area that will not be deallocated as long as the program needs the value. For example, you can store a variable in the stack, provided that the stackframe in which the variable resides is not released. Thus, the routine in which you allocate this variable must not return and release the stackframe until PRIMENET is finished with the variable, or the request for the argument or call logically ends.

The program fragments provided below show correct and incorrect methods of storing variables. Refer to Chapter 4, IPCF Subroutines, for descriptions of the IPCF parameters appearing in these fragments.

Example 1: Incorrectly Storing a Variable

A program using the following BAD_XMIT function may encounter difficulties because the arguments to X$TRAN are stored in the stack, while the function returns immediately to the caller, freeing that stack frame. If the request does not receive an immediate response, the network server may return later and find that some other routine has been called after BAD_XMIT and is using the same stack frame for a different purpose. In this case, the network server sends 65 bytes of the wrong data.

```
      INTEGER FUNCTION BAD_XMIT (VCID)
      EXTERNAL X$TRAN
      INTEGER VCID, MESSAGE_STATUS
      CHARACTER*71 XMIT_MESSAGE

$INSERT SYSCOM>X$KEYS.INS.FTN

C   End of Declarations

      XMIT_MESSAGE = 'Information on the stack could be overwritten.'

1000  CALL X$TRAN (VCID, XT$LV0, XMIT_MESSAGE, 71, MESSAGE_STATUS)
      BAD_XMIT = MESSAGE_STATUS
      RETURN
      END                              /* FUNCTION BAD_XMIT */
```

Example 2: Correctly Storing a Variable in the Stack

The following function works correctly. Although the arguments to X$TRAN are in the stack (and therefore disappear when the function returns from the routine), the network software is finished with the variables by the time the function returns.

```
      INTEGER FUNCTION XMIT_MSG (VCID)
      EXTERNAL X$TRAN, X$WAIT, SLEEP$

      INTEGER  VCID,                  /* from previous X$CONN/X$ACPT */
     X         MESSAGE_STATUS
      CHARACTER*32 XMIT_MESSAGE

$INSERT SYSCOM>X$KEYS.INS.FTN

C   End of Declarations

      XMIT_MESSAGE = 'This one works, because it waits'
1000  CALL X$TRAN (VCID, XT$LV0, XMIT_MESSAGE, 32, MESSAGE_STATUS)
2000  GOTO (                        /* ANALYZE RESULT
     X     2100,                    /* XS$NET: net down
     X     2300,                    /* XS$CMP: operation complete
     X     2200,                    /* XS$IP:  operation in progress
     X     2100,                    /* XS$BVC: bad VC
     X     2100,                    /* XS$BPM: bad parameter
     X     2100,                    /* XS$CLR: VC cleared
     X     2400,                    /* XS$RST: VC Reset
     X     2400,                    /* XS$IDL: VC idle (?)
     X     2100,                    /* XS$UNK: address unknown (?)
     X     2500,                    /* XS$MEM: insufficient buffers
     X     2100,                    /* XS$NOP: no call reqs pending(?)
     X     2100,                    /* XS$ILL: operation illegal
     X     2100,                    /* XS$DWN: node down (?)
     X     2500,                    /* XS$MAX: max requests made
     X     2100,                    /* XS$QUE: assign queued (?)
     X     2100                     /* XS$FCT: illegal facility (?)
     X         ) MESSAGE_STATUS + 2

2100  CONTINUE                      /* something went wrong
      XMIT_MSG = MESSAGE_STATUS
      RETURN
2200  CONTINUE                      /* nothing happened yet
      CALL X$WAIT (0)               /* sleep until something happens
      GOTO 2000
2300  CONTINUE                      /* Success
      XMIT_MSG = MESSAGE_STATUS
      RETURN
```

```
2400  CONTINUE                      /* request was flushed:
      GOTO 1000                     /* we'll have to send it again.
2500  CONTINUE                      /* temporary blockage:
      CALL SLEEP$ (001000)          /*   sleep,
      GOTO 1000                     /*   and try again
      END                           /* FUNCTION xmit_msg
```

Example 3: Storing a Variable in a Static Area

The following STATXMIT routine functions correctly because the storage is statically allocated. The caller must be careful not to reuse the same storage for another purpose (for example, calling STATXMIT again) before the network software is finished with the original contents. To detect this condition, check the value of MESSAGE_STATUS as it was checked in XMIT_MSG, in Example 2.

```
      SUBROUTINE STATXMIT (VCID)
      INTEGER VCID, MESSAGE_STATUS
      EXTERNAL X$TRAN
      CHARACTER*48 XMIT_MESSAGE
      COMMON /XMITMSG2/MESSAGE_STATUS, XMIT_MESSAGE
$INSERT SYSCOM>X$KEYS.INS.FTN
C   End of Declarations

      XMIT_MESSAGE = 'This works because the arguments are static'
      CALL X$TRAN (VCID, XT$LV0, XMIT_MESSAGE, 48, MESSAGE_STATUS)
      RETURN
      END                           /* SUBROUTINE STATXMIT
```

For further information, refer to the Program Closedown sections later in this chapter, and to the descriptions of XCLR/XFCLR/XLCLR in Chapter 4, IPCF Subroutines.

Performance Issues

Although the IPCF interface to PRIMENET functions independently of transmission media such as RINGNET™ or PSDN synchronous links, throughput varies greatly between these media. An application that runs well over RINGNET, due to the very high throughput over the ring, may turn out to be very slow over a synchronous line. In general, application designs should

- Minimize the amount of data transferred over the network
- Use as few messages at the user level as possible
- Use as few protocol turnarounds (when one of the programs has to wait for the other to send a message back before it can proceed) as possible

The message transfer structure of X$TRAN and X$RCV makes the partitioning of messages into proper X.25 protocol data packets invisible for the application programmer. However, each message that is handed to X$TRAN will be packetized inside PRIMENET. Thus, you may be able to improve performance by making the message size match the packet size of the virtual circuit, or integer multiples thereof.

You should ensure that your application always provides sufficient buffer space to handle incoming data. Further, you can optimize program performance by specifying a large window and packet size for the virtual circuit. A large packet size may reduce overhead, since fewer packets have to be analyzed and handled. The larger window size allows PRIMENET to have more packets outstanding, reducing the time interval between them. This is particularly important over slower media. For specific information about how to control window and packet sizes, refer to Appendix A, X.25 Programming Guidelines.

The Fast Select Facility

PRIMENET supports the X.25 Fast Select facility. This facility allows you to transmit a limited amount of user data in the packets that set up a virtual circuit.

For example, suppose you want the process that initiates a call to identify itself to the receiving system for purposes of validation. To save time, you might want this validation to occur while the call is being established, rather than afterwards. With the Fast Select facility, you could arrange for the initiating system to include an identifying string with its call request. The receiving system could then include an acknowledgment of that string when it accepted the call. By the time the call was established, the process of validation would be over. On the other hand, if the initiating system omitted the string or sent the wrong one, the receiving system could assume that the call was invalid and prevent its completion.

In general, the Fast Select facility is most useful when you need to exchange only a small amount of data. As in the example above, this small data exchange could be used to determine whether a lengthier conversation is required.

The subroutine descriptions in Chapter 4, IPCF Subroutines, include the information you need to use the Fast Select facility. Routines whose names begin with X$F are defined specifically for Fast Select. For an example of an application that uses Fast Select, refer to Chapter 5, IPCF Programming Examples. Fast Select is fully defined in the X.25 standard.

Network Event Waiting

One essential feature of IPCF applications is their asynchronous nature. Your user program initiates data transfers and other network activities, but returns from many of the IPCF subroutines immediately after the request is launched, rather than after it completes. This is to your advantage. Your program can perform other functions while PRIMENET is performing tasks for your application.

An IPCF subroutine may return a status code immediately after a request is made. In some cases, this status code may indicate a serious error or an inability to initiate the requested action. For example, a status code of XS$BPM means that your call arguments contain illegal values. The return code XS$MEM indicates work contention inside your local PRIMENET, meaning that PRIMENET has temporarily run out of buffers.

An IPCF subroutine may also have returned a status code at a later time — for example, after a request has completed, or when some other network action has occurred. When updating a status array, the subroutine also notifies the process' network semaphore. Your program, when idle, can wait on its network semaphore. When the semaphore is notified, the program can check the status arrays to find out what has happened. To wait on its network semaphore, your program calls the X$WAIT routine. You can use a combined timeout/network-event wait. The returned function value indicates whether the program woke up on timeout or on an actual event.

Semaphores, which are discussed in the *Subroutines Reference Guide, Volume III*, usually include event counters that monitor the number of waiting processes or events to be handled. PRIMENET uses a different principle. It generates only one collective notification for all your network events, until you wait on the semaphore again by invoking X$WAIT. This method avoids the problem of overnotification in case the IPCF user program does not wait on the semaphore. Therefore, if your application has several outstanding network requests (such as multiple supplied receive buffers) you should check the status for *all* of them before waiting again on the network semaphore, or your program might hang because the notify had already taken place.

Once a virtual circuit has passed into data transfer phase, there are no timeout mechanisms inside PRIMENET for an idle circuit with no data to transfer. If one side issues X$RCV, followed by an infinite wait for the other end to respond, and the other end crashes without clearing the virtual circuit before transmitting, the receiving side hangs. It would be better to have a long timeout period and send the user a message about possible problems.

Checking Return Codes

Calls to X$CLRA and X$UASN/XLUASN always return without error; all other calls return with (or affect the value of) a status word or array. The immediate-return design of the IPCF routines implies that several return status values are reasonable. Normally, an IPCF application program will have several code paths following each IPCF call.

For example, when an application has reached the data transfer phase and is manipulating a number of calls to X$TRAN and X$RCV, the return statuses XS$IP, XS$CMP, possibly XS$RST, and XS$CLR are all normal and must be handled. (For explanations of these codes, refer to the descriptions of X$TRAN and X$RCV in Chapter 4, IPCF Subroutines.) Thus, the application might contain a sequence of statements such as the following:

```
IF (returned_status .EQ. XS$xxx) GO TO yyy
```

In this case, you must use a local copy of returned status in the sequence of IF statements. This guarantees that the branching is correct and is not upset by a sudden change of the returned status value by PRIMENET during the testing. (The most common change would be the transition from

XS$IP to some other status, indicating that the operation terminated.) This technique is illustrated in the following sample code.

```
50    CALL X$WAIT(10)              /* Idle a while...
      J = VCSTAT(1)                /* Copy circuit status value.
      IF (J .EQ. XS$IP) GOTO 50    /* Keep idling...
      IF (J .EQ. XS$CLR) GOTO 10   /* OK to accept next call.
      IF (J .EQ. XS$CMP) GOTO 50   /* Wait for clear.
```

VCSTAT(1) is copied into J, and the three comparisons are made against J, rather than VCSTAT(1), ensuring that the value does not change between the tests.

The return code XS$BVC occurs when you order actions on a virtual circuit that you do not control. It is usually a fatal error code, implying that your program is in error. However, sometimes it also can be a normal (nonfatal) return code for calls to X$TRAN, X$RCV, and XCLR/XFCLR/XLCLR. XS$BVC may be nonfatal if the virtual circuit has been cleared by the other side or by the network after your last status test (which did not return XS$CLR), but before your call to X$TRAN, X$RCV, or XCLR/XFCLR/XLCLR.

IPCF applications should always check the status array returned or affected by any IPCF subroutine call. The following example shows some pitfalls that you can avoid.

```
NPORT = 3279
CALL X$ASGN(NPORT, 1, NSTAT)
CALL X$WAIT(0)
```

Because the port number 3279 is invalid, X$ASGN returns with the error XS$BPM in NSTAT. The example fails to discover the error, and waits forever for a network event on an unassigned port.

The Virtual Circuit Status Array

The virtual circuit status array, which is described in Chapter 2, provides processes with an easy way to poll the state of their virtual circuits. A process that is managing many virtual circuits, each with several data-moving operations in progress, need only poll the virtual circuit status arrays for the completed status (XS$CMP) to see which circuit(s) are ready for more traffic. PRIMENET will change the virtual circuit status from XS$IDL or XS$IP to XS$CMP to indicate that it is ready for more data transfers.

Just as in the case of return codes, your application should take into account the possibility that the virtual circuit status array may change between consecutive tests. PRIMENET may write into the virtual circuit status array throughout the life of the virtual circuit. In addition, PRIMENET data communication is handled by a special process, NETMAN, that is classified as user type NSP. This process runs at high priority, and thus its activity might interrupt a program during its execution. An interruption affects the program's handling of the virtual circuit status array (as

well as of other returned status variables). As in the case of return codes, obtain a local copy of the virtual circuit status array before doing repeated tests to ensure that the array remains unchanged during the tests.

Defining Your Own Message Protocols

The goal of the X.25 protocol levels 2 and 3 is to deliver data in order and without errors. However, there is always a slight possibility that the error recovery routines of levels 2 and 3 may fail. Should this occur, the virtual circuit is reset.

Circuit resets can result in loss of data, because packets that are traveling in the network may disappear. In most cases, you can detect a reset because uncompleted receive requests (calls to X$RCV) return the status code XS$RST rather than XS$CMP. However, you may not be able to detect a reset that occurs after the transmitting program sends data by means of X$TRAN but before your program issues the corresponding call to X$RCV. No data is received, but X$RCV does not know that a reset has occurred, and thus does not return a status of XS$RST. The only way to detect the reset in this case is to check the virtual circuit status array. To avoid this problem be sure that the receiving process calls X$RCV and establishes the receive buffer *before* the transmitting process calls X$TRAN. This practice ensures that if a reset occurs after the data is sent, the pending X$RCV call returns a status of XS$RST.

Some other methods of ensuring data integrity are

- Periodically send checkpoint messages that can be distinguished from normal data messages. If a reset or other error occurs, you can begin resending any messages sent since the last checkpoint message.
- Include a sequence number on each message.
- Pass length information with the X$TRAN message.

Virtual Circuit Clearing

The X.25 protocol states that when either side of a circuit requests to clear the circuit, packets in transmission can either be delivered properly or be dropped. For this reason, the circuit should be cleared by the side that receives the last message. If the transmitting side clears the circuit, the receiving side may detect the clear before the last call to X$RCV. In this case, the last transmitted message is dropped. This problem can occur even if the transmitting side refrains from clearing the circuit until the last call to X$TRAN returns a status of XS$CMP.

Program Closedown

When you design the termination of an IPCF application, be sure to consider the following important points:

- When you have assigned ports for receiving incoming calls, make sure that you call either X$UASN/XLUASN or X$CLRA to release these ports. Otherwise, these ports will remain assigned for you, routing incoming calls to you and preventing other applications from using them to receive calls.
- As long as you have a running virtual circuit on which you have sent a clear request (by calling X$CLR or any corresponding routine), PRIMENET will write into your virtual circuit status array to indicate that the remote end has confirmed the clear request. To prevent overwriting of other programs that may be executed later, you should not CALL EXIT or return to command level until one of the following has occurred:
  - o All virtual circuit status arrays' first words have changed to XS$CLR, indicating that all clear requests are confirmed.
  - o You have called X$CLRA, which forces an immediate drop of all your virtual circuit references, including any nested EPFs. (PRIMENET will still handle the clear confirmation properly.) The call to X$CLRA will generate a clear request with diagnostic byte 0 if the circuit has not already been cleared.

Note

If you request a clear immediately for an incoming call, without first accepting it, no virtual circuit status array is created, so you do not need to wait for confirmation.

The Effect of START_NET and STOP_NET on IPCF Programs

Your IPCF application should provide for events that occur when the network is stopped via STOP_NET or started via START_NET. This consideration is especially important for servers running as phantoms on nodes that execute STOP_NET.

When NETMAN is stopped, active virtual circuits are cleared — an event that is indicated by the Prime-defined clearing diagnostic, CD$NSV. When an IPCF application receives this clearing diagnostic, either its local NETMAN or the remote node's NETMAN has been closed down.

A user program that has assigned a port and is waiting for an incoming call on its network semaphore is notified when the network is stopped. The normal action for the program is to call X$GCON/X$FGCN/XLGCON/XLGC$ to find out about the supposedly incoming call. The returned status will be "Networks not configured", which will indicate to the program that STOP_NET has been executed.

When START_NET is later invoked, the local network configuration database is reinitialized, and all previous port assignments are nullified. START_NET does not notify waiting processes that this has occurred. Therefore, if your application failed to detect that STOP_NET was issued, it might continue to wait on its network semaphore, assuming its old port assignment, which would no longer be in place. In that case, the program would be left hanging. The system operator should keep track of IPCF servers waiting for incoming calls, and make sure they are properly restarted in connection with STOP_NET and START_NET activity.

The "Network not running" status, XS$NET, is returned only by IPCF routines that initiate network connections, such as X$ASGN, X$CONN/X$FCON/XLCONN, and X$GCON/ X$FGCN/XLGCON/XLGC$, and by the status routine X$STAT. Other routines combine this status with XS$BVC, which is also returned if you require an action for a virtual circuit that does not belong to you.

4
IPCF Subroutines

This chapter describes the subroutines that make up the Prime Interprocess Communications Facility (IPCF), and is divided into four parts:

- IPCF overview
- Naming conventions
- Summary of subroutines
- Subroutine descriptions

This chapter assumes that you understand basic network programming concepts, ports, and virtual circuits.

IPCF Overview

Each IPCF subroutine has a simple form (the short form) with a short parameter list. Most routines also have a long form with a larger number of parameters. The short forms can be used without a detailed knowledge of X.25 protocol. The information you need to use the short forms is contained in this chapter.

The long forms allow you to use X.25 functionality more fully and are most useful in handling connections to non-Prime systems. To use the long forms, you must know the X.25 protocol and understand the basic concepts of network programming. Appendix A, X.25 Programming Guidelines, contains additional information about the X.25 protocol.

Note

Read the Restrictions section of Appendix A before using the long forms of the IPCF subroutines.

Naming Conventions

Each IPCF subroutine name begins with one of the following prefixes:

- X$
- X$F

- X$S
- XL

The X$ prefix indicates a short form or a subroutine that has only one form. For example, there is only one wait subroutine, X$WAIT; on the other hand, X$CONN is the short form of the call request subroutine.

The X$F prefix indicates a short-form subroutine tailored for the Fast Select facility. For example, X$FCON is the Fast Select version of the call request subroutine.

The X$S prefix designates a short-form subroutine that has included the KEY argument from the long form of the subroutine. This is used to accommodate an added value to the current KEY options.

The XL prefix indicates a long form subroutine. For example, XLCONN is the long form of the call request subroutine.

Summary of IPCF Subroutines

The IPCF subroutines listed below can be called by any application that runs as a V-mode or I-mode program. To load the IPCF library into your program, use the LI VNETLB subcommand with BIND or SEG. The definitions of key and error codes for FORTRAN, PL/I, and Pascal are kept in the SYSCOM>X$KEYS.INS.FTN, SYSCOM>X$KEYS.INS.PL1, and SYSCOM>X$KEYS.INS.PASCAL files, respectively. The SYSCOM>X$KEYS insert file is kept for compatibility with old program source files. It is a copy of the SYSCOM>X$KEYS.INS.FTN file. Programmers using other languages must create their own versions of these files.

Subroutine	*Function*
X$ASGN **XLASGN**	Assigns a port either by number or by specifying a mask that traps incoming calls
X$CONN **X$SCON** **X$FCON** **XLCONN**	Requests a virtual circuit connection
X$GCON **X$FGCN** **XLGCON** **XLGC$**	Provides information about incoming calls
X$ACPT **X$FACP** **XLACPT**	Accepts a connection request to complete a connection
XLGA$	Provides information about Call Connected packets
X$TRAN	Transmits a message
X$RCV	Prepares for receiving a message

X$CLR **X$FCLR** **XLCLR**	Clears a virtual circuit connection
XLGI$	Provides information about cleared calls
X$UASN **XLUASN**	Deassigns a port
X$WAIT	Does a timed wait for the next network event completion
X$CLRA	Clears all active virtual circuits and deassigns all ports
X$GVVC **XLGVVC**	Passes control of a virtual circuit to another application process
X$STAT	Returns various status information related to PRIMENET
X$RSET	Resets a virtual circuit

Subroutine Descriptions

This section describes the IPCF subroutines in detail. The subroutines are grouped functionally and are presented in the logical calling order listed above.

Note

The integer default is INTEGER*2 in FTN and INTEGER*4 in F77. To avoid incorrect results, declare all integers explicitly. All integer arguments to these routines are INTEGER*2.

PL/I programmers should combine the components of *key* with a logical OR rather than adding them. FORTRAN programmers should add them.

Subroutine: X$ASGN XLASGN

Description: Assigning a Port

To receive incoming network connection requests, an application must assign one or more of the available network ports. Each call to X$ASGN or XLASGN assigns one port to the calling process. Once a port is assigned to a process, each incoming connection request specifying that port is directed to the connection request queue of the process. The process may read the queue by calling the X$GCON, X$FGCN, or XLGCON subroutine. As many as 255 port assignments may be made.

Note

The limit of 255 port assignments is a limit that applies to each system. Remote login uses two port assignments and each configured slave process uses one port assignment. This limit need concern only large systems that have both many slave processes and many processes receiving incoming network connection requests.

The X$ASGN subroutine is adequate for most port assignments. The XLASGN subroutine creates an extended port assignment. Extended port assignments may be made by network servers. Network servers are spawned from the supervisor terminal or other privileged processes by invoking the START_NSR command:

START_NSR server.comi -USER_ID server_name

If a process requests a port legally, PRIMENET assigns the port unless the buffer space is insufficient. PRIMENET puts multiple requests for the same port in a first in/first out queue. When the process at the head of the queue deassigns the port (see X$UASN/XLUASN), the next process waiting in queue for the port begins receiving incoming connection requests.

The assigning process may request automatic deassignment of a particular port after a specified number of connection requests are directed to the port. The *count* argument in the X$ASGN or XLASGN call is used to request automatic deassignment. If *count* is a positive integer, PRIMENET removes the assigning process from the port after processing *count* connection requests for the port. The assigning process must assign the port again to reenter the queue. A *count* of 0 prevents automatic deassignment.

A *count* of –1 causes the assignment request to be placed and kept at the back of the assignment queue for the specified port. PRIMENET queues ahead of this request any process that calls X$ASGN or XLASGN and has a nonnegative *count.* A process with a *count* of –1 handles a request only when no other processes are available. The process with the *count* of –1 returns to the bottom of the queue when another process with a nonnegative *count* assigns the specified port.

If more than one process with a *count* of –1 assigns the same port, only the first handles bottom-of-queue requests, until it deassigns itself or logs out. At that time, the next process with a *count* of –1 assumes the bottom-of-queue position.

For example, suppose several server processes, each with a *count* of 1, assign one port. Each process handles a single request for service, and is immediately deassigned, allowing the next

incoming request to be taken by the next server in the queue. An error-handling process is given a *count* of –1 and sits at the bottom of the queue. When no server process is available, the error handler takes the incoming requests.

If one process makes two successive calls to X$ASGN or XLASGN with the same port, the *count* of the queued first assignment request call is replaced by the *count* of the second. The position of the request in the assignment queue remains unchanged.

In the case of an extended assignment (a call to XLASGN), calls are trapped by a combination of the Called Address Extension, Protocol ID, and/or User Data fields supplied in the call. The particular combination is specified in the *key* argument.

The call syntax for XLASGN follows the call syntax for X$ASGN.

Call syntax

```
CALL X$ASGN(port, count, status)
```

Arguments

port

INTEGER*2. (INPUT). The number of the port to be assigned. 0 through 99 for unprivileged processes; 0 through 255 for privileged processes.

count

INTEGER*2. (INPUT). If greater than zero, specifies the number of incoming requests to be directed to the assigning process before automatic deassignment. If zero or –1, the port assignment remains in force until manually removed by XLUASN or X$UASN, or until the user calls X$CLRA or issues the ICE command. If –1, the assignment is placed after all previous assignments of the same class with a nonnegative value of *count*.

status

INTEGER*2. (OUTPUT). The returned status of the call. The following status codes may be returned by a call to X$ASGN or XLASGN:

XS$BPM	At least one of the parameters (port or count) specified in the call is not in the legal range.
XS$CMP	The assign request has been successfully placed at the front of the assign queue for the specified port.
XS$MEM	No system resources are currently available for more assign requests.
XS$NET	Networks are not configured for this system.
XS$ILL	Either a privileged key has been supplied by a nonprivileged user or you have assigned an invalid port number.
XS$QUE	The assign request is behind at least one other identical request. For X$ASGN, this means that this port is behind request(s) by other users.

Note

The calling sequence for the routine XLASGN is not consistent with the other IPCF routines. The arguments that are normally FORTRAN arrays of characters or PL/1 character nonvarying strings are PL/1 character varying strings for XLASGN.

To call XLASGN from FORTRAN, you must build a data structure that is interpreted as a character varying string, that is, a 16-bit length field followed by an array of characters. For example, the *prid* argument is defined as CHAR(4)VAR. The equivalent FORTRAN declaration is INTEGER*2 PRID(3), with PRID(1) set = 4, and PRID(2) and PRID(3) holding the four bytes of the *prid* field. To pass a value of 999912341234 in the *tadr* field, declare INTEGER*2 TADR(9), and set TADR(1) = 12 and put the characters 999912341234 in TADR(2) through TADR(7).

Call syntax

```
DCL XLASGN ENTRY(BIT(16), CHAR(15)VAR, CHAR(15)VAR, CHAR(4)VAR,
          CHAR(128)VAR, FIXED BIN(15), CHAR(41)VAR, FIXED BIN(15),
          FIXED BIN(15), FIXED BIN(15), FIXED BIN(15));

CALL XLASGN ENTRY(key, tadr, tsadr, prid, udata, port, txadr, rsvd1,
          rsvd2, rsvd3, status);
```

Arguments

key

INPUT. Structure as defined below, an overlay for bit(16).

```
DCL 1 key,
      2 mbz bit(6),      /* not used, must be 0 */
      2 daxr bit(1),     /* XK$DAX: select on called addr extn */
      2 flush bit(1),    /* XK$FLU: remove all port assignments;
                            not used for XLASGN */
      2 dadr bit(1),     /* XK$DAD: select on address prefix */
      2 dasr bit(1),     /* XK$DSA: select on address suffix */
      2 dprid bit(1),    /* XK$DPR: select on prid */
      2 dudata bit(1)    /* XK$DUD: select on user data */
      2 dport bit(1),    /* XK$DPO: select on port number */
      2 mbz2 bit(1),     /* not used, must be 0 */
      2 dme bit(1),      /* XK$DME: select only calls to local node */
      2 mbz3 bit(1);     /* not used, must be 0 */
```

Specifies which incoming calls are to be returned. Formed from the parts listed below. Some parts are required and others are optional, as indicated in the list. With the exception of XK$DME and XK$DPO, the parts may be supplied by privileged processes only.

One of the following parts is required:

XK$DAD	Returns incoming calls whose Called Addresses start with the digits specified in *tadr* and are not among the X.25 addresses configured for this node
XK$DME	Returns incoming calls whose Called Addresses are among the X.25 addresses configured for this node

Optional part is

XK$DSA	Returns incoming calls whose Called Addresses end in *tsadr* and whose remaining digits comply with the XK$DME/XK$DAD specification

You can also use the optional parts described below. You can use any one of them individually, or XK$DPR and XK$DUD together.

XK$DAX	Returns incoming calls when the Called Address Extension facility is present and when the value of the Called Address Extension begins with *txadr*
XK$DPO	Returns incoming calls whose port numbers are *port* (and that do not have a Called Address Extension facility present)
XK$DPR	Returns incoming calls whose Protocol ID fields begin with *prid* (and that do not have a Called Address Extension facility present)
XK$DUD	Returns incoming calls whose User Data fields begin with *udata* (and that do not have a Called Address Extension facility present)

tadr

INPUT. Holds a char varying string of ASCII digits, the Called Address prefix. Used only if *key* includes XK$DAD.

tsadr

INPUT. Holds a char varying string of ASCII digits, the Called Address suffix. Used only if *key* includes XK$DSA.

prid

INPUT. Holds a char varying string of bytes, the Protocol ID field. If both *prid* and *udata* are supplied, then the length of *prid* must be 4. Used only if *key* includes XK$DPR.

udata

INPUT. Holds a char varying string of bytes, the User Data field. The maximum length is 12 if the *prid* argument is supplied, and 16 otherwise. Used only if *key* includes XK$DUD.

Note

If both XK$DUD and XK$DPR are used, then *prid* and *udata* are concatenated at runtime. If only XK$DUD is supplied, then the User Data field is assumed to include the Protocol ID field.

port

INPUT. The number of the port to be assigned. 0 through 99 for unprivileged processes; 0 through 255 for privileged processes. Used only if *key* includes XK$DPO.

txadr

INPUT. Holds a string of bytes, the Called Address Extension prefix to use in the search. Used only if *key* includes XK$DAX.

Supply the Called Address Extension prefix in binary-coded decimal, using a maximum of 41 BCD digits. The first digit has the following meaning:

0 Full OSI NSAP Address
1 Partial OSI NSAP Address
2 Non-OSI NSAP Address
3 Reserved by ISO

The remaining digits may take any value from 0 through 9.

rsrvd1

Reserved for future use. Set to 0.

rsrvd2

Reserved for future use. Set to 0.

rsrvd3

Reserved for future use. Set to 0.

count

INPUT. If greater than zero, specifies the number of incoming requests to be directed to the assigning process before automatic deassignment. If zero or –1, the port assignment remains in force until manually removed by XLUASN or X$UASN, or until the user calls X$CLRA or issues the ICE command. If –1, the assignment is placed after all previous assignments of the same class with a nonnegative value of *count*.

status

OUTPUT. The returned status of the call.

The following status codes may be returned by a call to X$ASGN or XLASGN:

XS$BPM	At least one of the parameters specified in the call is not in the legal range.

XS$CMP	The assign request has been successfully placed at the front of the assign queue for the specified port.
XS$MEM	No system resources are currently available for more assign requests.
XS$NET	Networks are not configured for this system.
XS$ILL	Either a privileged key has been supplied by a nonprivileged user or you have assigned an invalid port number.
XS$QUE	The assign request is behind at least one other identical request. For X$ASGN, this means that this port is behind request(s) by other users.

Subroutine: X$CONN X$SCON X$FCON XLCON

Description: Requesting a Call

Any of the call-requesting subroutines initiates the establishment of a virtual circuit. The application supplies a virtual circuit status array (the *vcstat* argument). The result of the call request is returned into this virtual circuit status array. Normally, this call request status changes with time as the call progresses. The call request subroutine also returns a virtual circuit ID (VCID) that is to be used with all subsequent IPCF subroutine calls related to this virtual circuit.

To make a call request, the caller must specify the target node. For short form routines, only configured node names can be used. The long form routine (XLCONN), however, permits numeric addresses as well as node names.

X$CONN, the short form routine, is intended to initiate a connection to another PRIMENET application that is executing on any other Prime node in the network.

X$SCON, an extended short form routine that permits a process on either end of a virtual circuit (as the receiver) to control the flow of data from the other end (the sender).

X$FCON, the short form for Fast Select, adds the capability of sending the call User Data field and retrieving called or clear user data returned by the callee.

XLCONN, the long form routine, allows you to specify in detail each of the fields in a call request packet. In addition, you can partially control which network paths are to be used, and retrieve returned user data fields. Refer to Appendix A, X.25 Programming Guidelines, for further information about X.25 facility fields.

Note

When PRIMENET is connected to a Packet Switched Data Network, the agency controlling that network may impose restrictions on the use of the fields in the Call Request packet.

Call syntax

```
CALL X$CONN(vcid, port, tadr, tadrn, vcstat)

CALL X$SCON(key, vcid, port, tadr, tadrn, vcstat)

CALL X$FCON(key, anskey, vcid, port, tadr, tadrn, udata, udatan,
            vcstat, rudat, rudatn, rudabc)

CALL XLCONN(key, vcid, port, tadr, tadrn, fcty, fctyn, prid, pridn,
            udata, udatan, vcstat [, rudat, rudatn, rudabc])
```

Arguments

key

INTEGER*2. Describes the form of the call and the physical paths to be allowed for the connection. The *key* has six parts:

1. Address format can be one of the following:

XK$NAM	*tadr* contains an ASCII PRIMENET node name. (This is the default, so XK$NAM may be omitted.)
XK$ADR	*tadr* contains an ASCII representation of the subscriber address.

2. Path specification

XK$ANY	Any available network path OK
XK$LAN	LAN300 path OK
XKK$PDN	Path through packet network OK
XK$RNG	RINGNET (PNC or PNC II) path OK
XK$RTE	Route-through (gateway) path OK
XK$SYN	Synchronous link path OK

Note

At least one path specifier *must* be present in the call.

3. Facility option request (used with XLCONN or X$SCON)

XK$FCT	A default facilities field will be provided by PRIMENET, appropriate for the connection type (RING, LAN300, or specific PSDN). Not specified by X$CONN.
0	A default facilities field will be specified separately by the application, or will not be used. (The facilities field might be added by your PSDN.)

Note

Do not use XK$FCT with X$FCON, since X$FCON supplies its own facility field. If you do use XK$FCT with X$FCON, the XS$BPM status code is returned.

4. Return data option (used only with XLCONN and X$FCON)

XK$RTD	This key indicates that the optional return data arguments for the Fast Select facility have been supplied.

5. Return extended Clear Packet option (used with XLCONN or X$SCON)

   XK$RXC — If this key is supplied, an incoming extended clear (that is, one that contains user data or facilities) will not be confirmed until the user calls a clearing routine, issues the ICE command, or logs out, or until 100 seconds have elapsed. The application may obtain the contents of an extended incoming Clear Packet by calling XLGI$.

6. The flow control option (used with X$SCON, X$FCON, or XLCONN)

   XK$FCW — If this key is supplied, it specifies that the flow control window should be opened only by the amount that the user has posted receive buffers.

anskey

INTEGER*2. For X$FCON (Fast Select calls) selects restricted response or not:

XK$ACC — The callee may accept or clear the call.

XK$CLR — The callee *must* clear the call.

vcid

INTEGER*2. Returned. The VCID to be used for this connection. Not valid when the XK$CLR *anskey* is used with X$FCON.

port

INTEGER*2. The port assigned by the process that is the target for this connection request. (Ignored if the *prid* argument is used.)

tadr

Array. Holds a string of bytes (char nonvarying in PL/I). This array contains the PRIMENET node name of the target node, with a maximum of 6 characters. For XLCONN, combined with XK$ADR, *tadr* holds the target node's subscriber address, with a maximum of 15 characters.

tadrn

INTEGER*2. The number of characters in *tadr*.

fcty

Array. Contains the bytes to go into the Call Request packet facilities field. (Ignored if *fctyn* is 0.)

fctyn

INTEGER*2. The number of bytes in *fcty*. Legal range is 0 through 109. (Must be 0 if XK$FCT is used.) For connections to 1980 PSDNs or pre-Rev. 21.0 Prime systems, the maximum legal value is 63.

prid

Array. A buffer that contains the four bytes to go into the Protocol ID field in the Call Request packet. (Ignored if *pridn* is 0.)

pridn

INTEGER*2. The number of bytes in *prid*. Legal values are 0 and 4.

Note

If *pridn* is 0, PRIMENET uses the Protocol ID field, *prid*, to pass the port specified in *port*. In this case, *prid* is used for a host-to-host protocol format defined for PRIMENET. If *pridn* is 4, the application-supplied bytes are used. In this case, the value specified in *port* is not used.

udata

Array. A buffer that holds the bytes to go into the User Data field in the Call Request packet. (Ignored if *udatan* is 0.)

udatan

INTEGER*2. The number of bytes in *udata*. Legal values are 0 through 12, except for Fast Select calls, for which the range is 0 through 124.

vcstat

Two-word array, INTEGER*2. Returned. Used for the virtual circuit status array. The list of returned virtual circuit status codes follows separately after the argument descriptions. PRIMENET may write into the *vcstat* argument during the whole life of the virtual circuit. Be sure *vcstat* is the correct data type.

The following three arguments are optional for X$FCON and XLCONN:

rudat

Array. To hold a string of bytes. Returned. This array receives returned User Data fields (if present) from Call Accept and Clear packets. *rudat* is intended for use with Fast Select calls. Note that the entire X.25 User Data field is returned here.

Notes

If both Call Accept and Call Clear user data are expected, use XLGA$ to fetch the Call Accept user data, since the clearing user data may overwrite *rudat* before the application has finished processing *rudat*. If *vcstat* is not the same before fetching the contents of *rudat* as after, then overwriting may have occurred.

Pre-Rev. 21.0 Prime systems and 1980 PSDNs do not support use of clear user data on established circuits.

rudatn

INTEGER*2. The maximum number of characters to be returned into *rudat*.

rudabc

INTEGER*2. Returned. The actual number of characters returned into *rudat*.

Note

rudat and *rudabc* are only valid when the Call Request has completed, that is, when *vcstat* equals either XS$CMP or XS$CLR.

The following codes can be returned into the first word of the virtual circuit status array *vcstat*. These codes may be returned immediately on return from the call request subroutine.

XS$BPM	One of the arguments to the call is missing, out of range, or in conflict with other arguments.
XS$DWN	The target node specified in *tadr* is currently unavailable through PRIMENET.
XS$FCT	Bad facility field supplied (XLCONN only).
XS$IP	The connection request (or data transfer) has been successfully initiated. See XS$CMP.
XS$MAX	This request exceeds the maximum number of virtual circuits allowed. This error occurs if you run out of resources on the source node, or when a PSDN limits then number of circuits allowed.
XS$MEM	There is temporary buffer congestion in the local network. The system currently does not have the resources required to process the request. Retry the request later.
XS$NET	PRIMENET is not configured on this Prime system.
XS$UNK	The target node specified in *tadr* is unknown (not configured) in the network.

As a result of data transfers (network and remote user actions), the following codes may be returned later:

XS$CLR	The connection has been cleared and is no longer usable. If the connection was cleared by the network or the remote process (rather than the local process), the second word of the virtual circuit status array holds the clearing cause and diagnostic code. (Refer to Chapter 2, Ports and Virtual Circuits, for information about clearing causes and diagnostic codes.)
XS$CMP	The connection attempt or a data transfer has successfully completed.

XS$MAX	This request exceeds the maximum number of virtual circuits allowed. This error occurs if you run out of resources on the source node, or when a PSDN limits the number of circuits allowed.
XS$RST	The virtual circuit has been reset. All operations in progress have been aborted.

Subroutine: X$GCON X$FGCN XLGCON

Description: Finding Information About an Incoming Call

These subroutines return information about Incoming Call packets from the receiver's point of view. Once a process has assigned a port, PRIMENET places all incoming call requests that specify that port on a call request queue for the process. Because each PRIMENET process has only one such queue, when a process has several ports assigned, the connection requests for each of them are placed on this same queue in a first in/first out fashion.

A call to X$GCON, X$FGCN, or XLGCON copies information about the first call request on the process's queue without dropping the request from the queue. The process should then dispose of the pending connection by either accepting or clearing the call. Either action drops the pending request from the call request queue. Requests not handled by the process in 100 seconds are automatically cleared by PRIMENET. X$GCON is the short form subroutine, primarily for use between Prime nodes.

X$FGCN is the short form subroutine for retrieving information for Fast Select calls. It gives the name of the caller's node and the call User Data field. It also tells if the caller required restricted response or not. (Fast Select calls that require restricted response must be cleared.)

XLGCON allows the caller to extract almost all of the fields in an X.25 Call Request packet.

Call syntax

```
CALL X$GCON(vcid, port, status)

CALL X$FGCN(key, anskey, vcid, port, fadr, fadrn, fadrbc,
            udata, udatan, udatbc, status)

CALL XLGCON(key, vcid, port, fadr, fadrn, fadrbc, fcty, fctyn,
            fctybc, prid, pridn, pridbc, udata, udatan,
            udatbc, status)
```

Arguments

key

INTEGER*2. X$FGCN and XLGCON: Describes the format of the calling node name to be placed into *fadr*.

XK$NAM	*fadr* will receive the ASCII PRIMENET node name.
XK$ADR	*fadr* will receive the ASCII subscriber address.

anskey

INTEGER*2. Returned. For X$FGCN (Fast Select calls), selects restricted response or not.

XS$ACC	The callee may accept or clear the call.
XS$CLR	The callee *must* clear the call.
XS$NOT	This is *not* a Fast Select call.

vcid

INTEGER*2. Returned. This *vcid* is to be used for all subsequent IPCF calls relating to this virtual circuit.

port

INTEGER*2. Returned. The port to which this call request is directed.

fadr

Array. Holds a string of bytes (char nonvarying in PL/I). Returned. Receives either the PRIMENET node name or the system address for the node at which this call originated.

fadrn

INTEGER*2. The maximum number of bytes *fadr* may receive.

fadrbc

INTEGER*2. Returned. The number of bytes returned into *fadr*.

fcty

Array. Returned. Holds a string of bytes (char nonvarying in PL/I). Receives a copy of the call request packet facilities field. (Ignored if *fctyn* is 0.)

fctyn

INTEGER*2. The maximum number of bytes the *fcty* buffer may receive.

fctybc

INTEGER*2. Returned. The number of bytes returned into *fcty*.

prid

Array. Returned. Holds a string of bytes (char nonvarying in PL/I). Receives the bytes from the call request packet Protocol ID field. (Ignored if *pridn* is 0.)

pridn

INTEGER*2. The maximum number of bytes *prid* may receive.

pridbc

INTEGER*2. Returned. The number of bytes returned into *prid*.

udata

Array. Returned. Holds a string of bytes (char nonvarying in PL/I). Receives the bytes from the user data field of the call request packet. (Ignored if *udatan* is 0.)

udatan

INTEGER*2. The maximum number of bytes *udata* may receive.

udatbc

INTEGER*2. Returned. The number of bytes returned into *udata.*

status

Two-word array, INTEGER*2. Returned. Contains the returned status of the call.

The following status codes may be returned in the first status word.

XS$NOP	No call requests pending.
XS$CMP	Pending call request: return arguments are valid.
XS$BPM	Invalid key argument in the call.
XS$NET	Networks not configured.

The *second* status word is valid only when the first word has the value XS$CMP. The second word may have a value of either 1 or 2. These codes are defined below.

1. A new incoming call request.
2. A transferred virtual circuit; refer to the section on X$GVVC/XLGVVC.

Subroutine: XLGC$

Description: Finding Information About an Incoming Call

Like XLGCON, XLGC$ returns the contents of an Incoming Call packet received from the remote node. However, XLGC$ has a key, XK$REG, that allows the application to mark a given call as "seen" so that subsequent calls to XLGC$, XLGCON, X$GCON, or X$FGCN skip that call and return information on the next call in the queue.

Call syntax

```
CALL XLGC$(key, vcid, port, gfi, vcn, cmnd, fadr, fadrn,
           fadrbc, tadr, tadrn, tadrbc, fcty, fctyn, fctybc,
           prid, pridn, pridbc, udata, udatan, udatbc, status)
```

Arguments

key

INTEGER*2. Set to one of the following:

0	Return details of the first call for this user that has not yet been either answered or marked as already seen. Do not mark the call.
XK$REG	Return details of the first call for this user that has not yet been either answered or marked as already seen. Mark the call as seen.
XK$SAV	Same as XK$REG, but record *status* as the status vector for this circuit until an accept call is made. This key is useful for programmers writing servers.

Note

The status vector exists only from the time when XLGC$ is called with the XK$SAV key until the time when a call is made to accept a connection request through either X$ACPT, X$FACP, X$SACP, or XLACPT.

vcid

INTEGER*2. Returned. VCID for this circuit.

port

INTEGER*2. Returned. PRIMENET port number for this circuit.

gfi

INTEGER*2. Returned. X.25 GFI for this packet.

vcn

INTEGER*2. Returned. X.25 logical channel number for this circuit.

cmnd

INTEGER*2. Returned. X.25 command byte for this packet (always 11).

fadr

Array. Returned. Holds a string of bytes (char nonvarying in PL/I). Will contain the subscriber address for the node at which the call originated.

fadrn

INTEGER*2. Maximum number of bytes *fadr* may receive. Addresses are a maximum of 15 bytes in length.

fadrbc

INTEGER*2. Returned. The number of bytes returned into *fadr*.

tadr

Array. Returned. Holds a string of bytes (char nonvarying in PL/I). Will contain the address for the node to which the call is directed.

tadrn

INTEGER*2. Maximum number of bytes *tadr* may receive. Addresses are a maximum of 15 bytes in length.

tadrbc

INTEGER*2. Returned. The number of bytes returned into *tadr*.

fcty

Array. Returned. Holds a string of bytes (char nonvarying in PL/I). Receives a copy of the Incoming Call facilities field. (Ignored if *fctyn* is 0).

fctyn

INTEGER*2. Maximum number of bytes that the *fcty* buffer may receive. The legal maximum for facility fields is 109 bytes (or 63 bytes, for calls to 1980 PSDNs or pre-Rev. 21.0 Prime systems).

fctybc

INTEGER*2. Returned. The number of bytes returned into *fcty*.

prid

Array. Returned. Holds a string of bytes (char nonvarying in PL/I). Receives the bytes from the call request packet Protocol ID field. (Ignored if *pridn* is 0.)

pridn

INTEGER*2. 0 or 4. The maximum number of bytes *prid* may receive. Incoming Call packets may have at most 128 bytes of User Data. Four bytes of User Data (the Protocol ID) are copied into *prid*; the remaining bytes are copied into *udata*.

pridbc

INTEGER*2. Returned. The number of bytes copied into *prid*.

udata

Array. Returned. Holds a string of bytes (char nonvarying in PL/I). Receives the bytes from the user data field of the call request packet. (Ignored if *udatan* is 0.)

udatan

INTEGER*2. 0 to 124. The maximum number of bytes *udata* may receive.

udatbc

INTEGER*2. Returned. The number of bytes of user data copied into *udata*.

status

INTEGER*2. Returned. Normally contains the immediate return status of the call. If the XK$SAV key is used, *status* is a two-word vector used to return circuit status dynamically until an accept call is made.

The following status codes may be returned in the first status word.

XS$NOP	No call requests pending.
XS$CMP	Pending call request: return arguments are valid.
XS$BPM	Invalid key argument in the call.
XS$NET	Networks not configured.

The *second* status word is valid only when the first word has the value XS$CMP. The second word may have a value of either 1 or 2. These codes are defined below.

1. A new incoming call request.
2. A transferred virtual circuit; refer to the section on X$GVVC/XLGVVC.

Subroutine: X$ACPT X$FACP X$SACP XLACPT

Description: Accepting a Call

After identifying a caller through a "get connection data" subroutine (X$GCON, X$FGCN, XLGCON, or XLGC$), the called application either accepts or clears the connection. Any of the call accept subroutines can be used to accept a connection request and complete the connection. The status of the call is returned in the *vcstat* array. This virtual circuit status array is used throughout the life of the virtual circuit.

X$ACPT is the short form of the subroutine, primarily for use between Prime nodes.

X$FACP is the short form of the subroutine intended for Fast Select call accepts, in which returned user data is transferred within the Call Accept packet.

X$SACP is the short form of the subroutine that incorporates the *key* argument from the long form.

XLACPT allows you to specify in detail the X.25 packet-level protocol Call Accept packet. This long form may contain facilities, Protocol ID, and/or User Data fields.

Notes

When PRIMENET is connected to a Packet Switched Data Network, the agency controlling that network may impose restrictions on the use of the fields in the Call Accept packet.

The X.25 protocol only permits the transfer of called user data with Fast Select calls.

Call syntax

```
CALL X$ACPT(vcid, vcstat)

CALL X$FACP(vcid, udata, udatan, vcstat)

CALL X$SACP(key, vcid, udata, udatan, vcstat)

CALL XLACPT(key, vcid, fcty, fctyn, prid, pridn, udata, udatan,
            vcstat, rudat, rudatn, rudabc)
```

Arguments

key

INTEGER*2. Normally the key is zero, in which case the user may (but need not) supply facilities. Four other parts are supported:

XK$FCT	Indicates that PRIMENET should supply facilities appropriate to the particular virtual circuit.
XK$RTD	Indicates that clear user data may be expected from the initiator of the circuit.

XK$RXC	Indicates that extended Clear packets should be made available to the user, who can retrieve them by calling XLGI$.
XK$FCW	Indicates that the flow control window should be opened only by the amount that the user has posted receive buffers.

vcid

INTEGER*2. The virtual circuit ID for this circuit. This value was obtained by a preceding call to X$GCON, X$FGCN, XLGCON, or XLGC$. The value can be 1 through 255.

fcty

Array. Contains the bytes to go into the Call Accept packet facilities field. (Ignored if *fctyn* is 0.) Must not be used when *key* is set to XK$FCT.

fctyn

INTEGER*2. The number of bytes to be taken from *fcty*. Legal range is 0 to 109. (Must be 0 if XK$FCT is used.) For connections to X.25 1980 PSDNs or pre-Rev. 21.0 Prime systems, the maximum legal value is 63.

prid

Array. A buffer that contains the four bytes to go into the Call Accept packet Protocol ID field. (Ignored if *pridn* is 0.) If *udatan* is greater than zero and *prid* is not supplied by the application, PRIMENET sets a default format for *prid*.

pridn

INTEGER*2. The number of bytes to be taken from *prid*. Legal values are 0 and 4. (The value should be 0 for non-fast-select accepts through a PSDN.)

udata

Array. A buffer that holds the bytes to go into the User Data field of the Call Accept packet. (Ignored if *udatan* is 0.)

udatan

INTEGER*2. The number of bytes to be taken from *udata*. Legal values are 0 to 12, except for accepts on Fast Select calls, when the range is 0 to 124. (The value should be 0 for calls that are not Fast Select through a PSDN.)

vcstat

Two-word array, INTEGER*2. Returned. Used for the virtual circuit status array. PRIMENET may write into the *vcstat* argument during the whole life of the virtual circuit. Be sure *vcstat* is the correct data type.

The following status codes may be returned in the first word of *vcstat*.

XS$BPM	Invalid arguments in the call (X$FACP and XLACPT only).
XS$BVC	The calling process does not control the virtual circuit specified by *vcid*.
XS$FCT	Bad facility field supplied (XLACPT only).
XS$IDL	The operation was successful and the virtual circuit is now idle, awaiting data traffic.
XS$ILL	The process tried to accept a virtual circuit that was not in the call request pending state, or tried to accept a call set up for Fast Select with restricted response.
XS$MEM	There is temporary buffer congestion in the local PRIMENET node. Retry the accept several seconds later.

If there is a need later to extract the User Data field from a remotely originated clear request, then you must specify the key XK$RTD and supply the following additional arguments.

rudat

Array. To hold a string of bytes. Returned. This array receives returned User Data fields (if present) from Clear packets. Intended for use with Fast Select calls. Note that the entire X.25 User Data field is returned.

Note

Pre-Rev. 21.0 Prime systems and 1980 PSDNs do not support use of clear user data on established circuits.

rudatn

INTEGER*2. The maximum number of characters to be returned into *rudat*.

rudabc

INTEGER*2. Returned. The actual number of characters returned into *rudat*.

As a result of data transfers, actions in the network, or remote user actions, the following codes may be returned later:

XS$CLR	The connection has been cleared. The second word of the virtual circuit status array is now the valid clearing cause or diagnostic code.
XS$CMP	A data transfer operation has completed successfully.
XS$RST	The virtual circuit has been reset. All operations in progress have been aborted.

Subroutine: XLGA$

Description: Finding Information About a Connected Call

XLGA$ returns the contents of a Call Connected packet received from the remote node in response to an outgoing Call Request. Call this routine only when the status array set up by XLCONN has been set to XS$CMP and before calling X$TRAN.

Call syntax

```
CALL XLGA$(key, vcid, port, gfi, vcn, cmnd, fadr, fadrn,
           fadrbc, tadr, tadrn, tadrbc, fcty, fctyn, fctybc,
           prid, pridn, pridbc, udata, udatan, udatbc, status)
```

Arguments

key

INTEGER*2. Unused. Set to 0.

vcid

INTEGER*2. VCID for this circuit.

port

INTEGER*2. PRIMENET port number for this circuit.

gfi

INTEGER*2. X.25 GFI for this packet.

vcn

INTEGER*2. X.25 logical channel number for this circuit. This is not related to the VCID.

cmnd

INTEGER*2. X.25 command byte for this packet (always 15).

fadr

Array. Returned. Holds a string of bytes (char nonvarying in PL/I). Will contain the subscriber address for the node at which the call originated.

fadrn

INTEGER*2. Maximum number of bytes *fadr* may receive. Addresses are a maximum of 15 bytes in length.

fadrbc

INTEGER*2. Returned. The number of bytes returned into *fadr*.

tadr

Array. Returned. Holds a string of bytes (char nonvarying in PL/I). Will contain the address for the called node.

tadrn

INTEGER*2. Maximum number of bytes *tadr* may receive. Addresses are a maximum of 15 bytes in length.

tadrbc

INTEGER*2. Returned. The number of bytes returned into *tadr*.

fcty

Array. Returned. Holds a string of bytes (char nonvarying in PL/I). Will receive a copy of the Call Accept facilities field. (Ignored if *fctyn* is 0.)

fctyn

INTEGER*2. Maximum number of bytes the *fcty* buffer may receive. The legal maximum for facility fields is 109 bytes (or 63 bytes, for calls to 1980 PSDNs or pre-Rev. 21.0 Prime systems).

fctybc

INTEGER*2. Returned. The number of bytes returned into *fcty*.

prid

Array. Returned. Receives the first four bytes of the User Data field from the Call Accept packet. (Ignored if *pridn* is 0.)

pridn

INTEGER*2. 0 to 4. The maximum number of bytes *prid* may receive. Call Accept packets may have at most 128 bytes of user data. Four bytes of User Data (the Protocol ID) are copied into *prid*; the remaining bytes are copied into *udata*.

pridbc

INTEGER*2. Returned. The number of bytes copied into *prid*.

udata

Array. Returned. The buffer that will receive the User Data field from the Call Accept packet. (Ignored if *udatan* is 0.)

udatan

INTEGER*2. 0 to 124. The maximum number of bytes *udata* may receive.

udatbc

INTEGER*2. Returned. The number of bytes of user data copied into *udata*.

status

INTEGER*2. Returned. Contains the immediate return status of the call.

The following status codes may be returned in the status word:

XS$CMP	Operation complete.
XS$BVC	User does not own the virtual circuit specified by *vcid*.
XS$ILL	The circuit is not in the call established state. Either the virtual call has not yet been accepted, or the virtual call has been cleared, or the user has already transmitted some data using X$TRAN.
XS$NET	Networks are not currently running.

Subroutine: X$TRAN

Description: Transmitting Data

X$TRAN is the IPCF transmit-message subroutine. An application calls X$TRAN to send the contents of a buffer through the network to the process on the opposite end of a virtual circuit. PRIMENET automatically splits the message into X.25 packets of appropriate size for transmission, and then recombines the packets at the receiving end.

PRIMENET supports X.25's two data levels and interrupt procedure. When applications call X$TRAN, they supply an argument *level* with one of three values (the SYSCOM>X$KEYS.INS.XXX files have defined mnemonics for each value) to indicate one of the following:

- A message (data packet sequence): Q-bit set to 0 (XT$LV0)
- A message (data packet sequence): Q-bit set to 1 (XT$LV1)
- An interrupt packet (XT$INT)

Both XT$LV0 and XT$LV1 are requests to move up to 32,767 (32K) bytes of data. The only difference between them is their data level. PRIMENET passes the data level transparently through the circuit to the receiver so that an application may distinguish between data messages with the Q-bit values set differently. PRIMENET treats XT$LV0 and XT$LV1 data packets the same way, handling data transmission requests of both types in a single queue in first in/first out fashion.

Interrupt packets (XT$INT) are handled separately, in compliance with the X.25 packet-level protocol. Each can carry up to 32 bytes of data; however, only a single byte of data is allowed over links to pre-Rev. 21.0 systems or to PSDNs not using X.25 1984 protocol. The interrupt packet is placed at the top of the queue ahead of all ordinary data packets. As a result, an interrupt packet may arrive at its destination earlier than normal data sent before it. For the effect of interrupts on the receiving side, see X$RCV.

Call syntax

```
CALL X$TRAN(vcid, level, buffer, bufbc, status)
```

Arguments

vcid

INTEGER*2. The VCID for this circuit.

level

INTEGER*2. The data level of this message is

XT$LV0 (0) For normal data packets with the X.25 Q-bit set to 0

XT$LV1 (1) For normal data packets with the X.25 Q-bit set to 1

XT$INT (2) For an interrupt packet

buffer

Any array. The data buffer to be moved through the virtual circuit. The buffer must *not* cross a segment boundary.

bufbc

INTEGER*2. The number of bytes to copy from *buffer*. *bufbc* is 0 to 32767 except for Interrupt packets. For Interrupt packets, *bufbc* is 1 to 32, except over links to pre-Rev. 21.0 systems or 1980 PSDNs, in which case *bufbc* must 1.

status

INTEGER*2. Returned. The status of this transmit.

The following codes can occur in *status* immediately on return from X$TRAN:

XS$BPM	The call contains invalid arguments.
XS$BVC	The calling process does not control the virtual circuit specified in *vcid.*
XS$ILL	The transmit operation is illegal because a circuit connection request or a clear request is pending. This error is the result of attempting transmission over an "almost-open" or "almost-closed" circuit.
XS$IP	The transmit is in progress. *status* will be further updated upon the completion or failure of the operation. This is the normal immediate return code from X$TRAN.
XS$MAX	This request to initiate a new transmission is denied because the virtual circuit is already carrying the maximum number of transmissions that can be in progress simultaneously over a single virtual circuit.
XS$MEM	There is temporary PRIMENET buffer congestion on your local node that prevents the acceptance of the transmit request at this time.

The following codes can occur in *status* immediately (especially in the case of loopback, where both ends of the virtual circuit are on the same system), or can occur at a later time:

XS$CLR	The virtual circuit has been cleared. Check the virtual circuit status array to find the clearing cause and diagnostic code.
XS$CMP	The transmit is complete. The message has been copied out of the sender's buffer and transmission has been initiated. (A transmit status of complete means only that PRIMENET will attempt to deliver the message. Applications requiring assured delivery must implement their own end-to-end acknowledgment in a higher-level protocol of their own.)
XS$RST	The virtual circuit has been reset. The status of this transmit request is unknown and no further attempts will be made to complete it.

Subroutine: X$RCV

Description: Receiving Data

X$RCV is the IPCF receive-message subroutine. An application offers a buffer into which PRIMENET places received messages from the specified virtual circuit. The application receiving a message should establish its receive buffer before the sending application attempts to transmit data.

In all cases, data messages transmitted through the X$TRAN subroutine are reproduced identically in the receive buffer. However, three special cases are worthy of note: mismatched buffer sizes, interrupt handling, and circuit resets.

Mismatched Buffer Sizes: The simplest case is a receive buffer that is the same size as or bigger than an incoming data message. In such a case, the receive status is set to indicate a completed receive as soon as the entire message is copied into the receive buffer.

When the receive buffer is too small to contain an incoming message, the specified buffer is filled, the receive status is set to indicate a completed operation, and the remainder of the message is held until another buffer, presented by another call to X$RCV, is available. PRIMENET attempts to complete the delivery of the message; if necessary, it fills the second buffer and again holds the remainder. This process continues until the complete message is copied into the receiver's buffers.

Interrupt Handling: A process may send an interrupt across a virtual circuit even when regular data transmissions are in progress. (See X$TRAN.) Because an interrupt may pass normal data moving in the network, a partially completed receive may be interrupted by such an interrupt message. An application receives interrupts in the same way it receives regular data with X$RCV.

If a receive buffer offered in a call to X$RCV is partially filled with level 0 or level 1 data when an interrupt message is received, the following actions take place. The status word of the X$RCV request that is currently being filled is marked as completed, even if the receive buffer is not yet completely filled. The *next* pending call to X$RCV serves to receive the interrupt, and the call to X$RCV after that receives the remainder of the original message. No data loss results in this exchange, but the original data message is broken into two pieces, so that an extra call to X$RCV is needed. The receiving process should always inspect the returned status code to find out the level of received data. When the status code indicates that an interrupt has occurred, the interrupt-handling code should issue extra calls to XS$RCV as needed.

Since interrupt packets can be as many as 32 bytes long, a receive buffer that is smaller than 32 bytes could result in the loss of part of an interrupt packet. However, even if part of the interrupt packet is lost, the *status* array still reports the presence of an interrupt.

Circuit Resets: Circuit resets can result in loss of data because packets that are traveling in the network may disappear. In most cases, you can detect a reset because uncompleted receive requests return the status code XS$RST rather than XS$CMP. However, you may not be able to detect a reset that occurs after the transmitting program sends data by means of X$TRAN but before your program issues the corresponding call to X$RCV. No data is received, but X$RCV

does not know that a reset has occurred, and thus does not return a status of XS$RST. The only way to detect the reset in this case is to check the virtual circuit status array. You can follow either of two procedures to avoid this problem:

- Be sure the receiving process calls X$RCV and establishes the receive buffer *before* the transmitting process calls X$TRAN. This practice ensures that if a reset occurs after the data are sent, the pending X$RCV call returns a status of XS$RST.
- Establish data flow checkpoints so that data can be retransmitted, rather than lost, in the event of a reset.

Call syntax

CALL X$RCV(vcid, buffer, bufn, status)

Arguments

vcid

INTEGER*2. The VCID for this circuit.

buffer

Any array. Returned. The data buffer into which incoming data should be moved. The buffer *must not* cross a segment boundary.

bufn

INTEGER*2. The maximum number of bytes that may be moved into *buffer*.

status

Three-word array, INTEGER*2. Returned.

The *first* word is the receive request status word.

The *second* word is set to the level of the incoming data (XT$LV0, XT$LV1, or XT$INT).

The *third* word is set to the number of bytes moved into *buffer*.

The following codes can occur in the first word of *status* immediately on return from X$RCV:

XS$BPM	The call contains invalid arguments.
XS$BVC	The calling process does not control the virtual circuit specified in *vcid*.
XS$ILL	The receive operation is illegal because a circuit connection request or a clear request is pending. This error is the result of setting up a receive on an "almost-open" or "almost-closed" circuit.
XS$IP	The receive is in progress. *status* will be further updated upon the completion or failure of the operation. This is the normal immediate return code from X$RCV.

XS$MAX	This request to initiate a new receive is denied because the virtual circuit is already carrying the maximum number of receives that can be in progress simultaneously over a single virtual circuit.
XS$MEM	There is temporary PRIMENET buffer congestion on your local node that prevents the acceptance of the receive request at this time.

The following codes can occur in *status* immediately (especially in the case of loopback, where both ends of the virtual circuit are on the same system), or can occur at a later time:

XS$CLR	The virtual circuit has been cleared. Check the virtual circuit status array to find the clearing cause and diagnostic code.
XS$CMP	The receive is complete. The incoming data have been moved to *buffer*, and the second and third words of *status* are updated.
XS$RST	The virtual circuit has been reset. The status of this operation is unknown and no further attempts will be made to complete it.

Subroutine: X$CLR X$FCLR XLCLR

Description: Clearing a Call

At any time during the life of a virtual circuit, an application may clear (break) the circuit by calling one of the clearing subroutines. These subroutines disconnect a virtual circuit by initiating transmission of a Clear Request packet.

To use the clearing subroutines, an application must already have received the virtual circuit ID number (VCID) of the circuit to be cleared. (The VCID would have been passed to the application through a call to X$CONN/X$SCON/X$FCON/XLCONN or X$GCON/X$FGCN/XLGCON/XLGC$.) A call to XCLR/XFCLR/XLCLR specifying the VCID in question cancels all activities in progress, releases any resources, and notifies the process on the other end of the circuit that a clear has been requested.

A called user may request a clear, rather than accepting a call, immediately after receiving a call request. The called party may return one diagnostic byte giving the reason for the clear request. In the case of a Fast Select call, the clear request may contain a User Data field of as many as 128 bytes. An accepted Fast Select call may be cleared with user data by either the caller or the called party.

The actual clearing process happens in two stages. First, the call to the clearing subroutine initiates transmission of a Clear Request packet to the other end of the virtual circuit. To indicate this stage, the clear subroutine returns a *status* code immediately. If the call is successful, all transmit and receive requests in progress on the virtual circuit in question are immediately aborted, and their status codes are changed to XS$CLR.

Secondly, when the remote system receives the clear request, it answers by transmitting back a clear confirmation message. Only when the clear confirmation has been returned successfully does the clear requesting caller see the circuit as cleared in the first word of the virtual circuit status array (*vcstat*). The second word of the virtual circuit status array is invalid in this case. Normally, if the clear was requested before the virtual circuit was accepted, there is no virtual circuit array defined; therefore, the application cannot detect the confirmation. However, if you used the XK$SAV option in a call to XLGC$ you will have the status of the circuit.

When the remote process initiates the clear request, the local PRIMENET process (NETMAN) automatically sends the clear confirmation, aborts all data transfer operations in progress, and sets the first word of the virtual circuit status array to the circuit cleared value (XS$CLR). The second word is set to the valid clearing cause and diagnostic code. These codes are described in Chapter 2, Port Assignments and Virtual Circuits, and listed in Appendix C, Clearing Causes and Diagnostic Codes.

Note

The delay for the clear confirm might be noticeable, especially over long-distance PSDN links. The application that requests a clear should wait until the XS$CLR status has been returned into the virtual circuit status array before returning to PRIMOS (with CALL EXIT). You can also call X$CLRA immediately if you are unable to wait for the XS$CLR status. However, calling X$CLRA has the effect of wiping out ALL virtual circuits, including those created at lower command levels.

X$CLR is the short form clearing routine, primarily intended for use between Prime nodes.

X$FCLR is specifically intended for the clearing of Fast Select calls, when the application wants to return clear user data.

XLCLR, the long form clearing subroutine, allows you to include X.25 facilities when clearing a call.

Note

The Clear User Data field's value is given by a single argument, *clrudat*, that corresponds to the concatenated arguments *prid* and *udata* of the X$FACP/XLACPT routines.

Call syntax

```
CALL X$CLR(vcid, why, status)

CALL X$FCLR(vcid, why, clrudat, clrudatn, status)

CALL XLCLR(key, vcid, why, fcty, fctyn, clrudat, clrudatn,
           rsrvd1, status)
```

Arguments

key

INTEGER*2. Set to 1.

vcid

INTEGER*2. The VCID of the circuit to be cleared.

why

INTEGER*2. The low-order byte of *why* may take on values from 0 through 255 and is taken as the diagnostic code. (Refer to Chapter 2, Port Assignments and Virtual Circuits, for information about diagnostic codes.) The high-order byte of *why* is usually 0, or may exceptionally take on a value from 128 through 255.

fcty

Array. Contains the bytes to go into the Clear Request packet facilities field. (Ignored if *fctyn* is 0.)

fctyn

INTEGER*2. The number of bytes to be taken from *fcty*. Legal range is 0 to 109. For connections to X.25 1980 PSDNs or pre-Rev. 21.0 Prime systems, the maximum legal value is 63.

clrudat

Array. A buffer that holds the bytes to go into the User Data field of the Clear Request packet. (Ignored if *clrudatn* is 0.)

clrudatn

INTEGER*2. The number of bytes to be taken from *clrudat*. The legal value is 0, except following Fast Select calls, when the range is 0 through 128.

rsrvd1

INTEGER*2. Reserved for future use. Set to 0.

status

INTEGER*2. Returned. Contains the immediate return status of the call.

The following status codes may be returned.

XS$BPM	One of the arguments to the call has an illegal value.
XS$BVC	The calling process does not control the virtual circuit specified by the *vcid.*
XS$CLR	It was not possible to update the virtual circuit status array or the status vector associated with the user's pending operation. However, the circuit has been cleared.
XS$CMP	The operation is successful. All pending transmits and receives are aborted with a status of XS$CLR.
XS$FCT	Bad facility field supplied. (XLCLR only.)
XS$ILL	A clear with user data has been requested for a Fast Select virtual circuit, but it does not immediately follow the call request. Also may indicate that you tried to clear a circuit that is not in a state to be cleared.
XS$MEM	(Occurs only if user data or facilities are present.) There is temporary buffer congestion in the local network. The system currently does not have the resources required to process the request. Retry the request later, unless you called X$CLR, which uses a built-in memory mechanism to initiate the clear request later.

Subroutine: XLGI$

Description: Finding Information About a Cleared Call

XLGI$ returns the contents of an extended Clear Indication packet (that is, a Clear Indication packet containing facilities or user data) if the user requested that extended Clear packets be held. (This request is made by calling XLCONN or XLACPT with the XK$RXC key.) Call this routine only after the status vector passed to XLCONN or XLACPT has changed to XS$CLR. After the packet has been read, call X$CLR to release the virtual circuit.

Call syntax

```
CALL XLGI$(key, vcid, port, gfi, vcn, cmnd, fadr, fadrn,
           fadrbc, tadr, tadrn, tadrbc, fcty, fctyn, fctybc,
           prid, pridn, pridbc, udata, udatan, udatbc, status)
```

Arguments

key

INTEGER*2. Unused. Should be 0.

vcid

INTEGER*2. Returned. VCID for this circuit.

port

INTEGER*2. Returned. PRIMENET port number for this circuit.

gfi

INTEGER*2. Returned. X.25 GFI for this packet.

vcn

INTEGER*2. Returned. X.25 logical channel number for this circuit.

cmnd

INTEGER*2. Returned. X.25 command byte for this packet (always 19).

fadr

Array. Returned. Holds a string of bytes (char nonvarying in PL/I). Usually null; may contain the subscriber address for the node at which the call originated.

fadrn

INTEGER*2. Maximum number of bytes *fadr* may receive. Addresses are a maximum of 15 bytes in length.

fadrbc

INTEGER*2. Returned. The number of bytes returned into *fadr*. This argument normally will be 0 on return.

tadr

Array. Returned. Holds a string of bytes (char nonvarying in PL/I). Usually null; may contain the address for the node at which the call was received.

tadrn

INTEGER*2. Maximum number of bytes *tadr* may receive. Addresses are a maximum of 15 bytes in length.

tadrbc

INTEGER*2. Returned. The number of bytes returned into *tadr*. Normally this argument will be 0 on return.

fcty

Array. Returned. Holds a string of bytes (char nonvarying in PL/I). Receives a copy of the clear indication facilities field. (Ignored if *fctyn* is 0.)

fctyn

INTEGER*2. Maximum number of bytes *fcty* may receive. The legal maximum for facility fields is 109 bytes (or 0 for calls to 1980 PSDNs or pre-Rev. 21.0 Prime systems).

fctybc

INTEGER*2. Returned. The number of bytes returned into *fcty*.

prid

Array. Returned. Receives the first four bytes of the User Data field from the Clear Indication packet. (Ignored if *pridn* is 0.)

pridn

INTEGER*2. 0 to 4. The maximum number of bytes *prid* may receive. Clear Indication packets may have at most 128 bytes of user data. Four bytes of user data (the Protocol ID) are copied into *prid*; the remaining bytes are copied into *udata*.

pridbc

INTEGER*2. Returned. The number of bytes copied into *prid*.

udata

Array. Returned. Receives the User Data field from the Clear Indication packet. (Ignored if *udatan* is 0.)

udatan

INTEGER*2. 0 to 124. The maximum number of bytes *udata* may receive.

udatbc

INTEGER*2. Returned. The number of bytes of user data copied into *udata*.

status

INTEGER*2. Returned. Contains the immediate return status of the call.
The following status codes may be returned in the status word:

XS$CMP	Operation complete.
XS$NOP	No call requests pending.
XS$BPM	Invalid key argument in call.
XS$NET	Networks are not currently running.

Subroutine: X$UASN XLUASN

Description: Deassigning a Port

X$UASN and XLUASN are used to deassign ports — that is, to remove an application from the assignment queues of currently assigned ports. At any time, an application may deassign any or all of the ports assigned by it. The application's assign request for the specified port is immediately deleted from the assignment queue regardless of its position in the queue.

If the value of the specified *port* is < 0, *all* of the application's port assignment requests are dropped from the assignment queues.

The X$UASN subroutine is adequate for most port deassignments. The XLUASN subroutine is intended for use in writing servers that will be privileged processes. (Privileged processes are defined in the section describing X$ASGN/XLASGN.) To remove an extended assignment previously created with XLASGN, call XLUASN with the same arguments used in the XLASGN call, with the exception of *count*.

This operation always completes successfully. If the port passed in the call is not assigned at the time of the X$UASN or XLUASN call, no action is taken. The call syntax for XLUASN follows the call syntax for X$UASN.

Call syntax

```
CALL X$UASN(port)
```

Arguments

port

INTEGER*2. (INPUT). The number of the port to be deassigned. 0 through 99 for unprivileged processes; 0 through 255 for privileged processes. –1 causes all ports to be deassigned.

Note

The calling sequence for XLUASN is inconsistent with the other IPCF routines. The arguments that are normally FORTRAN arrays of characters or PL/I character nonvarying strings are PL/I character varying strings for XLUASN.

To call XLUASN from FORTRAN, you must build a data structure that is interpreted as a character varying string, that is, a 16-bit length field followed by an array of characters. For example, the *prid* argument is defined as CHAR(4)VAR. The equivalent FORTRAN declaration is INTEGER*2 PRID(3), with PRID(1) set = 4, and PRID(2) and PRID(3) holding the 4 bytes of the *prid* field. To pass a value of 999912341234 in the *tadr* field, declare INTEGER*2 TADR(9), and set TADR(1) = 12 and put the characters 999912341234 in TADR(2) through TADR(7).

Call syntax

```
DCL XLUASN ENTRY(BIT(16), CHAR(15)VARY, CHAR(15)VAR, CHAR(4)VAR,
          CHAR(128)VAR, FIXED BIN(15), CHAR(41)VAR, FIXED BIN(15),
          FIXED BIN(15), FIXED BIN(15), FIXED BIN(15));

CALL XLUASN ENTRY(key, tadr, tsadr, prid, udata, port, txadr, rsvd1,
          rsvd2, rsvd3, status);
```

Arguments

key

INPUT. Structure as defined below, an overlay for bit(16).

```
DCL 1 key,
      2 mbz bit(6),      /* not used, must be 0 */
      2 daxr bit(1),     /* XK$DAX: select on called addr extn */
      2 flush bit(1),    /* XK$FLU: remove all port assignments */
      2 dadr bit(1),     /* XK$DAD: select on address prefix */
      2 dasr bit(1),     /* XK$DSA: select on address suffix */
      2 dprid bit(1),    /* XK$DPR: select on prid */
      2 dudata bit(1),   /* XK$DUD: select on user data */
      2 dport bit(1),    /* XK$DPO: select on port number */
      2 mbz2 bit(1),     /* not used, must be 0 */
      2 dme bit(1),      /* XK$DME: select only calls to local node */
      2 mbz3 bit(1);     /* not used, must be 0 */
```

Specifies which types of incoming calls are no longer to be returned. Formed from the parts listed below. Some parts are required and others are optional, as indicated in the list. With the exception of XK$DME and XK$DPO, the keys may be supplied by privileged users only.

One of the following parts is required:

XK$DAD	Deassigns incoming calls whose called addresses start with the digits specified in *tadr* and are not among the X.25 addresses configured for this node.
XK$DME	Deassigns incoming calls whose called addresses are among the X.25 addresses configured for this node.

The following parts are optional:

XK$FLU	Removes all port assignments.
XK$DSA	Deassigns incoming calls whose Called Addresses end in *tsadr* and whose remaining digits comply with the XK$DME/XK$DAD specification.

You can also add any of the following parts. Use them individually or use XK$DPR and XK$DUD together.

XK$DAX	Deassigns incoming calls when the Called Address Extension facility is present and when the value of the Called Address Extension begins with *txadr*
XK$DPO	Deassigns incoming calls whose port numbers are *port* (and that do not have a Called Address Extension facility present)
XK$DPR	Deassigns incoming calls whose Protocol ID fields begin with *prid* (and that do not have a Called Address Extension facility present)
XK$DUD	Deassigns incoming calls whose User Data fields begin with *udata* (and that do not have a Called Address Extension facility present)
XK$SAD	Deassigns incoming calls whose Calling Addresses begin with *fadr*
XK$SSA	Deassigns incoming calls whose Calling Addresses end with *fsadr*

tadr

INPUT. Holds a char varying string of ASCII digits, the Called Address prefix. Used only if *key* includes XK$DAD.

tsadr

INPUT. Holds a char varying string of ASCII digits, the Called Address suffix. Used only if *key* includes XK$DSA.

prid

INPUT. Holds a char varying string of bytes, the Protocol ID field. If both *prid* and *udata* are supplied, then the length of *prid* must be 4. Used only if *key* includes XK$DPR.

udata

INPUT. Holds a char varying string of bytes, the User Data field. The maximum length is 12 if the *prid* argument is supplied, and 16 otherwise. Used only if *key* includes XK$DUD.

Note

If both XK$DUD and XK$DPR are used, then *prid* and *udata* are concatenated at runtime. If only XK$DUD is supplied, then the User Data field is assumed to include the Protocol ID field.

port

INPUT. The number of the port to be deassigned. 0 through 99 for unprivileged processes; 0 through 255 for privileged processes. –1 causes all ports to be deassigned. Used only if *key* includes XK$DPO.

txadr

INPUT. Holds a char varying string of bytes, the Called Address Extension prefix to use in the search. Used only if *key* includes XK$DAX.

Supply the Called Address Extension prefix in binary-coded decimal, using a maximum of 41 BCD digits. The first digit has the following meaning:

0 Full OSI NSAP address
1 Partial OSI NSAP Address
2 Non-OSI NSAP Address
3 Reserved by ISO

The remaining digits may take any value from 0 through 9.

rsrvd1

Reserved for future use. Set to 0.

rsrvd2

Reserved for future use. Set to 0.

rsrvd3

Reserved for future use. Set to 0.

status

OUTPUT. The returned status of the call.

The following status codes may be returned by a call to XLUASN:

XS$BPM	At least one of the parameters specified in the call is not in the legal range.
XS$CMP	The port has been unassigned.
XS$NET	Networks are not configured for this system.

Subroutine: X$CLRA

Description: Reinitializing an Application's Network Environment

X$CLRA reinitializes the network environment of a process, including calls held by nested EPFs. Any pending network operations are aborted and all of the virtual circuits held by the application are cleared.

Note

Unlike XCLR, XCLRA does not wait for confirmation from the application on the other side of the circuit before marking the circuit cleared. Therefore, the virtual circuit status word of a circuit cleared by a call to X$CLRA is never updated to show that the circuit is cleared.

In addition to clearing all open virtual circuits, X$CLRA deassigns all ports. In this regard, it is equivalent to the call: X$UASN (–1).

This subroutine also drains the application's network (X$WAIT) semaphore, reducing the chances for spurious network event signals. Refer to *Subroutines Reference III* for complete information on semaphores.

Call syntax

```
CALL X$CLRA
```

This operation always completes successfully. If no virtual circuits are open or no ports are assigned, no action is taken. X$CLRA should be used before an exit from any process or subsystem within an application, or before an exit to PRIMOS. The first Note in the section on XCLR/XFCLR/XLCLR explains why.

Subroutine: X$WAIT

Description: Waiting for Completed PRIMENET Action

X$WAIT performs a semaphore wait for a network activity to complete. Optionally, this wait can be combined with a finite timeout period. Most of the IPCF subroutines initiate an activity and then immediately return so that the application can continue processing while the requested network action completes. The X$WAIT call provides a mechanism by which applications can ask to have processing suspended until any of their network actions completes.

When treated as an INTEGER*2 function, X$WAIT returns a *code* argument that indicates whether the cause of the resumption of execution was a completed PRIMENET action or a timeout.

A network activity is considered complete whenever the status of the corresponding request indicates that PRIMENET will take no further action on that request. In general, this includes any return status except the operation-in-progress code (XS$IP). (A code of XS$IP is always updated by PRIMENET as soon as the relevant activity completes.) A suspended process is also awakened when PRIMENET receives a connection request for that process.

Note

X$WAIT is implemented as a PRIMOS-quittable semaphore. As such, a suspended process may be awakened even though the action has not completed. Code using X$WAIT should make provision for this fact. Refer to the *Subroutines Reference III* for information about semaphores.

To prevent event count rollover, the network semaphore collects network events into a combined single notification when the application is not waiting on its network semaphore. A process should take into account that when it is awakened by X$WAIT, multiple network activities may have completed. In this case, a new X$WAIT call anticipating completion of one of those activities does not wake the application again until another request completes. This situation can cause the program to hang. Accordingly, applications should test *all* outstanding requests.

Call syntax

```
CALL X$WAIT(time)
code = X$WAIT(time)
```

Arguments

time

INTEGER*2. The number of tenths of seconds to remain suspended if no network action completes. (If time is 0, wait indefinitely.)

code

INTEGER*2. Returned function value. May be 0 or 1, as shown below.

0	Some network action (not necessarily the awaited one) completed before the timer expired.
1	The timer may have expired before any network action completed. The expiration of a timer may indicate a network semaphore notify. User programs should check for network activity.

Subroutine: X$GVVC XLGVVC

Description: Transferring a Call

X$GVVC/XLGVVC transfers control of a virtual circuit to another process on the same PRIMENET node. The process issuing the virtual circuit relinquishes the right to clear and to send or receive data through the specified circuit. The circuit is placed in the connection request pending queue for the target process, and is treated thereafter in the same manner as an incoming connection request.

The subroutine can transfer calls to a specific application through the application's user number or to a numbered port (and thus to an application that has assigned the port). If the original Call Request/Call Accept packet has been released, the application can generate a simulated Call Request packet to transfer control of the virtual circuit. You can use this feature to transfer information to the target application through the User Data field. Call Request packets are released when the virtual circuit is accepted; Call Accept packets are released by the first transmit/receive request on the circuit.

An application can transfer control of a circuit in the incoming request queue without accepting the circuit. The initial choice of accepting or clearing the circuit is then left to another application. In this case, the process to which the circuit is transferred sees the circuit connection request as a new request, not as one being transferred.

Control of a circuit cannot be transferred under the following conditions:

1. The application issued the call to X$CONN, X$FCON, or XLCONN to create this circuit and the circuit establishment has not yet been completed (that is, the virtual circuit status has not yet been set to XS$CMP).
2. The application wishing to transfer control of the circuit has a call in progress to X$RCV or X$TRAN.
3. The virtual circuit is already in the process of being transferred.
4. The virtual circuit is a remote login circuit.
5. The contents of the existing Call Request/Call Accept packet and the contents of the user-requested Call Request packet conflict.

Note

While being transferred, a virtual circuit has no owner, and thus has no virtual circuit status vector.

The target application requests and receives information about transferred virtual circuits by calling X$GCON/X$FGCN/XLGCON/XLGC$. Any of these routines returns the VCID, by which the target application identifies the connection being transferred. Just as with new incoming connection requests, transferred connection requests must be cleared or accepted before data transfer can occur. As with new incoming connection requests, a transferred connection that has not been accepted or explicitly cleared within 100 seconds after it enters the incoming request queue is automatically cleared by PRIMENET.

X$GVVC, the short form routine, transfers a call by target process user number.

XLGVVC, the long form routine, permits transfer by either user number or numbered port, and optionally supplies call request data for the transfer.

Call syntax

```
CALL X$GVVC(vcid, userno, status)

CALL XLGVVC(key, vcid, rsvrd1, rsvrd2, userno, port, fadr, fadrn, tadr,
            tadrn,fcty, fctyn, prid, pridn, udata, udatan, status)
```

Arguments

key

INTEGER*2. Specifies selection of target process.

XK$USR	Transfer to user number *userno*
XK$PRT	Transfer by PRIMENET port *port*

vcid

INTEGER*2. The VCID for the circuit being transferred.

userno

INTEGER*2. The user number of the process to which this circuit is being transferred. Used only when *key* = XK$USR, or with X$GVVC. The legal range is from 2 through the sum of the CONFIG directives NTUSR + NPUSR + NRUSR + NSLUSR.

rsvrd1

ANY. Reserved for future use. Set to 0.

rsvrd2

ANY. Reserved for future use. Set to 0.

port

INTEGER*2. The port number to be used to transfer the virtual circuit. Used only when *key* = XK$PRT. Range is 1 through 255.

fadr

Array. Holds a string of bytes (char nonvarying in PL/I). Contains the address of the simulated call-originating network node.

fadrn

INTEGER*2. The number of characters in *fadr*.

tadr

Array. Holds a string of bytes (char nonvarying in PL/I). Contains the address of the target node. For PRIMENET nodes, the maximum length is 6. If you are using an address, the maximum is 15.

tadrn

INTEGER*2. The number of characters in *tadr*.

fcty

Array. Holds a string of bytes. Contains the bytes to go into the simulated call request packet facilities field. (Ignored if *fctyn* is 0.)

fctyn

INTEGER*2. The number of bytes to be taken from *fcty*. Legal range is 0 to 109 (0 to 63 for pre-X.25 1984 connections).

Note

In contrast to initial call requesting and acceptance, there is *no* check done for legality of the supplied facility field. Also, the facilities requested by this facility field are completely ignored. The virtual circuit's parameters remain unchanged. It is suggested that the facility field from the virtual circuit's creation be copied, if the application wants to transmit a facility field.

prid

Array. A buffer that contains the four bytes to go into the simulated call request packet Protocol ID field. (Ignored if *pridn* is 0.)

pridn

INTEGER*2. The number of bytes to be taken from *prid*. Legal values are 0 and 4.

Note

If *pridn* is 0, PRIMENET uses the Protocol ID field to transfer the port specified in *port*. In this case, this field is used for a host-to-host protocol format defined for PRIMENET.

If *pridn* is 4, the application-supplied bytes are used. In this case, the value specified in *port* will still control the virtual circuit transfer.

udata

Array. A buffer that holds the bytes to go into the User Data field of the Call Request packet. (Ignored if *udatan* is 0.)

udatan

INTEGER*2. The number of bytes to be taken from *udata*. Legal values are 0 to 124. Should be 0 for all calls that are not Fast Select calls.

status

INTEGER*2. Returned. Contains the return status of the call.

The following status codes may be returned by a call to X$GVVC/XLGVVC.

XS$BVC	The calling process does not control the virtual circuit specified by *vcid.*
XS$BPM	One of the arguments to the call is missing, out of range, or in conflict with other arguments.
XS$CMP	The operation was successful. This virtual circuit is now pending on the target process's connection request queue.
XS$ILL	This virtual circuit is in one of the states described above during which transfer is prohibited.
XS$MEM	There is temporary local PRIMENET buffer congestion. The system currently does not have the resources required to process the request. Retry the request later.
XS$UNK	The target application is not logged in or the call cannot be transferred by use of the specified port.

Subroutine: X$STAT

Description: Finding Network Status

An application may call X$STAT at any time to determine the state of the network. The value given in *key* specifies the type of status information to be returned. X$STAT returns information about the local system's PRIMENET configuration, the currently open virtual circuits, and the mapping of ASCII PRIMENET names to their X.25 addressing form equivalents. The parameters *num*, *array1*, *alen1*, *array2*, and *alen2* are input arguments, returned values, or unused, depending on the value of *key*.

Call syntax

```
CALL X$STAT(key, num, array1, alen1, array2, alen2, code, time)
```

Arguments

key

INTEGER*2. Specifies information to be returned.

XI$ADR	Returns all X.25 addresses in the network.
XI$AVC	Returns VCIDs of all circuits that are open to or from a specific X.25 address.
XI$VCD	Returns information about a specific virtual circuit.
XI$XTP	Returns the PRIMENET name associated with an X.25 address.
XI$PTX	Returns the X.25 address associated with a PRIMENET name.
XI$MYN	Returns the X.25 address and PRIMENET name of the application's system.
XI$PDN	Returns names of all accessible Packet Switched Data Networks.
XI$PVC	Returns VCIDs of all circuits that are open to or from a specific Packet Switched Data Network.
XI$RLG	Returns the VCID and remote address of a process's remote login circuit.

num

INTEGER*2. Conditionally returned. The number of network addresses returned, the number of virtual circuit IDs returned, the number of accessible Packet Switched Data Network names returned, or no meaning, depending on *key*.

array1

Array. Conditionally returned. Defined by INTEGER*2 words, you should ensure that this is dimensioned according to the size of the network configuration currently active on the node. A buffer containing X.25 addresses, packet switched data network names, and/or PRIMENET names (in ASCII, with two characters per *array* entry), depending on *key*.

alen1

INTEGER*2. Conditionally returned. Indicates how much of *array1* was actually used.

array2

Array. Conditionally returned. If defined by INTEGER*2 words, should be dimensioned to at least 256 words. A buffer containing virtual circuit identifiers, virtual circuit status information, the number of characters in each X.25 address, the number of characters in the PRIMENET system name, or the number of characters in the PSDN name, depending on *key*.

WARNING

It may be necessary to increase the lengths of the two return arrays *array1* and *array2* to cope with large network configurations. As a result, old programs risk being overwritten when PRIMENET needs to use larger arrays than previously dimensioned. Users should review their applications that use X$STAT.

---

alen2

INTEGER*2. Conditionally returned. Indicates how much of *array2* was actually used.

code

INTEGER*2. Returned. Indicates outcome of call.

XS$CMP	The operation was performed successfully.
XS$BPM	Invalid arguments in the call.
XS$NET	No network is configured.
XS$UNK	The X.25 address, virtual circuit, PRIMENET name, or PSDN name is unknown.

time

Returned. INTEGER*2. The current time; retrieved in quad seconds since midnight.

Each type of status call is described below. The meanings of *code* and *time* are the same for all values of *key*. Starred arguments (*) are input arguments, and the other arguments are returned by the call.

XI$ADR — *num* contains the number of addresses in the network. *array1* contains the addresses, two characters per entry, one name right after the other. *alen1* contains the used length of *array1*. Each entry in *array2* specifies the number of ASCII digits in the network addresses given in *array1*. *alen2* contains the number of used words in *array2*, which equals the value of *num*.

To find the offset into *array1* for a specific address, add the lengths of the previous addresses, converted into needed array words per address. Each address uses an integer number of array words, even if it has an odd number of digits.

All PRIMENET nodes have addresses, even if they are not connected to a PSDN. In the latter case, PRIMENET provides a fictitious number calculated from the node name, starting with 9999. To find the node name corresponding to an address, call X$STAT using key XI$XTP.

XI$AVC

num contains the number of virtual circuits open to or from a specific network address. *array1** specifies the address of interest. *alen1** is the number of ASCII digits in the address of interest. The entries in *array2* contain the VCIDs. *alen2* is set to the actual used length of *array2*, which equals the value of *num*.

XI$MYN

num has no meaning and is not modified. *array1* is set to the PRIMENET system name of the local node. *alen1* contains the number of characters in the system's name. *array2* contains the X.25 address for the local node. *alen2* contains the number of ASCII digits in the X.25 address.

XI$PDN

num contains the number of accessible Packet Switched Data Networks. *array1* contains the names of these networks. *alen1* contains the used length of *array1*. Each entry in *array2* contains the number of ASCII characters per corresponding network name. *alen2* contains the number of used words in *array2*, which equals the value of *num*.

XI$PTX

num has no meaning and is not modified. *array1** specifies the PRIMENET system name of interest. *alen1** specifies the number of characters in *array1*. *array2* contains the X.25 address. *alen2* contains the number of ASCII digits in *array2*.

XI$PVC

num contains the number of virtual circuits open to the specified Packet Switched Data Network. *array1** specifies the Packet Switched Data Network of interest. *alen1** specifies the number of characters in the name of the PSDN. The entries in *array2* are the circuit numbers. *alen2* is set to the actual used length of *array2*, which equals the value of *num*.

XI$RLG

num is set to the VCID of the process's remote login circuit. *array1* contains the remote address, two characters per word. *alen1* is set to the length of *array1*, in characters. *array2* and *alen2* are not used.

XI$VCD

*num** specifies the VCID of interest. Each entry in *array2* is described below. *alen2* is 13 words. *array1* and *alen1* are not used.

array2(1)	Circuit status. See notes below.
array2(2)	User process number.
array2(3)	Maximum packet size in bytes.

array2(4)	Packet-level window, that is, the maximum number of outstanding packets. Note that the packet size and window size returned are the *input* direction sizes. Usually the *output* direction sizes are the same, but X.25 permits facility negotiations that can make them unequal. (If desired, the output sizes could be found by retrieving the input sizes at the other end of the virtual circuit. If a PSDN is part of the virtual circuit, the packet and window sizes may be changed when they pass through the PSDN.)
array2(5)	Port number of call.
array2(6)	Number of resets since call began.
array2(7)	Minutes open.
array2(8)	First word of the number of packets received.
array2(9)	Second word of the number of packets received. (*array2(8)* concatenated with *array2(9)* thus forms an INTEGER*4 variable.)
array2(10)	First word of the number of packets sent.
array2(11)	Second word of the number of packets sent. (*array2(10)* concatenated with *array2(11)* thus forms an INTEGER*4 variable.)
array2(12)	Controller type. See notes below.
array2(13)	RINGNET node ID, logical SMLC line number, or LAN300 node index. Loopback returns a value of –1.

Values for circuit state and controller type are listed below.

Circuit State	*Meaning*
1	Remote login (on this system).
2	Unused.
3	Unused.
4	Circuit being transferred.
5	User data transfer.
6	User local call request pending.

7	User remote call request pending.
8	User local clear request pending.
9	Unused.
10	Unused.
11	Unused.
12	Clear desired but no memory available. PRIMENET will automatically retry clear request.
13	Unused.
14	Remote log-through (placing a login to another node).
15	Have received Clear Indication; waiting for Clear Confirm.
16	Call Request awaiting Restart.

Controller Type	*Meaning*
1	Reserved
2	SMLC
3	Ring (PNC)
4	Local connection within same machine
5	LAN300 Host Controller (LHC)

XI$XTP

num has no meaning and is not modified. *array1** specifies the X.25 address of interest. *alen1** specifies the number of ASCII digits in *array1*. *array2* contains the PRIMENET system name. *alen2* contains the number of ASCII characters in *array2*.

Subroutine: X$RSET

Description: Resetting a Virtual Circuit

X$RSET resets the virtual circuit and aborts all operations in progress. The call to X$RSET causes PRIMENET to transmit a reset request packet to the other end of the virtual circuit, complete all X$TRAN and X$RCV operations (with status = XS$RST), and reinitialize flow control variables in accordance with X.25 packet level protocol. If PRIMENET cannot send a reset immediately, it sends the reset when resources are available.

Caution

Very few applications need to use resets. Resetting a virtual circuit is an extreme procedure. Use it only if you have a thorough understanding of the X.25 protocol. A reset can result in the loss of data.

Call syntax

CALL X$RSET(vcid,why,status)

Arguments

vcid

INTEGER*2. The virtual circuit ID for this circuit.

why

INTEGER*2. The low-order byte of *why* may take on values from 0 through 255 and is taken as the diagnostic code. (Refer to Chapter 2, Port Assignments and Virtual Circuits, for information about diagnostic codes.) The high-order byte of *why* is usually 0, or may exceptionally take on a value from 128 through 255, in which case it will be used as a cause code.

status

INTEGER*2. Returned. Contains the returned status of the call. The following status codes may be returned in the status word:

XS$CMP	The operation is successful. All pending transmits and receives are aborted with a status of XS$RST.
XS$BVC	The calling process does not control the virtual circuit specified by the *vcid.*
XS$NET	Networks not configured.
XS$ILL	The operation is illegal. This error is the result of attempting transmission over "an almost-open" or "almost-closed" circuit.
XS$CLR	The connection has been cleared and is no longer usable.

5

IPCF Programming Examples

The purpose of the following sample programs is to illustrate typical code for basic IPCF data transfer. The first example contains only the basic code path needed for error-free function. The second example presents full error handling, following the guidelines set forth in Chapter 3, IPCF Programming Principles. Also, the general design rules discussed in Chapter 3 are used in the second example. Each example is preceded by a brief description.

File-transmission System

This example consists of two programs, NETSND and NETRCV. Together, they form a rudimentary PRIMENET file-transmission system. They use a common subroutine WAITIL to check for completed network requests. The code of WAITIL follows NETRCV. Notice that the receiver program NETRCV uses double receive buffers to offload PRIMENET.

For simplicity, filenames of only eight characters or less are allowed; access to other directories is not provided for. This example's error handling consists only of STOP statements, which would not be sufficient for a real-life application.

The Transmitting Side

The following program represents the send side of a network copy program.

```
C NETSND.FTN - A SIMPLE, FAST NETWORK FILE COPY PROGRAM
C
C This is the transmitting side of the program; see also NETRCV.
C
$INSERT SYSCOM>X$KEYS.INS.FTN
$INSERT SYSCOM>ERRD.INS.FTN
$INSERT SYSCOM>KEYS.INS.FTN
C
      INTEGER*2 J,FUNIT,CODE,NWR,LEVEL,RSTATE(3),XSTATE,VCID,
     *  VCSTAT(2),FILNAM(4),SYSTEM(3),BUF(1024)
C
C
C The basic idea is to:
C     *  Open the requested file in the current ufd.
C     *  Establish a circuit with a server on another system.
C     *  Send over the filename to the server so it can open
C        a file of the same name for writing.
C     *  Send over 1K blocks of the file until we read to the
C        end of the file.
C     *  Signal EOF to the server by sending a different
C        user data level.
C     *  The server will acknowledge our end of file signal by
C        sending us the code from its close of the target file.
C
C  STOPS are used to signal error conditions:
C        :20 error in circuit establishment
C        :24 error in transmit of filename
C        :25 bad state from transmit of data
C        :30 bad state in acknowledgement receipt
C        :32 bad level in acknowledgement receipt
C        :34 bad length in acknowledgement receipt
C        :50 bad status on clear of virtual circuit
C
C
      FUNIT=1
      WRITE(1,11)
11    FORMAT(' INPUT FILE NAME, 8 CHARS OR LESS')
      READ(1,12) FILNAM              /* Ask for a file to open
12    FORMAT(4A2)
      WRITE(1,13)
13    FORMAT(' INPUT REMOTE SYSTEM NAME')
      READ(1,14) SYSTEM
14    FORMAT(3A2)
```

```
C
C  Clear everything
C
      CALL X$CLRA
C
C  Open the file

C
      CALL SRCH$$(K$READ,FILNAM,8,FUNIT,J,CODE)
C
C  and make sure the file was found.
C
      CALL ERRPR$(K$SRTN,CODE,'ON OPEN',7,0,0)
C
C  Now we set up the virtual circuit.
C  We assume that the receiving half of this program
C  is running on the
C  remote machine and that it has assigned PRIMENET port 45.
C  Connect to remote node. Await non-XS$IP status.
C
      CALL X$CONN(VCID,45,SYSTEM,6,VCSTAT)
      CALL WAITIL(VCSTAT)
      IF (VCSTAT(1).NE.XS$CMP) STOP :20
C
C
C  Send over the filename. Level 1 for control info, level 0 for
C  file data.
C
      CALL X$TRAN(VCID,XT$LV1,FILNAM,8,XSTATE)
C
      CALL WAITIL(XSTATE)
      IF (XSTATE.NE.XS$CMP) STOP :24 /* Could not xmit filename.
C
C
C  Now just keep sending till EOF
C
      LEVEL=XT$LV0                    /* Use data level 0 'til EOF.
C
30    CALL PRWF$$(K$READ,FUNIT,LOC(BUF),1024,000000,NWR,CODE)
C
      IF (CODE.EQ.0) GOTO 35          /* Read OK.
      IF (CODE.NE.E$EOF)              /* If not EOF, it's an error.
     *   CALL ERRPR$(K$SRTN,CODE,'READ',4,0,0)
C
      LEVEL=XT$LV1                    /* Switch LEVEL for EOF.
```

```
35    CALL X$TRAN(VCID,LEVEL,BUF,NWR*2,XSTATE)
      CALL WAITIL(XSTATE)
      J=XSTATE
      IF (J.NE.XS$XMP) CALL X$CLR(VCID,0,J)
      IF (J.NE.XS$CMP) STOP :25
      IF (LEVEL.EQ.0) GOTO 30        /* Keep sending 'til EOF.
C
C
C  Now wait for acknowledgement from
C  the remote node in the form of a
C  1 word (2 bytes) standard file system error code.
C
      CALL X$RCV(VCID,CODE,2,RSTATE)
      CALL WAITIL(RSTATE)
      IF (RSTATE(1).NE.XS$CMP) STOP :30 /* Bad status
      IF (RSTATE(2).NE.XT$LV1) STOP :32 /* Bad data level for
C                                            control msg
      IF (RSTATE(3).NE.2) STOP :34      /* Bad length
C
C  Always RETURN to clear the virtual circuit.
C
      CALL ERRPR$(K$IRTN,CODE,'REMOTE CLOSE',13,0,0)
      CALL SRCH$$(K$CLOS,0,0,FUNIT,J,CODE)
      CALL X$CLR(VCID,0,J)              /* Clear virtual circuit.
      IF (J.NE.XS$CMP) STOP :50         /* Check status.
      CALL EXIT
      END
```

The Receiving Side

The following program represents the receiving end of a network copy program.

```
C  NETRCV.FTN - A SIMPLE, FAST NETWORK FILE COPY UTILITY
C
C This is the passive receiving part of a pair of programs.
C The active part, NETSND, runs on another system in this
C network. See NETSND for more information.
C
C
C  The basic idea is to:
C     *  Clear all ports.
C     *  Assign port 45.
C     *  Wait for connection requests.
C     *  Accept the call.
C     *  Get an 8-byte message with the filename to be opened.
C     *  Open the file for writing.
C     *  Receive data messages and write them to the file,
C        while received
C        data messages are not 'USER LEVEL 1'.
C     *  Write a 'LEVEL 1' message to the file.
C     *  Close the file.
C     *  Send an acknowledgement to the sender.
C     *  Wait for the sender to clear the circuit.
C     *  Wait for the next connection request.
C
C STOPS are used to signal error conditions.
C   :10 error on assign of port
C   :14 error in X$GCON call
C   :20 error in X$ACPT call
C   :30 bad receive of filename
C   :32 bad level on receive of filename
C   :34 bad length on receive of filename
C   :40 error while receiving data for the file
C   :50 error transmitting status to sender
C
C NETRCV is designed to be run as a phantom.  The phantom command
C file would have the following format:
C          COMO output file
C          A ufd                  This is the directory files will
C                                 be copied to.
C          EXECUTE NETRCV_runfile
C          LO                     Make sure we log out on error.
C          CO TTY
C
```

```
$INSERT SYSCOM>X$KEYS.INS.FTN
$INSERT SYSCOM>KEYS.INS.FTN
$INSERT SYSCOM>ERRD.INS.FTN
C
      INTEGER BUF(1024,2),CODE,FUNIT,J,Q,RSTATE(3,2),VCSTAT(2),
     +    STAT(2),VCID,I,L
C
C
C  It is a very good programming practice to always have a RECEIVE
C  pending, to relieve the operating system of buffering problems.
C  The buffer and receive status vector will be able to handle two
C  messages at once.
C
      CALL X$CLRA                    /* Clear-up, in case port 45
C                                       previously in use.
      CALL X$ASGN(45,0,J)            /* Assign port 45 forever.
      IF (J.NE.XS$CMP) STOP :10      /* Bad assign.
10    CALL X$WAIT(0)                 /* Wait for connection request.
      CALL X$GCON(VCID,J,STAT)
      L=STAT(1)                      /* Temporary copy.
      IF (L.EQ.XS$NOP) GOTO 10       /* Not really a connect yet.
      IF (L.NE.XS$CMP) STOP :14      /* Bad call to X$GCON.
C
C
C  Here, if we have gotten a connect request.
C
      CALL X$ACPT(VCID,VCSTAT)       /* Accept the connection.
      L=VCSTAT(1)
      IF (L.NE.XS$IDL .AND. L.NE.XS$CMP)
     *  STOP :20                     /* Some error on ACCEPT.
C
C
C  Now get the file name and open the file.
C
      CALL X$RCV(VCID,BUF(1,1),8,RSTATE(1,1))
      CALL WAITIL(RSTATE(1,1))             /* Wait for not XS$IP.
      IF (RSTATE(1,1).NE.XS$CMP) STOP :30 /* Bad receive.
      IF (RSTATE(2,1).NE.1) STOP :32       /* Wrong level.
      IF (RSTATE(3,1).NE.8) STOP :34       /* Bad length.
C
      FUNIT=3
      CALL SRCH$$(K$WRIT,BUF(1,1),8,FUNIT,J,CODE) /* Open new file.
      CALL ERRPR$(K$SRTN,CODE,'OPEN',4,0,0)
C
C  Virtual circuit and file are both ready.
C  Let's get into the main copy loop.
C  This loop uses double buffering of receives.
```

```
C
C  Start first receive.
C
      CALL X$RCV(VCID,BUF(1,1),2048,RSTATE(1,1))
C
C  Start second receive.
C
      CALL X$RCV(VCID,BUF(1,2),2048,RSTATE(1,2))
      I=2                              /* Init double buffer pointer.
C
20    I=I+1                            /* Indexes the buffer number.
      IF (I.EQ.3) I=1                  /* Flip flops 1,2,1,2,1,2....
      CALL WAITIL(RSTATE(1,I))
      J=RSTATE(1,I)
      IF (J.NE.XS$CMP) STOP :40        /* Error on RCV.
      L=RSTATE(3,I)/2                  /* Convert bytes to words.
      CALL PRWF$$(K$WRIT,FUNIT,LOC(BUF(1,I)),L,000000,J,CODE)
      CALL ERRPR$(K$SRTN,CODE,'WRITE',5,0,0)
      IF (RSTATE(2,I).EQ.XT$LV1)  GOTO 30       /* Level 1 <=> EOF.
      CALL X$RCV(VCID,BUF(1,I),2048,RSTATE(1,I)) /* Issue receive.
      GOTO 20
C
30    CALL SRCH$$(K$CLOS,0,0,FUNIT,J,CODE)  /* Close the file.
      CALL X$TRAN(VCID,XT$LV1,CODE,2,J) /* Return close code
C                                                  to sender.
      CALL WAITIL(J)               /* Wait for transmit complete.
      IF (J.NE.XS$CMP) STOP :50    /* Error on transmit.
50    CALL X$WAIT(0)               /* Wait for CLEAR from sender.
      J=VCSTAT(1)                  /* Copy over circuit status.
      IF (J.EQ.XS$CLR) GOTO 10     /* OK to accept next call.
      IF (J.EQ.XS$CMP) GOTO 50     /* Wait for CLEAR,
      STOP :54                     /* else some circuit error.
      END
```

The following subroutine, which simply waits for the next network event, is called by NETSND and NETRCV.

```
C WAITIL.FTN
C
C This subroutine returns when its argument
C (an asynchronously updated
C network status word) is anything other than 'XS$IP'.
C
      SUBROUTINE WAITIL(STWORD)
C
$INSERT SYSCOM>X$KEYS.INS.FTN
C
      INTEGER STWORD
      IF (STWORD.NE.XS$IP) RETURN    /* Nothing to wait for?  Don't,
C                                       unless process gets hung on
C                                       the semaphore.
10    CALL X$WAIT(0)
      IF (STWORD.EQ.XS$IP) GOTO 10
      RETURN
      END
```

Database Example Using Fast Select Calls

This example illustrates the use of Fast Select calls and the corresponding short form IPCF subroutines. In addition, a call transfer to a second server type illustrates the use of XLGVVC. The protocol at the user level is designed to minimize network overhead and connection time. The intention of this scenario is to offer a set of sample programs that provide a query and update service on a database, with the users spread over a network but the database centralized to one node. The majority of the transactions are supposed to be status questions, which have brief answers. However, some updates and queries have long answers. The implementation contains

- A user program
- A query server (single-threaded, running in multiple invocations)
- One update server (multi-task design)

The user program handles screen layouts and data compression/expansion. The user program connects to a query server to obtain answers or perform updates. The transaction input data always fits into a record of 80 bytes. The brief answers occupy a record of 100 bytes. Long answers are of various lengths. Updates also require answers of varying lengths. These long answers are a maximum of 2000 bytes long. They are transferred as single IPCF messages, and the "receive complete" status indicates the end of transfer.

The majority of exchanged data records will be either 80 or 100 bytes, both of which fit into the User Data fields of Fast Select calls. Thus, questions that have brief answers are handled as Fast Select calls, given a fast clear by the query server. Long answers and updates are also initiated as

Fast Select calls, but these are accepted. From the server the user program acknowledges the call by clearing the VC with an appropriate clearing diagnostic.

The system designer has decided to run all updates by a separate single update server, having the multiple query servers to detect update calls and pass these off to the update server.

Capacity and Service Busy

Several query servers run in parallel to handle the stream of queries. The number of servers is expected to be so large that the probability of no available server is acceptably low. Consequently, there is no special server to indicate that no server is available. The application will deduce from the clearing diagnostic CD$PNA (port not assigned) that all query servers are currently busy.

This example uses only one update server. However, this server can have multiple active requests going, each with its own VC. Again, the maximum number of service VCs in the update server is expected to be sufficiently large. If the maximum is exceeded, the update server will clear the new incoming call with an application-specific diagnostic (CD$NVC). The calling user program will receive this diagnostic and handle it appropriately. If the update server is not running, any query server that attempts to pass an update will fail, and will clear the call with the same diagnostic.

Timing Aspects

The protocol attempts to keep turnaround times small. In fact, the majority of transactions, queries with brief answers, are handled with only two information-carrying packets, the Call Request packet and the Clear Request packlet. Notice that long answers are submitted to PRIMENET as single messages, and that the final user side acknowledgement of a long answer is combined with the clear request, generated by the user program.

The query server extracts the answer to a question by a subroutine call, before either clearing or accepting the virtual circuit. This means that this database routine must return within the timeframe set by PRIMENET so that the user process can act on the call request. Otherwise, PRIMENET does a safety clear, and the user gets no answer. Similarly, the update server is busy when calling the update routine. This means that request for call transfer will temporarily hang, and the same maximum timeframe for delays exists here.

If the update times tend to be considerable, you might consider splitting the update server into two processes, one dealing with PRIMENET and the other with the database. The network process would simply queue updates for its mate, and sort responses on appropriate virtual circuits for transmission. The link between these two update mates could either be a virtual circuit, the file system, or shared memory with write access. The program structure of the network handler would remain much the same, with additional code in the "per active virtual circuit" loop, to find completed answers and transmit them.

Virtual Circuit Timeout Handling

Notice also the difference in handling aging virtual circuits. The query server is a single-path program, and the final timeout before safety clearing is simply implemented as a long wait on the network semaphore, combined with a safety call to X$CLRA before restarting.

In contrast, the update server must run frequently to ensure that every virtual circuit moves along. If awakened, it may well be on behalf of another virtual circuit, so it is difficult to implement long timeouts by using the network semaphore. The solution chosen is to run a watchdog timer for each circuit, and to force the clear when this expires. Furthermore, this server must *not* call X$CLRA, as this would obviously kill all active circuits.

The service period for a particular VC slot within the update server allows for the possibility for the VC to be cleared while that VC index gets reused for another incoming call to the update server. In this case two VC slots will refer to the same PRIMENET VC index. Potentially this could cause some confusion. In the example below, the program ignores the previous occurrence of a particular VC index, presuming that PRIMENET correctly cleared the first virtual circuit before opening the second. You may want to account for this possibility differently in other contexts.

The Code

The routines for handling the user terminal and the database are omitted. The names of the tasks are included. The programs are coded in F77. Extensive use of explicit INTEGER*2 declarations has been made, to remind the reader that F77's default integer mode of INTEGER*4 causes a risk for wrong argument values when called routines expect INTEGER*2 arguments, as the IPCF routines do.

Common Insert Files

```
C FSX_DATA.INS.F77, Common declaration of action keys and
C query message structures for Fast Select program example
C
      NOLIST
C
C PRIMENET node and port data. Values to be tailored
C for each installation.
C
      INTEGER*2 query_port, update_port
      PARAMETER (query_port = $$, update_port = &&)
C
      CHARACTER*6 server_node /'XXXXXX'/
C
C Action keys, contained in request message and initial response.
C
```

```
      INTEGER*2 query, update, brief_response, long_response,
     +      update_started, exit
      PARAMETER (query = 1, update = 2, brief_response = 3,
     +      long_response = 4, update_started = 5, exit = 6)
C
C Clearing diagnostics, used for various return status messages.
C
      INTEGER*2 CD$SHR, CD$LNG, CD$EOU, CD$TMO, CD$RST, CD$NVC,
     +      CD$QIT
      PARAMETER (CD$SHR = 1,  /* Fast clear, proper short reply
     +      CD$LNG = 2,       /* Clear, ack of long reply
     +      CD$EOU = 3,       /* Clear, ack of update response
     +      CD$TMO = 4,       /* Clear by impatient server
     +      CD$RST = 5,       /* Clear after reset
     +      CD$NVC = 6,       /* Update server has no VC
C                              /* (or is not running)
     +      CD$QIT = 7)       /* User early abort
C
C User to server request message (80 bytes, first two
C to hold the action_key).
C
      INTEGER*2 msg_size, data_size
      PARAMETER (msg_size = 80, data_size = msg_size-2)
C
      INTEGER*2 message(msg_size/2), action_key,
     +      data_string(data_size/2)
C
      EQUIVALENCE (action_key, message(1)),
     +      (data_string(1), message(2))
C
C Server to user response:
C [brief_response]:
C brief response is 100 bytes, first two to hold response key.
C [long_response]:
C long response up to 2000 bytes, -"-
C     and the first 100 come in the fast accept, the rest by X$RCV
C [update_started]:
C two bytes only, the rest by later X$RCV.
C
      INTEGER*2 resp_size, answer_size, total_size, remndr_size
      PARAMETER (resp_size = 100, answer_size = resp_size-2,
     +      total_size = 2000, remndr_size = total_size - resp_size)
C
      INTEGER*2 response(total_size/2), response_key,
     +      answer(answer_size/2), remainder(remndr_size/2)
```

```
C
      EQUIVALENCE (response_key, response(1)),
     +      (answer(1), response(2)),
     +      (remainder(1), response(resp_size/2+1))

C RETURNED is the array to catch the retrieved User Data field
C (rudat). By equivalencing, the interesting response is extracted
C from behind the four protocol id bytes.
C
      INTEGER*2 return_size
      PARAMETER (return_size = resp_size + 4)
C
      INTEGER*2  returned(return_size/2)
      EQUIVALENCE (response(1), returned(3))
C
      LIST



C QUIT_HANDLING.INS.F77, Defines variables for the quit-handler to
C issue a virtual circuit clear and return to the main program.
C
      NOLIST
C
      COMMON /QUITVARS/ user_vc, restart_stmt, confirmed_stmt
      INTEGER*2 user_vc
      REAL*8 restart_stmt, confirmed_stmt
C
      LIST



C MULTI_VC.INS.F77, multiple VC database for update server
C
      NOLIST
C
C History:
C 1983-09 B. Lindblad Initial coding
C 1984-10 B. Lindblad Removed external declarations for
C                 functions; they are in the calling programs.
C
      INTEGER*2 pool_size
      PARAMETER (pool_size = 10)
C
```

```
      COMMON /ADMNVC/ next_free, total_used
      COMMON /MANYVC/ in_use(pool_size),
     +      vc_id(pool_size),
     +      vc_status(2, pool_size),
     +      xmit_status(pool_size),
     +      zero_time(pool_size)
C
      LOGICAL*2 in_use
      INTEGER*2 next_free, total_used, vc_id, vc_status,
     +      xmit_status, zero_time
C
      EXTERNAL INITVC, FREEVC, ORGSTAMP
C
      LIST



C UPDATE_DATA.INS.F77, Per virtual circuit update message buffers.
C This requires previous insert of (multi_vc fsx_data).ins.f77.
C
      NOLIST
C
      COMMON /UPDIN/ in_msg(msg_size, pool_size)
      INTEGER*2 in_msg
C
      COMMON /UPDOUT/ out_msg(total_size, pool_size)
      INTEGER*2 out_msg
C
      LIST
```

User Program

```
C FS_USER.F77 Sample program for fast-select PRIMENET connections.
C
C This is the interactive user program, to be run when needed.
C Any user action initiates a Fast Select call to one of
C the query_servers, that either responds or passes the call to
C the update_server.
C
C Short replies arrive back with a fast clear. Updates and long
C answers are both accepted and yield further data transfer,
C in the form of one long message. On receiving this, the user
C acknowledges by a normal clear with appropriate diagnostic.
C
C History:
C 1983-09 B. Lindblad Initial coding
C 1986-10 B. Lindblad Added missing code, handling busy updateserver
C 1986-11 B. Lindblad Added on_unit for QUIT$, permitting user abort
```

```
C
      PROGRAM main
C
$INSERT syscom>x$keys.ins.ftn
$INSERT *>fsx_data.ins.f77
$INSERT *>quit_handling.ins.f77
C
      INTEGER*2 vc_status(2), rcv_stat(3), statword, timeout,
     +      actual_ret_size, temp, clr_cause, clr_diag
C
      INTEGER*2 X$WAIT
C
      INTRINSIC INTS, INTL, LT, RT
      EXTERNAL X$WAIT, X$FCON, X$RCV, X$CLR, X$CLRA, SLEEP$,
     +      TNOU, MKON$P, MKLB$F
      EXTERNAL getinput, dispdata, dispansw, noserver,
     +      busyupdateserver, nepr, quit_clear
C
C
C Create onunit for QUIT$; do a virtual circuit clear and return
C to the program's user command level.
C We have to create labels for nonlocal "goto" statements.
C
      CALL MKON$P('QUIT$',INTS(5),quit_clear)
      CALL MKLB$F($1, restart_stmt)
      CALL MKLB$F($240, confirmed_stmt)
C
C
C Keep executing this until the user says 'exit'.
C
1     CONTINUE
C
C Get the user input - this will include screen formatting, etc.
C
      CALL getinput(message)
C
      IF (action_key .EQ. exit) RETURN
C
C Whether the user asks for a query or an update, the program calls
C the query-server.
C
      CALL X$FCON(XK$NAM+XK$ANY, XK$ACC, user_vc,
     +      query_port, server_node, INTS(6), message, msg_size,
     +      vc_status,
     +      returned, return_size, actual_ret_size)
C
```

```
C Status test:
C XS$CMP and XS$CLR imply completed connection;
C XS$IP: wait for Primenet to complete the connection,
C        then get new copy of the VC status;
C All others: crash. Clear everything, return to command level.
C
100   temp = vc_status(1)                  /* Get local copy
C
      IF (temp .EQ. XS$IP) THEN
         timeout = X$WAIT(INTS(100))       /* Arbitrary 10 seconds
         GO TO 100
      ELSE IF (temp .EQ. XS$CLR) THEN
         GO TO 150
      ELSE IF (temp .EQ. XS$CMP) THEN
         GO TO 200
      ELSE
         CALL nepr(vc_status, INTS(2))
         GO TO 9000
      ENDIF
C
C ---   ---   ---
C Some sort of connection success has occured (XS$CLR or XS$CMP).
C If the connect was completed, this means that the server
C accepted the call, to do further data transfer.
C If the connect was cleared, this could either be by a failure,
C or by a Fast Select clear.
C In the latter case, there is data to display.
C
C ---
C Cleared connection:
C IF the cause and diagnostic are correct, display results and
C restart, else just restart.
C
150   clr_cause = LT(vc_status(2), 8)
      clr_diag = RT(vc_status(2), 8)
C
      IF (clr_cause .EQ. CC$CLR .AND. clr_diag .EQ. CD$SHR) THEN
         CALL dispansw(response)
         GO TO 1
      ELSE IF (clr_cause .EQ. CC$CLR .AND. clr_diag .EQ. CD$PNA) THEN
         CALL noserver
         GO TO 1
      ELSE IF (clr_cause .EQ. CC$CLR .AND. clr_diag .EQ. CD$NVC) THEN
         CALL busyupdateserver
         GO TO 1
```

```
      ELSE
         CALL nepr(vc_status, INTS(2))
         GO TO 400
      ENDIF
C
C ---
C Accepted connection:
C Issue receive call for returned data, then display the result.
C The invented protocol controls the choice of array to supply to
C X$RCV.
C EITHER it is a long answer, in which case the beginning is
C already here,
C OR it is just the key "update_started" and the whole answer,
C except the action key, will come later.
C
200   IF (actual_ret_size .LT. INTS(6)) GO TO 400   /* Key missing!!
C
210   IF (response_key .EQ. update_started) THEN
         CALL X$RCV(user_vc, answer, answer_size + remndr_size,
     +      rcv_stat)
      ELSE
         CALL X$RCV(user_vc, remainder, remndr_size, rcv_stat)
      ENDIF
C
C Status test:
C XS$IP means still coming in, wait a bit( on network semaphore);
C XS$CMP is OK - all done, now ack by a clear and then restart;
C XS$RST means Reset occured: clear the circuit;
C XS$CLR is not anticipated - restart;
C XS$MEM means attempt failed, retry the receive shortly;
C all others fatal, crash!
C
220   temp = rcv_stat(1)
      IF (temp .EQ. XS$IP) THEN
         timeout = X$WAIT(20)              /* Arbitrary 2 seconds
         GO TO 220
      ELSE IF (temp .EQ. XS$CMP) THEN
         GO TO 230
      ELSE IF (temp .EQ. XS$CLR) THEN
         CALL nepr(vc_status, INTS(2))
         GO TO 400
      ELSE IF (temp .EQ. XS$MEM) THEN
         CALL SLEEP$(INTL(1000))           /* Wait a sec...
         GO TO 210
```

```
      ELSE IF (temp .EQ. XS$RST) THEN     /* Reset occured, clear
         CALL X$CLR(user_vc, CD$RST, statword)
         IF (statword .NE. XS$CMP) THEN
            CALL nepr(statword, INTS(1))
            GO TO 9000
         ENDIF
         GO TO 240                        /* Await confirmation
      ELSE
         CALL nepr(rcv_stat, INTS(3))
         CALL nepr(vc_status, INTS(2))
         GO TO 9000
      ENDIF
C
C Receive complete; now send acknowledging clear, and display
C data. Tell the display routine the entire response length.
C
230   IF (response_key .EQ. update_started) THEN
         CALL X$CLR(user_vc, CD$EOU, statword)
         CALL dispdata(answer, resp_size + rcv_stat(3))
      ELSE
         CALL X$CLR(user_vc, CD$LNG, statword)
         CALL dispdata(answer, rcv_stat(3))
      ENDIF
C
C Verify correct status for the clear request.
C The only reasonable return code here is XS$CMP.
C
      IF (statword .EQ. XS$CMP) THEN
         GO TO 240                        /* Await confirmation
      ELSE
         CALL nepr(statword, INTS(1))
         GO TO 9000                       /* Fatal error
      ENDIF
C
C Tidy up for restart: Await confirmation of requested clear.
C If not arrived in 30 seconds (will the user stand more?),
C forget the circuit and restart.
C
240   IF (vc_status(1) .NE. XS$CLR) THEN
         PRINT 2000
         timeout = X$WAIT(INTS(300))
      ENDIF
2000  FORMAT ('Disconnecting...')
C
      IF (vc_status(1) .NE. XS$CLR)
     +      PRINT 2010
```

```
2010  FORMAT ('Clear request unconfirmed - restarting')
      CALL X$CLRA
      GO TO 1                  /* Restart
C
C
C - - - - - - - -
C Protocol problems or network transmission problems
C
400   CALL TNOU('Transfer failure', INTS(16))
      CALL X$CLRA
      GO TO 1                  /* Restart
C
C
C = = = = = = = =
C Fatal errors: crash exit to Primos command level
C after global network cleanup.
C
9000  CALL TNOU('Network failure', INTS(15))
      CALL X$CLRA
      RETURN
C
      END



C QUIT_CLEAR.F77, Quit handler for user fast-select example program
C
C This quit-handler clears the user's virtual circuit with a
C specific "aborted" diagnostic byte and then returns to the
C main command interface.
C
C History:
C 1986-11 B. Lindblad Initial coding
C
      SUBROUTINE quit_clear(ptr)
C
      INTEGER*4 ptr
C
$INSERT fsx_data.ins.f77
$INSERT quit_handling.ins.f77
$INSERT syscom>x$keys.ins.ftn
C
      EXTERNAL X$CLR, X$CLRA, PL1$NL, TNOU, nepr
      INTRINSIC INTS
      INTEGER*2 statword
C
C Clear the circuit, telling the server that user lost patience.
```

```
C
      CALL X$CLR(user_vc, CD$QIT, statword)
      CALL TNOU('Aborting transaction...',INTS(23))
C
C Verify correct status for the clear request.
C Reasonable return codes here are XS$CMP,
C and XS$BVC, if the circuit was otherwise cleared.
C In either case, return to suitable place in main program.
C
      IF (statword .EQ. XS$CMP) THEN
         CALL PL1$NL(confirmed_stmt)      /* Fall into main program
      ELSE IF (statword .EQ. XS$BVC) THEN
         CALL X$CLRA                      /* Safety cleanup
         CALL PL1$NL(restart_stmt)
      ELSE
         CALL nepr(statword, INTS(1))     /* Fatal error - print it!
         CALL PL1$NL(restart_stmt)        /* Then fall back
      ENDIF

      END
```

Query Server

```
C QUERY_SERVER.F77 Example server program for Fast Select
C PRIMENET connections.
C
C The query_server program is expected to run as several
C parallel processes. Therefore it assigns its port for
C one call only, and always reassigns on completed service.
C Enough servers are expected to be running to ensure almost
C 100 percent availablility. If, however, the application
C runs out of servers, PRIMENET's clearing with CD$PNA
C (port not assigned) will indicate no servers available.
C
C The first action is to detect updates, and to pass these
C to the update server. If the pass fails, the circuit
C is cleared with a special diagnostic.
C
C Depending on the query type the server will either send
C a short answer by a Fast Select clear, or accept the call
C with part of the answer, and then send the rest separately.
C To ensure that the full answer gets through, the user clears
C in this case, thereby acknowledging. The server has a safety
C timeout as well.
C
C History:
C 1983-09 B. Lindblad Initial coding
C 1986-10 B. Lindblad Corrected error regarding returned VC status
C                     on too early user/network clear
```

```
C
      PROGRAM main
C
$INSERT syscom>x$keys.ins.ftn
$INSERT *>fsx_data.ins.f77
C
      INTEGER*2 statword, status(2), vc_status(2), xmt_status,
     +      answer_key, server_vc, dummy_port, rn_len, msg_bytes,
     +      rem_length, clr_cause, clr_diag, timeout,
     +      not_used, must_be_0, temp, junk
      CHARACTER*6 remote_node
      LOGICAL*2 long_flag
C
      PARAMETER (must_be_0 = 0)
C
      INTRINSIC INTL, INTS, LT, RT
      INTEGER*2 X$WAIT
      EXTERNAL X$ASGN, X$WAIT, X$FGCN, X$FCLR, X$FACP, XLGVVC,
     +      X$TRAN, X$CLR, X$CLRA, SLEEP$
      EXTERNAL dbanswer, nepr
C
C
C Restart point -  assign the query-server port to take ONE call
C
1     CALL X$ASGN(query_port, INTS(1), statword)
C
C Error test: XS$CMP or XS$QUE are satisfactory,
C for all others fatal crash.
C
      IF (statword .NE. XS$CMP .AND. statword .NE. XS$QUE) THEN
         CALL nepr(statword, INTS(1))
         GO TO 9000
      ENDIF
C
C Wait for somebody to call. When woken up from
C the network semaphore, find out about the call.
C (For safety, make routine wake-up once every minute.)
C
10    timeout = X$WAIT(INTS(600))
      CALL X$FGCN(XK$NAM, answer_key, server_vc, dummy_port,
     +      remote_node, INTS(6), rn_len,
     +      message, msg_size, msg_bytes,
     +      status)
C
C Status test: XS$NOP means spurious wake up and therefore to wait more,
C              XS$CMP means call to handle,
C              for all others fatal crash.
```

```
C
      temp = status(1)
      IF (temp .EQ. XS$NOP) THEN
         GO TO 10
      ELSE IF (temp .EQ. XS$CMP) THEN
         GO TO 100
      ELSE
         CALL nepr(status, INTS(2))
         GO TO 9000
      ENDIF
C
C ---
C Handle the call.
C Sort out updates, pass them to the query server.
C Otherwise, send the message to the database,
C retrieve the answer, and analyze its length.
C Long answers force call to be accepted, with later
C transmission of the actual answer,
C short ones are "fast_cleared", carrying the answer.
C
100   IF (message(1) .EQ. update) GO TO 300
C
      CALL dbanswer(message, response, long_flag, rem_length)
      IF (long_flag) GO TO 200
C
C ---
C Short answer <-> fast clear:
C Note: here we use the "returned" aggregate array, not to
C place data overlapping PRID field.
C Also the diagnostic code should be CD$SHR, for the user
C program's validity test.
C
110   CALL X$FCLR(server_vc, CD$SHR, returned, return_size, statword)
C
C Status test:
C XS$CMP is OK - all done, restart.
C     NOTE: There is no vc_status array set up yet, so we cannot
C           check for clear confirmation!
C XS$MEM means attempt failed, retry soon.
C XS$BVC occurs if user side cleared too quickly.
C For all others - fatal crash.
```

```
C
      temp = statword
      IF (temp .EQ. XS$CMP) THEN
         GO TO 8000
      ELSE IF (temp .EQ. XS$MEM) THEN
         CALL SLEEP$(INTL(1000))          /* Wait a sec...
         GO TO 110
      ELSE IF (temp .EQ. XS$BVC) THEN
         GO TO 8000
      ELSE
         CALL nepr(statword, INTS(1))
         GO TO 9000
      ENDIF
C
C ---
C Long answer:
C Accept the call, and then transmit the rest of the long answer.
C
200   CALL X$FACP(server_vc, response, resp_size, vc_status)
C
C Status test:
C XS$IDL/XS$CMP is OK - ready to transmit data.
C XS$MEM means attempt failed, retry shortly.
C XS$BVC means the user cleared 'too early'.
C XS$CLR is not expected, but let the server live
C       to do further work.
C For all others, fatal crash.
C
      temp = vc_status(1)                        /* Get local copy
      IF (temp .EQ. XS$IDL .OR. temp .EQ. XS$CMP) THEN
         GO TO 210
      ELSE IF (temp .EQ. XS$MEM) THEN
         CALL SLEEP$(INTL(1000))                 /* Wait a sec...
         GO TO 200
      ELSE IF (temp .EQ. XS$BVC) THEN
         CALL nepr(vc_status, INTS(2))
         GO TO 8000                              /* To restart
      ELSE IF (temp .EQ. XS$CLR) THEN
         CALL nepr(vc_status, INTS(2))
         GO TO 8000                              /* To restart
      ELSE
         CALL nepr(vc_status, INTS(2))
         GO TO 9000                              /* Fatal, death
      ENDIF
C
210   CALL X$TRAN(server_vc, XT$LV0,
     +      remainder, rem_length - resp_size, xmt_status)
```

```
C
C Status test:
C XS$IP means still transmitting, wait some time.
C XS$CMP is OK - all done, now await confirming clear and restart.
C XS$BVC - user side aborted with clear, log it and restart.
C XS$CLR - check if it was the "ack", then restart the server.
C XS$RST - reset occured, clear the circuit.
C XS$MEM means attempt failed, retry shortly.
C For all others, fatal crash.
C
220   temp = xmt_status
      IF (temp .EQ. XS$IP) THEN
         timeout = X$WAIT(20)                    /* Arbitrary 2 sec
         GO TO 220
      ELSE IF (temp .EQ. XS$CMP) THEN
         GO TO 230
      ELSE IF (temp .EQ. XS$BVC) THEN
         CALL nepr(vc_status, INTS(2))
         GO TO 8000                              /* Restarting...
      ELSE IF (temp .EQ. XS$CLR) THEN
         GO TO 240
      ELSE IF (temp .EQ. XS$MEM) THEN
         CALL SLEEP$(INTL(1000))                 /* Wait a sec...
         GO TO 210
      ELSE IF (temp .EQ. XS$RST) THEN            /* On reset, clear
         CALL X$CLR(server_vc, CD$RST, statword)
         IF (statword .NE. XS$CMP) THEN
            CALL nepr(statword, INTS(1))
            GO TO 9000
         ENDIF
         GO TO 7000                              /* Await confirmation
      ELSE
         CALL nepr(vc_status, INTS(2))
         GO TO 9000                              /* Fatal, death
      ENDIF
C
C Transmit complete. Now wait for the acknowledging clear,
C using CD$LNG diagnostic. If too long a time passes,
C then clear from this end, and restart.
C (Ensure the network semaphore is drained before starting
C waiting period of 2 minutes maximum.)
C
230   timeout = X$WAIT(INTS(1))
      IF (vc_status(1) .EQ. XS$CLR) THEN
         GO TO 240                          /* Verify diagnostic
C
```

```
      ELSE
         timeout = X$WAIT(INTS(1200))      /* Give user 2 minutes
         IF (vc_status(1) .EQ. XS$CLR) THEN
            GO TO 240                      /* Verify diagnostic
         ELSE
            PRINT 2300
            CALL X$CLR(server_vc, CD$TMO, statword)
C
C The only reasonable return code here is XS$CMP.
C
            IF (statword .EQ. XS$CMP) THEN
               GO TO 7000                  /* Await confirmation
            ELSE
               CALL nepr(statword, INTS(1))
               GO TO 9000                  /* Fatal error
            ENDIF
         ENDIF
      ENDIF
2300  FORMAT ('Forced server clear...')
C
C Verify cleared by user with correct diagnostic. If not
C correct, print warning message, then restart in any case.
C
240   clr_cause = LT(vc_status(2), 8)
      clr_diag = RT(vc_status(2), 8)
C
      IF (clr_cause .NE. CC$CLR .OR. clr_diag .NE. CD$LNG)
     +      CALL nepr(vc_status, INTS(2))
      GO TO 8000
C
C ---   ---   ---
C Pass over an update: This is done before call acceptance, and
C by port number. It requires XLGVVC, and since the original call
C request packet is still around, we are not allowed to try
C providing a new one.
C
300   CALL XLGVVC(XK$PRT, server_vc, not_used, must_be_0, must_be_0,
     +      update_port,
     +      junk, INTS(0), junk, INTS(0),       /* Unsupplied
     +      junk, INTS(0), junk, INTS(0),       /* packet
     +      junk, INTS(0),                      /* fields
     +      statword)
C
C Status test:
C XS$CMP is OK - all done, restart from the beginning.
C XS$UNK implies that the update-server is not running:
C        clear with diagnostic CD$NVC.
```

```
C XS$MEM means attempt failed, retry shortly.
C XS$BVC will occur if user side aborted too early by clearing.
C For all others, fatal crash.
C
      temp = statword
      IF (temp .EQ. XS$CMP) THEN
         GO TO 8000
      ELSE IF (temp .EQ. XS$BVC) THEN
         CALL nepr(statword, INTS(1))      /* Log it
         GO TO 8000
      ELSE IF (temp .EQ. XS$MEM) THEN
         CALL SLEEP$(INTL(1000))           /* Wait a sec...
         GO TO 300
C
      ELSE IF (temp .EQ. XS$UNK) THEN
         CALL X$CLR(server_vc, CD$NVC, statword)
C
C The only reasonable return code here is XS$CMP
C     NOTE: There is no vc_status array set up yet, so we cannot
C           check for clear confirmation!
C
         IF (statword .EQ. XS$CMP) THEN
            GO TO 8000                     /* Restart immediately
         ELSE
            CALL nepr(statword, INTS(1))
            GO TO 9000                     /* Fatal error
         ENDIF
C
      ELSE
            CALL nepr(statword, INTS(1))
            GO TO 9000                     /* Fatal error
      ENDIF
C
C
C = = = = =
C Tidy up for restart:
C Await confirmation of requested clear.
C If not arrived in 2 minutes, forget the circuit and restart
C (First wait a very short time, to ensure the network semaphore
C gets drained from previous notifications.)
C
7000  timeout = X$WAIT(INTS(1))
      IF (vc_status(1) .EQ. XS$CLR) THEN
         GO TO 8000                        /* That's it!
C
```

```
      ELSE
         timeout = X$WAIT(INTS(1200))
         IF (vc_status(1) .EQ. XS$CLR) THEN
            GO TO 8000
         ELSE
            IF (timeout .NE. 0) PRINT 7010
            CALL nepr(vc_status, INTS(2))
            PRINT 7020
            GO TO 8000
         ENDIF
      ENDIF
C
7010  FORMAT ('Two minutes time-out...')
7020  FORMAT ('Clear request unconfirmed - restarting')
C
C ---
C General restart code: Ensure we start in fresh environment by
C calling X$CLRA
C
8000  CALL X$CLRA              /* Haengslen & livrem...
      GO TO 1
C
C
C * * * * *
C Fatal error - print message and die...
C
9000  PRINT 9010
9010  FORMAT  ('Fatal error')
      CALL X$CLRA
      RETURN
C
      END
```

Update Server:

```
C UPDATE_SERVER.F77 Example server program for Fast Select
C PRIMENET connections.
C
C There is only one update server function, which thus must be
C able to handle multiple virtual circuits.
C
C The life of each individual virtual circuit is:
C X$FGCN -> X$FACP -> do update -> X$TRAN, wait for XS$CMP ->
C -> wait for clear request -> back to free pool...
C In case of the user failing to clear, there must also be a
C timeout mechanism for the virtual circuit to be released.
C
```

```
C A mechanism for allocating and releasing sets of status arrays
C per VC is used, making use of the variable 'next_free'. Since
C only a finite pool of VC-s can be run, calls, arriving when the
C pool is fully used, are cleared, with the diagnostic CD$NVC.
C
C The main structure of this server is to run a service loop,
C paced by the network semaphore. On wake_up it will look for
C new connections, and then check on all active virtual circuits.
C
C History:
C 1983-09 B. Lindblad Initial coding
C 1986-10 B. Lindblad Corrected error regarding returned VC status
C                     on too early user/network clear
C
      PROGRAM main
C
$INSERT syscom>x$keys.ins.ftn
$INSERT *>fsx_data.ins.f77
$INSERT *>multi_vc.ins.f77
$INSERT *>update_data.ins.f77
C
      LOGICAL*2 GETVC
      INTEGER*2 AGE
C
      INTEGER*2 statword, status(2), server_vc, answer_key,
     +      dummy_port, rn_len, msg_bytes, clr_cause, clr_diag,
     +      junk, index, update_length, temp, i
C
      CHARACTER*6 remote_node
C
      INTRINSIC INTS, INTL, LT, RT
      EXTERNAL X$ASGN, X$WAIT, X$FCLR, X$FACP, X$TRAN, X$CLR,
     +      X$CLRA, X$FGCN, SLEEP$
      EXTERNAL do_updat, nepr, getvc, age
C
C Initialize the virtual circuit pool!
C
      CALL initvc
C
C Assign the update-server port to take ALL calls
C
      CALL X$ASGN(update_port, INTS(0), statword)
C
C Error test: XS$CMP is satisfactory.
C For all others, fatal crash. (XS$QUE is not legal, since
C there shall be only one update server.)
C
```

```
      IF (statword .NE. XS$CMP) THEN
         CALL nepr(statword, INTS(1))
         GO TO 9000
      ENDIF
C
C
C START OF MAIN SERVICE LOOP. = = = = = = = = = = = = = = = = =
C Wait on network event, five_second safety time-out.
C (NOTE: This timeout MUST be short, since it controls
C the speed of servicing active virtual circuits.)
C
10    CALL X$WAIT(INTS(50))
C
C ---   ---   ---
C Look for new incoming calls. If a VC 'slot' is available,
C accept the call, else clear it.
C Carry on until calls are exhausted or when no VC slot available.
C
50    CALL X$FGCN(XK$NAM, answer_key, server_vc, dummy_port,
     +      remote_node, INTS(6), rn_len,       /* Collect call
     +      message, msg_size, msg_bytes,
     +      status)
C
C Status test:
C XS$CMP means call to handle.
C XS$NOP means spurious wake up or calls exhausted, continue to
C     do the service loop for active VC-s.
C For all others, fatal crash.
C
      IF (status(1) .EQ. XS$NOP) THEN
         GO TO 200
      ELSE IF (status(1) .EQ. XS$CMP) THEN
         GO TO 100
      ELSE
         CALL nepr(status, INTS(2))
         GO TO 9000
      ENDIF
C
C ---
C Handle the call. If VC available, accept it, else clear.
C
100   IF (.NOT. getvc(server_vc, index)) THEN
C
         CALL X$CLR(server_vc, CD$NVC, statword)
C
C The only reasonable return code here is XS$CMP
```

```
C     NOTE: There is no vc_status array set up yet, so we cannot
C           check for clear confirmation!
C
         IF (statword .EQ. XS$CMP) THEN
            GO TO 50                        /* Look for further calls
         ELSE
            CALL nepr(statword, INTS(1))
            GO TO 9000                      /* Fatal, death
         ENDIF
C
       ELSE
         out_msg(1,index) = update_started
105      CALL X$FACP(server_vc, out_msg(1,index), INTS(2),
     +      vc_status(1, index))
C
C Status test:
C XS$IDL/XS$CMP is OK - all done, restart from the beginning.
C XS$MEM means attempt failed, retry.
C XS$BVC occurs if other end cleared before we accept.
C XS$CLR is not expected, but let us carry on in that case.
C For all others, fatal crash.
C
         temp = vc_status(1,index)          /* Get local copy
C
         IF (temp .EQ. XS$IDL .OR. temp .EQ. XS$CMP) THEN
            GO TO 110
         ELSE IF (temp .EQ. XS$MEM) THEN
            CALL SLEEP$(INTL(1000))         /* Wait a sec...
            GO TO 105
         ELSE IF (temp .EQ. XS$BVC) THEN
C
C Other end cleared too quickly, so VC does not exist any more.
C Free it at once.
C
            CALL freevc(index)
            GO TO 50
         ELSE IF (temp .EQ. XS$CLR) THEN
C
C Unexpected. Look for more calls. The virtual circuit will be
C released in the "test if cleared" loop further below.
C
            GO TO 50
         ELSE
            CALL nepr(vc_status(1, index), INTS(2))
            GO TO 9000                      /* Fatal death
         ENDIF
C
```

```
C
C Circuit set up. Get the update done, transmit the update message,
C and start the clock for "time since transmit".
C
C (Note that the following do_updat call implies that "message" is
C copied to "in_msg(.,index)" BEFORE do_updat returns, so that
C "message" then can be reused.)
C
110       CALL do_updat(message, msg_bytes, index, update_length)
C
          CALL X$TRAN(server_vc, XT$LV0,
     +       out_msg(2,index), update_length, xmit_status(index))
          CALL orgstamp(index)
C
C To keep service speed up, we should NOT wait for XS$CMP here,
C but only sort out fatal errors. 'Expected' statuses will be
C handled later in the general all-VC loop.
C
          temp = xmit_status(index)
          IF (temp .EQ. XS$IP .OR. temp .EQ. XS$CMP .OR.
     +       temp .EQ. XS$CLR .OR. temp .EQ. XS$BVC .OR.
     +       temp .eq. XS$RST) THEN
             GO TO 50                        /* Look for more calls
          ELSE
             CALL nepr(xmit_status(index), INTS(1))
             GO TO 9000                      /* Fatal - die...
C
          ENDIF                         /* xmit-status
C
       ENDIF                            /* clear/accept
C
C
C ---   ---   ---
C Loop for running VC-s:
C This loop MUST ensure that all VC-s terminate properly and are
C released.
C - If the transmit completes and the user does not clear,
C   the server should clear;
C - if the transmit does not complete, the server should clear;
C - if resets occur, the server should clear.
C In this way the circuit will always have a clear request;
C the timer mechanism of PRIMENET (19.3 onwards) will then ensure
C that the confirming state XS$CLR is reached (with appropriate
C diagnostic), and the VC will be released.
C
200    DO 250 i = 1, pool_size
       IF (in_use(i)) THEN                  /* Skip inactive circuits
```

```
C
C Sort out VC-s, that have been cleared. Print error message if
C clearing cause/diagnostic are wrong. Free VC in any case.
C
         temp = vc_status(1, i)
         IF (temp .EQ. XS$CLR) THEN
C
            clr_cause = LT(vc_status(2, i), 8)
            clr_diag = RT(vc_status(2, i), 8)
            IF (clr_cause .NE. CC$CLR .OR. clr_diag .NE. CD$EOU)
     +           CALL nepr(vc_status, INTS(2))
            CALL freevc(i)
C
C Clear a virtual circuit that was reset. The circuit will then be
C released in a subsequent test loop.
C
         ELSE IF (temp .EQ. XS$RST) THEN
            CALL X$CLR(vc_id(i), CD$RST, statword)
            IF (statword .NE. XS$CMP) THEN
               CALL nepr(statword, INTS(1))
               GO TO 9000
            ENDIF
C
C Now look at the transmit status.
C
         ELSE
            junk = xmit_status(i)
C
C We can neglect XS$RST and XS$CLR, since they were already trapped
C in the previous vc_status test.
C
C If this transmit has XS$BVC, the cause is likely to be a clear
C before the transmit was attempted. This should already have
C been detected in the previous vc_status test, but "just in case"
C ensure the circuit becomes free.
C
            IF (junk .EQ. XS$BVC) THEN
               CALL freevc(i)
C
C If the transmit does not complete in a reasonable time, we suspect
C a "hang". Clear the circuit, and to prevent repeats, CHANGE the
C TRANSMIT status to XS$CLR. The time limit is set to 5 minutes
C (arbitrary choice).
C
```

```
            ELSE IF (junk .EQ. XS$IP .AND. age(i) .GE. INTS(5)) THEN
               CALL X$CLR(vc_id(i), CD$TMO, statword)
               IF (statword .NE. XS$CMP) THEN
                  CALL nepr(statword, INTS(1))
                  GO TO 9000
               ENDIF
               xmit_status(i) = XS$CLR
C
C Similarly, if the transmit completed but no clear arrives, again
C we clear a suspected "hang". To prevent repeats, CHANGE the
C TRANSMIT status to XS$CLR. The time limit is set to the same
C 5 minutes (arbitrary choice).
C
            ELSE IF (junk .EQ. XS$CMP .AND. age(i) .GE. INTS(5)) THEN
               CALL X$CLR(vc_id(i), CD$TMO, statword)
               IF (statword .NE. XS$CMP) THEN
                  CALL nepr(statword, INTS(1))
                  GO TO 9000
               ENDIF
               xmit_status(i) = XS$CLR
C
            ENDIF       /* Transmit status testing
C
         ENDIF          /* VC status testing
C
      ENDIF             /* VC active
C
250   CONTINUE
C
C All done. Go back to sleep
C
      GO TO 10
C
C * * * * *
C Fatal error -  die...
C
9000  PRINT 9010
9010  FORMAT  ('Fatal error!')
      CALL X$CLRA
      RETURN
C
      END
```

VC Pool handling

```
C HANDLE_VC.F77, subroutines to allocate update server VC-s
C
C History:
C 1983-09 B. Lindblad Initial coding
C 1986-11 B. Lindblad Added code to handle duplicates of a VC
C                     being used
C
C INITVC - initialize the VC flag and status database

      SUBROUTINE INITVC
C
$INSERT *>MULTI_VC.INS.F77
C
      INTEGER*2 i
C
      total_used = 0          /* No active
      next_free = 1           /* Try this one first
C
      DO 10 i = 1, pool_size  /* For all of them:
      in_use(i) = .FALSE.     /* Not in use
10    vc_id(i) = 0            /* so no known number
      RETURN
      END
C GETVC - returns whether VC block available or not
C         also flags duplicate occurence of a VC
C
      LOGICAL*2 FUNCTION GETVC(net_vc, this_index)
C
$INSERT *>MULTI_VC.INS.F77
C
      INTEGER*2 net_vc, this_index, i
C
      EXTERNAL TNOU, nepr
      INTRINSIC INTS
C
      IF (total_used .LT. pool_size) THEN
C
         getvc = .TRUE.             /* Indicate success
         this_index = next_free     /* Tell caller allocated slot
         in_use(next_free) = .TRUE.
         vc_id(next_free) = net_vc
         CALL orgstamp(next_free)   /* Set start of life
C
```

```
C Detect duplicates of this VC, free them, assuming they were
C properly cleared previously without this server detecting on time
C
         DO 5 i = 1, pool_size
         IF (in_use(i) .AND. i .NE. this_index) THEN
            CALL TNOU('Freeing duplicate use of VC', INTS(27))
            CALL nepr(vc_status(1,i), INTS(2))        /* Log last
C                                                     /* VC status
            CALL freevc(i)
         ENDIF
5        CONTINUE
C
C Now prepare for next allocation call
C
         total_used = total_used + 1
         IF (total_used .LT. pool_size) THEN
            DO 10 i = 1, pool_size
            IF (in_use(i)) GO TO 10
            next_free = i
            RETURN
10          CONTINUE
         ELSE
            next_free = 0
            RETURN
         ENDIF
C
      ELSE
         getvc = .false.      /* Indicate pool fully used
         RETURN
      ENDIF
      END
C FREEVC - return a VC block to the unused state
C
      SUBROUTINE FREEVC(index)
C
$INSERT *>MULTI_VC.INS.F77
C
      INTEGER*2 index
C
      in_use(index) = .FALSE.       /* Free it
C
C If pool WAS fully used, this must become next to use
C
      IF (total_used .EQ. pool_size) next_free = index
C
```

```
      total_used = total_used - 1    /* Out of used count
      RETURN
      END
C ORGSTAMP - set origin time for a VC (next full minute)
C
      SUBROUTINE ORGSTAMP(index)
C
$INSERT *>MULTI_VC.INS.F77
C
      INTEGER*2 index, arr(4)
      INTRINSIC INTS
      EXTERNAL TIMDAT
C
      CALL TIMDAT(arr, INTS(4))
      zero_time(index) = arr(4) + 1
      RETURN
      END
C AGE - return lifelength since origin time for a VC
C
      INTEGER*2 FUNCTION AGE(index)
C
$INSERT *>MULTI_VC.INS.F77
C
      INTEGER*2 index, arr(4)
      INTRINSIC INTS
      EXTERNAL TIMDAT
C
      CALL TIMDAT(arr, INTS(4))
      age = arr(4) - zero_time(index)
      RETURN
      END
```

A Common Network Error Message Routine

```
C NEPR.F77, routine to print network status arrays
C
C This routine gives a formatted output for various
C network status arrays. Its action depends on the number
C of words in the array.
C
C Word 1 translated to XS$xxx
C Word 2 given in decimal and split on CC$XXX/CD$XXX
C Word 3 given in decimal
C
```

```
C History:
C 1983-09 B. Lindblad Initial coding
C 1986-10 B. Lindblad Added check for undefined VC statuses
C 1986-11 B. Lindblad Added printout for new clearing diagnostic
C
      SUBROUTINE NEPR(array, no_words)
C
      INTEGER*2 array(1), no_words
C
$INSERT *>fsx_data.ins.f77
C
      INTRINSIC INTS, LT, RT
      EXTERNAL TNOUA, TODEC, TOOCT, TONL
C
      INTEGER*2 cdvalue(6), i
      CHARACTER*6 xsname(-1:14), cdname(6)
      DATA xsname/'XS$NET','XS$CMP','XS$IP','XS$BVC','XS$BPM',
     +      'XS$CLR','XS$RST','XS$IDL','XS$UNK','XS$MEM',
     +      'XS$NOP','XS$ILL','XS$DWN','XS$MAX','XS$QUE','XS$FCT'/
      DATA cdvalue/CD$SHR, CD$LNG, CD$EOU, CD$TMO, CD$RST, CD$NVC,
     +      CD$QIT/
      DATA cdname/'CD$SHR','CD$LNG','CD$EOU','CD$TMO','CD$RST',
     +      'CD$NVC','CD$QIT'/
C
      IF (no_words .LT. 1 .OR. no_words .GT. 3) RETURN
C
      IF (array(1) .LT. -1 .OR. array(1) .GT. 14) THEN
         PRINT 8990, array(1)
8990  FORMAT ('Undefined status code: ',I6)
      ELSE
         PRINT 9000, xsname(array(1))
9000  FORMAT ('Returned status code: ',A6)
      ENDIF
      IF (no_words .EQ. 1) RETURN
C
      CALL TNOUA('Second word (dec): ', INTS(19))
      CALL TODEC(array(2))
      CALL TONL
      CALL TNOUA('             (as CC$/CD$): ', INTS(26))
      CALL TOOCT(LT(array(2), 8))
      CALL TNOUA('- ', INTS(1))
      CALL TOOCT(RT(array(2), 8))
      DO 10 i = 1,6
      IF (RT(array(2),8) .EQ. cdvalue(i)) THEN
         CALL TNOUA(' (', INTS(2))
         CALL TNOUA(cdname(i), INTS(6))
         CALL TNOUA(') ', INTS(1))
      ENDIF
```

```
10    CONTINUE
      CALL TONL
      IF (no_words .EQ. 2) RETURN
C
      PRINT 9010, array(3)
9010  FORMAT ('Third word (dec): ',I5)
      RETURN
      END
```

6

FTS Programming

The File Transfer Service includes three commands: FTR, FTOP, and FTGEN. These commands comprise the user, operator, and System Administrator interfaces to FTS. In addition, the File Transfer Service provides a program interface in the form of one subroutine, FT$SUB. This chapter describes

- How to use the FT$SUB subroutine to submit, control, and determine the status of transfer requests
- Several sample programs using FT$SUB

Declaring FT$SUB

You must use the following declaration for FT$SUB in PL/I programs.

```
dcl ft$sub      entry(fixed bin(15),char(32) var,char(32) var, char(*) var,char(*)
                var,char(255) var,char(32) var, fixed bin(15),ptr,fixed bin(15),ptr,fixed
                bin(15), fixed bin(15));
```

The full calling sequence, rarely needed, is as follows.

```
call ft$sub     (key,request_name,internal_name,user_cmdl,prog_cmdl,
                '',queue,user_query,addr(request_data),1, addr(error_data),1,code);
```

Specific calling sequences needed for particular uses of FT$SUB are described in the subsequent sections of this chapter.

Defining Keys and Error Codes

To allow your program to represent FTS-specific numeric values for keys and error codes, you must include in your program a statement that defines the appropriate FTS user's file. The format of the statement depends on the language in which you are writing the program.

For a PL/I program, use the following statement:

```
%INCLUDE 'SYSCOM>FT$SUB.INS.PL1';
```

For a FORTRAN 77 (F77) program, use the following statement:

```
$INSERT 'SYSCOM>FT$SUB.INS.FTN'
```

For a FORTRAN (FTN) program, use the following statement:

```
$INSERT SYSCOM>FT$SUB.INS.FTN
```

Each statement causes the specified file to be logically included in your program. Each file contains a list of statements that define constants.

Invoking the FT$SUB Subroutine

The FT$SUB subroutine allows eight different functions to be performed for any given invocation. These transfer request functions are as follows:

- Submittal
- Parameter modification
- Canceling
- Aborting
- Holding
- Releasing
- Status retrieval of a user's requests
- Status retrieval of all requests on the system

These functions are distinguished by the first parameter of the FT$SUB subroutine. That parameter is a fixed bin(15) value. There are eight legal values for the argument, corresponding to the eight functions.

This section describes the categorization of these functions and fully describes the functions themselves. This section also describes the use of internal vs. external names. Finally, this section describes the following information returned by FT$SUB:

- The error code
- The request data structure
- The error data structure

Note

In general, user processes can operate only on their own submitted requests. There are two exceptions, however. First, a user process with the login ID SYSTEM can operate on any request. Second, any process can request status and parameter information for any request on the system.

Function Categories

The eight functions of FT$SUB may be logically grouped into four categories. The first and second functions, submission and parameter modification, are categories unto themselves — *submission* and *modification*. The next four functions belong to a category called *status change operations*, because they change only the status of a request. The final two functions belong to a category called *status retrieval operations*, because their purpose is to retrieve information about an existing transfer request, without changing the request itself.

Submission: The first function is transfer request submission. This is the only function that adds a new transfer request; the other functions operate on existing requests. The FTR command uses this function when it is submitting a new request. The parameters for the request are specified through two character strings that contain command line options.

Modification: The second function is transfer request modification. It is used by the FTR command when the –MODIFY option is specified. The parameters to be changed are specified through two character strings that contain command line options.

Status Change Operations: The next four functions, which are the cancel, abort, hold, and release functions, involve the modification of the status of a transfer request. The FTR command uses these functions when you specify the –CANCEL, –ABORT, –HOLD, or –RELEASE options.

Status Retrieval Operations: The final two functions obtain the status and parameter information for a request. The FTR command uses these functions when you specify the –STATUS or –STATUS_ALL options. One function obtains full information for a transfer request from the user who calls FT$SUB, and the other function obtains partial information for any transfer request on the system.

Transfer Request Submission

This function performs the initial submission of a transfer request. It is similar to the FTR: pathname command. To submit a transfer request, use the following calling sequence:

```
call ft$sub      (f$subm,'',internal_name,user_cmdl,prog_cmdl,'','',
                  0,addr(request_data),1,addr(error_data),1,code);
```

The following table consists of arguments that are passed to the FT$SUB subroutine as input parameters.

Input Argument	*Meaning*
F$SUBM	This key specifies that a submission operation is to be performed.
','','',0	Three null strings and a 0 stand for arguments that are not used by FT$SUB during a submission operation. You must pass these arguments exactly as shown, or FT$SUB will return an error code of E$BPAR.

internal_name,	Set to null on the initial call. Output after submission.
user_cmdl, prog_cmdl	These strings specify the command lines for the user end program. These command lines provide the details of the submission operation to FT$SUB.
addr(request_data),1	These arguments point to a request information structure (*addr(request_data)*) followed by the version number of that structure (*1*). Although the pointer itself is an input argument to FT$SUB, the structure it points to is modified by FT$SUB to reflect the results of the submission. This structure is described fully in the section The Request Data Structure, below. If you do not want to provide a request data structure, specify the address and the version number of the structure as *null(),0*. Any other combination of settings for these parameters will result in the error code E$BPAR being returned.
addr(error_data),1	These arguments point to an error information structure (*addr(error_data)*) followed by the version number of that structure (*1*). Although the pointer itself is an input argument to FT$SUB, the structure it points to is modified by FT$SUB to provide extra information in case an error occurs during the submission. This structure is described fully in the section Error Data Structure. If you do not want to provide an error data structure, specify the address and the version number of the structure as *null(),0*. Any other combination of settings for these parameters will result in the error code E$BPAR being returned.

There are two output arguments whose values are modified by FT$SUB for use by the calling program.

Output Argument	*Meaning*
internal_name	The internal name of the submitted request. This value is returned only if the returned error code is 0. You should output this field to the user after the submission operation, as does the FTR command.
code	The error code that represents the success or failure of the operation. This error code may be either a standard PRIMOS file system error code or an FTS-specific error code.

Setting Up for Submission: Before calling FT$SUB, initialize *user_cmdl* and *prog_cmdl* either to contain the user and program command lines or to pass constant strings, as appropriate.

The contents of *user_cmdl* are expected to be one or two pathnames followed by a list of options. Although the intent of *user_cmdl* is to contain a command line with options as specified by a

user, this is not a requirement. For example, the program may construct *user_cmdl* by allowing the user to select choices on a menu, with the program adding options for each choice.

prog_cmdl allows the program to provide recommendations for options should the user not specify them. Typical examples include the specification of –NO_QUERY, the setting of the –COPY and –NO_COPY switches, and the destination site (–DST_SITE). Whereas *user_cmdl* contains the source and destination pathnames, *prog_cmdl* can contain only options. The options specified in *user_cmdl* override corresponding specifications in *prog_cmdl.*

For example, suppose *user_cmdl* and *prog_cmdl* are set to the following values:

```
user_cmdl:   'IMPORTANT.MEMO JONES>IN_TRAY>MEM.AKB/001 –SRC_NTFY
               –LOG MY.LOG'

prog_cmdl:   '–DSTN_SITE BIRCH –NO_COPY –NO_QUERY
               –LOG USER_REQUESTS>REQUEST.LOG'
```

This results in a transfer request for the file IMPORTANT.MEMO to be copied into JONES>IN_TRAY>MEM.AKB/001 on the system named BIRCH (–DSTN_SITE). No temporary copy of the file is made on the local node (–NO_COPY). The requesting user is notified of the start and end of the transfer (–SRC_NTFY). A request log file called MY.LOG is also created.

Notice how –LOG MY.LOG in the *user_cmdl* overrides the –LOG USER_REQUESTS>REQUEST.LOG in *prog_cmdl.* The request is submitted with the –NO_COPY option in effect. This is because the option was present in *prog_cmdl.*

Note

You should always include the –NO_QUERY option in *prog_cmdl* to suppress user queries during request submission. The default is to query the user.

Error Recovery: The following table shows the FTS error codes that can be returned with submission. See also the section below on Error Codes for a description of general error codes.

Submission Error Codes

F$BDCL	Bad command line format
F$BDDN	Bad device name
F$BDKW	Unknown keyword
F$BDSN	Bad site name format
F$CNOP	Conflicting option
F$CPLS	Copy option applies only to local source file
F$DFAC	Destination file type invalid

F$DFNS	You did not specify the destination file
F$DLLS	Delete option applies only to local source file
F$DRNA	Device transfer from remote site is not allowed
F$DSNC	Destination site is not configured
F$DUIN	Destination user name invalid
F$DUNS	You did not specify the destination user when you requested the destination notify
F$DUOP	Duplicate option
F$FPTL	Full pathname too long
F$IDDT	Invalid defer date/time supplied
F$IDFT	Invalid destination file type
F$IFDC	Illegal file or directory conversion
F$INMS	Invalid message level
F$IPRI	Invalid priority supplied
F$ISFT	Invalid source file type
F$MCLP	Missing command line parameter
F$MBNL	Message level specified but you omitted the request log treename
F$NCLS	No copy option applies only to local source file
F$NDLS	No delete option applies only to local source file
F$NTCF	Not configured
F$PCAM	CAM files not supported on the remote system
F$PINS	Segment directory transfer to and from a Rev 1 site is not supported
F$PRAU	Priority x for administrator use only
F$PSFQ	Passworded pathname must be fully qualified
F$RLST	Request log treename same as source or target treename
F$RTIS	Remote treename incorrectly specified
F$SDSL	Source or destination site must be local
F$SFAC	Source file type invalid
F$SFNE	Source file does not exist
F$SFNS	You did not specify the source file
F$SFTD	Specified and actual source file types differ
F$SSNC	Source site is not configured
F$SUIN	Source user name invalid

F$SUNS	You did not specify source user when you requested source notify
F$TDFN	Transfer to a device as well as a file is not allowed
F$TDNS	Transferring a SEG directory to a device is not supported
F$TFNP	Transferring a file to itself is not possible
F$UNOP	Unknown option
Q$FULL	Queue full
Q$QBLK	Queue blocked
Q$QNEX	Queue does not exist
Q$UCTF	Unable to create temporary file

Example: The following example shows a simple use of FT$SUB for request submission. No declarations are provided, since they are described above and in the *Subroutines Reference Guide, Volume III.*

```
call tnoua('Enter command line: ',20);
call cl$get(command_line,160,code);
if code^=0 then return;

call ft$sub(f$subm,'',internal_name,command_line,
            '-SRC_NTFY -NO_COPY','','',0,null(),0,null(),0,
            code);
if code^=0 then return;

call tnoua('Your request is #',17);
call tnou(internal_name,length(internal_name));
return;
```

Transfer Request Modification

This function is used to change one or more parameters of a transfer request. The request must have already been submitted. This is similar in function to the command FTR –MODIFY *name*. To modify a transfer request, use the following calling sequence:

call ft$sub (f$mdfy,request_name,internal_name,user_cmdl,prog_cmdl,
",queue,0,addr(request_data),1,addr(error_data),1,code);

The table below shows arguments that are passed to the FT$SUB subroutine as input parameters.

Input Argument	*Meaning*
F$MDFY	This key specifies that a modification operation is to be performed.
request_name	This argument contains the internal or external name of the request to be modified.
internal_name	This argument contains a null string during the initial call. During subsequent calls, this argument contains either a null string or the internal name of a request from which point in the queue scanning is to start. See the section entitled Internal vs. External Names for more information.
user_cmdl, prog_cmdl	These strings specify the options that are to modify the request.
",0	A null string and a 0 stand for arguments that are not used by FT$SUB during a modification operation. You must pass these arguments exactly as shown, or FT$SUB will return an error code of E$BPAR.
queue	This string specifies the name of the FTS queue to search for the request. If all queues are to be searched, pass the null string in *queue*.
addr(request_data),1	These arguments point to a request information structure (*addr(request_data)*) followed by the version number of that structure (*1*). Although the pointer itself is an input argument to FT$SUB, the structure it points to is modified by FT$SUB to reflect the results of the operation. This structure is described fully in the section The Request Data Structure, below. If you do not want to provide a request data structure, specify the address and the version number of the structure as *null(),0*. Any other combination of settings for these parameters will result in the error code E$BPAR being returned.
addr(error_data),1	These arguments point to an error information structure (*addr(error_data)*) followed by the version number of that structure (*1*). Although the pointer itself is an input argument to FT$SUB, the structure it points to is modified by FT$SUB to provide extra information in case an error occurs during the operation. This structure is described fully in the section The Error Data Structure later in this chapter. If you do not want to provide an error data structure, specify the address and the version number of the structure as *null(),0*. Any other combination of settings for these parameters will result in the error code E$BPAR being returned.

The following are output arguments whose values are modified by FT$SUB for use by the calling program:

Output Argument	*Meaning*
internal_name	The internal name of the modified request. This value is returned only if the returned error code is 0 or F$TRPR (Transfer in progress). You should output this field to the user after the operation, as does the FTR command.
code	The error code representing the success or failure of the operation. This error code may be a standard PRIMOS file system error code, or may be an FTS-specific error code.

Setting Up for Modification: Before you call FT$SUB, initialize *user_cmdl* and *prog_cmdl* to contain the user and program command lines, or pass constant strings, as appropriate. These strings must contain only options, as the source and destination pathnames cannot be changed.

In addition, most applications will set *prog_cmdl* to the null string, since background option specifications are not normally needed during a modify operation. Any options that are specified in *prog_cmdl* will be overridden by any identical options in *user_cmdl.*

The following options (and their abbreviations) cannot be specified in either *user_cmdl* or *prog_cmdl*, because their corresponding parameters are not changeable.

-COPY	-NO_COPY
-DSTN_SITE	-QUEUE
-DSTN_FILE_TYPE	-SRC_FILE_TYPE
-HOLD	-SRC_SITE

If any of these options are present in *user_cmdl* or *prog_cmdl* during a modify operation, an error code will be returned.

Error Recovery: If any problems occur during the modification operation, a nonzero value will be returned in *code*, and the operation will not take place. Errors fall into one of the categories listed below. See the section on Error Codes below for a description of general error codes and the error codes listed in the section entitled The Error Data Structure.

- Illegal calling sequence. The arguments passed to the subroutine by the calling program are illegal. This can result in the E$BPAR (Bad.PARameter) error code being returned if incorrect version numbers are supplied for the request or error data structures, or if other arguments are not supplied as specified in the above description.
- Unrecognized option (error code F$UNOP) or keyword (error code F$BDKW).

- Unable to modify specified parameters. The modify function cannot be used to change certain parameters, described above in the list of illegal options. The following error codes are returned:

Code	*Meaning*
F$QNMD	Queue name may not be modified.
F$NCMD	NO_COPY flag may not be modified.
F$CPMD	COPY flag may not be modified.
F$SSMD	Source site may not be modified.
F$DSMD	Destination site may not be modified.
F$HDMD	Hold flag may not be modified.
F$SFMD	Source file type may not be modified.
F$DFMD	Destination file type may not be modified.

  If FT$SUB cannot locate the request with the *request_name* and *internal_name* fields that were passed to it by the calling program, it will return an error code of E$EOF (End of file). Your program should not generate the **`End of file`** error message after receiving the E$EOF error code from FT$SUB. Instead, it should generate a message such as **`Request not found`**. See the section entitled Internal vs. External Names, below, for more information.

- Insufficient access. If the request belongs to another user, and the user calling FT$SUB is not logged in as user SYSTEM, FT$SUB returns F$NERF (No Eligible Request of this name Found)
- Unable to modify the request. Two error codes may be returned if the request is in a state that prevents it from being modified. If the request is being processed, the error code F$TRPR (Transfer in progress) will be returned. If the request has been aborted, the error code F$RQAB (Request already aborting) will be returned.

Example: The following example shows a simple use of FT$SUB for request modification. No declarations are provided, as they are described above and in the *Subroutines Reference Guide, Volume III*.

```
call tnoua('Enter request name: ',20);
call cl$get(request_name,32,code);
if code^=0 then return;

call tnoua('Enter command line: ',20);
call cl$get(command_line,160,code);
if code^=0 then return;

internal_name='';
call ft$sub(f$mdfy,request_name,internal_name,command_line,
            '','','',0,null(),0,null(),0,code);
```

```
if code^=0 then return;

call tnoua('The modified request is #',25);
call tnou(internal_name,length(internal_name));
return;
```

Changing the Status of a Transfer Request

To change the status of a transfer request, use the following calling sequence:

```
call ft$sub     (key,request_name,internal_name,",",",",queue,
                user_query,addr(request_data),1,addr(error_data),1,
                code);
```

The table below shows arguments passed to the FT$SUB subroutine as input parameters.

Input Argument	*Meaning*
key	This argument specifies one of four operations to be performed on the request: F$CANC to cancel, F$ABRT to abort, F$HOLD to hold, and F$RLSE to release.
request_name	This string contains the internal or external name of the request to be operated upon.
internal_name	This argument contains a null string during the initial call. During subsequent calls, this argument contains either a null string or the internal name of a request from which point in the queue scanning is to start. See the section entitled Internal vs. External Names for more information.
",",","	Three null strings, for arguments that are not used by FT$SUB during a status change operation. You must pass these arguments exactly as shown, or FT$SUB will return an error code of F$CLMN.
queue	This string contains the name of the FTS queue to search for the request. If all queues are to be searched, pass the null string in *queue.*
user_query	This argument specifies whether the submitting user is to be queried or not. Currently, the setting of this argument has no effect.
addr(request_data),1	These arguments point to a request information structure (*addr(request_data)*) followed by the version number of that structure (*1*). Although the pointer itself is an input argument to FT$SUB, the structure it points to is modified by FT$SUB to reflect the results of the operation. This structure is described fully in the section The Request Data Structure, below.

If you do not want to provide a request data structure, specify the address and the version number of the structure as *null(),0*.

Any other combination of settings for these parameters will result in the error code E$BPAR being returned.

addr(error_data),1 These arguments point to an error information structure (*addr(error_data)*) followed by the version number of that structure (*1*). Although the pointer itself is an input argument to FT$SUB, the structure it points to is modified by FT$SUB to provide extra information in case an error occurs during the operation. This structure is described fully in the section The Error Data Structure.

If you do not want to provide an error data structure, specify the address and the version number of the structure as *null(),0*.

Any other combination of settings for these parameters will result in the error code E$BPAR being returned.

The following arguments have values that are modified by FT$SUB for use by the calling program.

Output Argument	*Meaning*
internal_name	The internal name of the affected request. This value is returned only if the returned error code is 0 or one of several possible error codes described below. You should output this field to the user after the operation, as does the FTR command.
code	The error code representing the success or failure of the operation. This error code may be either a standard PRIMOS file system error code or an FTS-specific error code.

Error Recovery: If any problems occur during the operation, a nonzero value will be returned in *code*, and the operation will not take place. These errors fall into one of the categories listed below. Also see the section Error Codes below for a description of general error codes.

- Illegal calling sequence. The arguments passed to the subroutine by the calling program are illegal. This can result in the E$BPAR (Bad Parameter) error code being returned either if incorrect version numbers are supplied for the request or error data structures, or if other arguments are not supplied as specified in the above description.
- Command lines not null. If your program does not pass null strings as the fourth and fifth arguments to FT$SUB, the error code F$CLMN (Command Lines Must be Null) will be returned.
- Unable to find specified request in database. If FT$SUB cannot locate the request with the *request_name* and *internal_name* fields that were passed to it by the calling program, it will return an error code of E$EOF (End of file). Your program should not generate the **`End of file`** error message after receiving the E$EOF error code from FT$SUB. Instead, a message such as **`Request not found`** should be generated. See Internal vs. External Names, below, for more information.

- Insufficient access. If the request belongs to another user, and the user calling FT$SUB is not logged in as user SYSTEM, FT$SUB returns F$NERF (No Eligible Request of this name Found).
- Unable to change the status of the request. Several error codes may be returned if the request is in a state that prevents its status from being changed. These error codes are

Code	*Meaning*
F$TRPR	Transfer in progress
F$RQHU	Request already put on hold by user
F$RQHO	Request already put on hold by operator
F$RQHF	Request already put on hold by FTS
F$RQAB	Request already aborted
F$RQWT	Request waiting
F$RHPR	Request held by operator

The first five codes are self-explanatory. The sixth code, F$RQWT, is produced when an attempt is made to release a request that has not been placed on hold. The last code, F$RHPR, is produced when an attempt is made by a user not logged in as SYSTEM to release a request placed on hold by an operator (user SYSTEM).

Example: The following example shows a simple use of FT$SUB for changing the status of a request. No declarations are provided, since they are described above or in the *Subroutines Reference Guide, Volume III*.

```
call tnoua('Enter request name: ',20);
call cl$get(request_name,32,code);
if code^=0 then return;

call tnou('Choose one of the following options:',36);
call tnou('',0);
call tnou('   1. Cancel the request',24);
call tnou('   2. Abort the request',23);
call tnou('   3. Hold the request',22);
call tnou('   4. Release the request',25);
call tnou('   5. Exit this program',23);
call tnou('',0);
call tnoua('Enter your choice: ',19);
call cl$get(command_line,160,code);
if code^=0 then return;
```

```
     if command_line='1' then key=f$canc;
else if command_line='2' then key=f$abrt;
else if command_line='3' then key=f$hold;
else if command_line='4' then key=f$rlse;
else if command_line='5' then return;
else do;
     call tnou('Illegal response.',17);
     code=e$ivcm;  /* Invalid command. */
     return;
     end;

internal_name='';
call ft$sub(key,request_name,internal_name,'','','','',0,
            null(),0,null(),0,code);
if code^=0 then return;

call tnoua('The affected request is #',25);
call tnou(internal_name,length(internal_name));
return;
```

Status Retrieval of a Transfer Request

To retrieve the status of a particular transfer request, use the following calling sequence:

call ft$sub (key,request_name,internal_name,"","","",queue,0,
addr(request_data),1,addr(error_data),1,code);

The information on the request will be returned in the *request_data* structure, described below. The arguments below are passed to the FT$SUB subroutine as input parameters.

Input Argument	*Meaning*
key	This argument specifies one of two forms of status retrieval: F$STAT to retrieve complete status and parameter information on a transfer request that belongs to the user calling FT$SUB; and F$STAL to retrieve partial status and parameter information on any transfer request on the system.
request_name	This string contains the internal or external name of the target request.
internal_name	This argument contains a null string during the initial call. During subsequent calls, this argument contains either a null string or the internal name of a request from which point in the queue scanning is to start. See the section entitled Internal vs. External Names, below, for more information.

",",",0	Three null strings and a 0 stand for arguments that are not used by FT$SUB during a status retrieval operation. You must pass these arguments exactly as shown, or FT$SUB will return an error code of F$CLMN.
queue	This string contains the name of the FTS queue to search for the request. If all queues are to be searched, pass the null string in *queue*.
addr(request_data),1	These arguments point to a request information structure (*addr(request_data)*) followed by the version number of that structure (*1*). Although the pointer itself is an input argument to FT$SUB, the structure it points to is modified by FT$SUB to return the status information of the request. This structure is described fully in the section The Request Data Structure, below. Any other combination of settings for these parameters will result in the error code E$BPAR being returned.
addr(error_data),1	These arguments point to an error information structure (*addr(error_data)*) followed by the version number of that structure (*1*). Although the pointer itself is an input argument to FT$SUB, the structure it points to is modified by FT$SUB to provide extra information in case an error occurs during the operation. This structure is described fully in the section The Error Data Structure, below. If you do not want to provide an error data structure, specify the address and the version number of the structure as *null(),0*. Any other combination of settings for these parameters will result in the error code E$BPAR being returned.

FT$SUB modifies the values of the arguments below for use by the calling program.

Output Argument	*Meaning*
internal_name	The internal name of the target request. This value is returned only if the returned error code is 0. If your program is retrieving status information for display purposes, you should output this field to the user along with other status and parameter information, as does the FTR command.
code	The error code representing the success or failure of the operation. This error code may be either a standard PRIMOS file system error code or an FTS-specific error code.

Error Recovery: If any problems occur during the status retrieval operation, a nonzero value will be returned in *code*, and the operation will not take place. These errors fall into one of several categories listed below. Also see Error Codes below for a description of general error codes.

- Illegal calling sequence. The arguments passed to the subroutine by the calling program are illegal. This can result in the E$BPAR (Bad Parameter) error code being returned either if incorrect version numbers are supplied for the request or error data structures, or if other arguments are not supplied as specified in the above description.
- Command lines not null. If your program does not pass null strings as the fourth and fifth arguments to FT$SUB, the error code F$CLMN (Command Lines Must be Null) is returned.
- Unable to find specified request in database. If FT$SUB cannot locate the request with the *request_name* and *internal_name* fields passed to it by the calling program, it returns an error code of E$EOF (End of file). Your program should not generate the **End of file** error message after receiving the E$EOF error code from FT$SUB. Instead, it should generate a message such as **Request not found**. See Internal vs. External Names, below, for more information.
- Insufficient access. If the request belongs to another user, and the user calling FT$SUB is not logged in as user SYSTEM, the error code F$NERF (No eligible request of this name found) is returned. This error code is returned only if *key* is F$STAT.

Example: The following example shows a simple use of FT$SUB for retrieving the status of a request. No declarations are provided, as they are described above or in the *Subroutines Reference Guide, Volume III.*

```
call tnoua('Enter request name: ',20);
call cl$get(request_name,32,code);
if code^=0 then return;

internal_name='';
call ft$sub(f$stat,request_name,internal_name,'','','','','',0,
            addr(request_data),1,null(),0,code);
if code^=0 then return;

call tnoua('The request is #',25);
call tnou(internal_name,length(internal_name));

call tnoua('Last attempt was '||request.last_date||' at ',29);
call tovfd$(divide(request.last_time,60,15));
call tnoua(':',1);
if mod(request.last_time,60)<10 then call tnoua('0',1);
call tovfd$(mod(request.last_time,60));
call tnoua(', ',2);
if request.ntry=1 then call tnou('one connection attempt.',23);
   else do;   A/* if request.ntry^=1 */
        call tovfd$(request.ntry);
        call tnou(' connection attempts.',21);
        end;  /* if request.ntry^=1 */

return;
```

Internal vs. External Names

For operations other than request submission, the FT$SUB subroutine is designed to allow one operation on one file transfer request per invocation. In some cases, you may want a calling program to perform operations over a set of transfer requests. Quite often, this set of requests constitutes one of the following:

- All of the requests in the system
- All of the requests that belong to the calling user
- All of the requests that have a certain external name that belong to the calling user

In each of these cases, FT$SUB provides a simple way of calling programs to invoke it for all of the transfer requests in the set. You do this by providing certain information to the calling program upon the completion of an FT$SUB function that can then be recycled by the program into another call to FT$SUB.

In addition, the calling program may further limit the set to only those requests in a specific FTS queue, by providing a non-null *queue* argument.

Calling Sequence Support for Iteration: In each of the calling sequences described for FT$SUB, except for the submission calling sequence, there are two arguments: *request_name* and *internal_name*. These are the second and third arguments of the FT$SUB calling sequence.

A request can reference either the name of the file being transferred, which is the *request_name*, or the number of the request, which is provided by FTS. The request number is the *internal name* of a request.

The *request_name* argument is an input-only argument. The calling program must fill this field with an identifier of the transfer request it is referencing. It can do this in one of three ways:

- By specifying the internal name of the transfer request. This limits the search by FT$SUB to a single request.
- By specifying the external name of the transfer request. This limits the search by FT$SUB to requests with that external name.
- By specifying the null string. This places no limits on the search by FT$SUB.

The *internal_name* argument is an input/output argument. It determines the search procedure for the transfer request specified in *request_name*. Before calling FT$SUB, the calling program sets *internal_name* to one of the following two character strings:

- The null string. In this case, FT$SUB will begin searching its database from the beginning of the queue. Before the initial call to FT$SUB, *internal_name* must be the null string.
- The internal name of a transfer request. This is used only after an initial call to FT$SUB has taken place. This will cause FT$SUB to search its database starting from the returned *internal_name* from the previous call.

As an output argument, *internal_name* contains the internal name of the first transfer request meeting the requirements imposed by *request_name*, *internal_name*, and *queue*. However, if FT$SUB was unable to find such a request, the error code E$EOF will be returned in *code*, and the contents of *internal_name* will be null.

Therefore, the calling program can iteratively call FT$SUB over a set of file transfer requests, optionally limiting the set to include only requests with a specified external name. It does this by

- Using the external name or the null string for *request_name*.
- Setting *internal_name* to the null string for the first call to FT$SUB, and using the returned *internal_name* argument on subsequent calls, until no more transfer requests are found.

The following table provides an example of how this procedure might work in a typical environment. Each line of the table shows the values of *code*, *internal_name*, and *request_name* before a call to FT$SUB. An imaginary call to FT$SUB occurs between each line. Indeterminate or unimportant values are indicated by —.

Code	*Internal Name*	*Request Name*
—	"	'MEMO_1'
0	'12'	'MEMO_1'
0	'16'	'MEMO_1'
E$EOF	"	—

In this example, two requests with the external name MEMO_1 were returned to the calling program. The first request had the internal name 12, and the second request had 16.

There are two further limitations that can be imposed on the set of transfer requests returned by FT$SUB. First, the requests can be limited to only those that belong to the calling user. This is normally the case. However, the restriction is lifted either if the *key* parameter in the FT$SUB call is set to F$STAL or if the calling user is SYSTEM. Second, you can limit the set to only those requests in a certain FTS queue, by specifying a non-null *queue* argument.

Operating on All Requests on the System: To operate on all requests on the system by using the F$STAL key, the calling program follows this basic form:

```
/* Initialize the request_data.valid bit to start up the do-loop,
   the found_request bit to indicate that no requests have been
   found, and initialize the internal name to null. */

request_data.valid='1'b;
found_request='0'b;
internal_name='';

/* Loop over the set of all FTS requests on the system.        */
```

```
do while(request_data.valid);  /* While there are requests.    */
   call ft$sub(f$stal,'',internal_name,'','','','',0,
               addr(request_data),1,null(),0,code);
   if code=0
      then do;                 /* If success, display the data.*/
           call display_request_data(request_data);
           found_request='1'b; /* Remember we found a request. */
           end;                /* if code=0 */
   end;                        /* do while(request_data.valid) */

if (code=e$eof & ^found_request)
   then call tnou('No requests in system.',22);
else if code=0
    then call tnou('Program error, code=0.',22);
    else call fts_error(code);  /* Display error message.*/
```

Notice that the second argument, *request_name*, is a null string. This lifts the restriction that the returned requests all have a particular external name. When FT$SUB returns a request, the calling program can determine the external name of the request by examining *request_data.extnam*, as described below.

Notice also that the program flow ignores errors returned from FT$SUB as long as FT$SUB manages to return a valid *request_data* structure (as indicated by *request_data.valid* being set to '1'b). The program is simply outputting the status of all the requests on the system and is willing to ignore error conditions that do not interrupt its scan for requests. (See the sections entitled Error Codes and The Request Data Structure for more information about the returned error code and its relationship to the *request_data* structure.)

Operating on All Requests for the Calling User: To perform an operation on all requests that belong to the user calling FT$SUB, and using the F$ABRT key, for example, the calling program should have the following form:

```
/* Initialize the request_data.valid bit to start up the do-loop,
   the found_request bit to indicate that no requests have been
   found, and initialize the internal name to null. */

request_data.valid='1'b;
found_request='0'b;
internal_name='';

/* Loop over the set of all FTS requests for this user.          */

do while(request_data.valid);  /* While there are requests.      */
   call ft$sub(f$abrt,'',internal_name,'','','','',0,
               addr(request_data),1,null(),0,code);
```

```
   if code=0
      then do;                    /* If success, display a message. */
           call display_abort_message(request_data);
           found_request='1'b; /* Remember we found a request.    */
           end;                   /* if code=0                      */
      else if code^=e$eof
              then do;  /* Error, print the error code. */
                   call fts_error(code);  /* Do error message. */
                   if request_data.valid  /* If the request data is
                                             present, */
                      then call display_request_info(request_data);
                      /* Display summary information on request. */
                   end;  /* if code^=e$eof */
           end;          /* if code^=0                             */
   end;                  /* do while(request_data.valid)           */

if (code=e$eof & ^found_request)
   then call tnou('No requests found.',18);
else if code=0
   then call tnou('Program error, code=0.',22);
```

Notice that the second argument, *internal_name*, is a null string. This lifts the restriction that the returned requests all have a particular external name. When FT$SUB returns a request, the calling program can determine the external name of the request by examining *request_data.extnam*, as described below.

Also notice how the error code returned from FT$SUB is handled. Information about the error code is always printed, but more descriptive information about the invalid request is displayed only if the request information is present. In addition, the main do-loop continues as long as a valid request was found, even if the abort operation itself failed. This way, the entire FTS database is scanned.

See the sections entitled Error Codes and The Request Data Structure for more information about the returned error code and its relationship to the *request_data* structure.

Operating on All Requests With a Specific External Name: You can perform an operation on all requests with a particular external name that belong to the user calling FT$SUB. If you use the F$HOLD key, for example, the calling program should have the following form:

```
/* Initialize the request_data.valid bit to start up the do-loop,
   the found_request bit to indicate that no requests have been
   found, and initialize the internal name to null. */

request_data.valid='1'b;
found_request='0'b;
internal_name='';
request_name='MEMO_TO_MARK';
```

```
/* Loop over the set of all FTS requests for this user.          */

do while(request_data.valid);  /* While there are requests.      */
   call ft$sub(f$hold,request_name,internal_name,'','','','','1'b,
               addr(request_data),1,null(),0,code);
   if code=0
      then do;                 /* If success, display a message. */
           call display_hold_message(request_data);
           found_request='1'b; /* Remember we found a request.   */
           end;                /* if code=0                      */
      else if code^=e$eof
              then do;  /* Error, print the error code. */
                   call fts_error(code);  /* Do error message. */
                   if request_data.valid  /* If the request data is
                                             present, */
                      then call display_request_info(request_data);
                      /* Display summary information on request. */
                   end;  /* if code^=e$eof */
           end;          /* if code^=0                             */
   end;                  /* do while(request_data.valid)           */

if (code=e$eof & ^found_request)
   then call tnou('No requests found.',18);
else if code=0
   then call tnou('Program error, code=0.',22);
```

The second argument, *request_name*, contains the external name specified by the user. This imposes the restriction that the returned requests all have the external name *request_name*.

Also notice how the error code returned from FT$SUB is handled. Information about the error code is always printed, but more descriptive information about the invalid request is displayed only if the requested information is present. In addition, the main do-loop continues as long as a valid request was found, even if the hold operation itself failed. Thus, the entire FTS database is scanned.

See the sections entitled Error Codes and The Request Data Structure for more information about the returned error code and its relationship to the *request_data* structure.

Error Codes

After a call to FT$SUB, the status of the call is returned in *code*. If the returned value is nonzero, the requested operation was not performed. However, some of the side effects of the call may have been performed, depending on the actual value set in *code*. For example, suppose a call is made to put a transfer request on hold, and the specified request is already on hold. The returned *internal_name* argument and the *request_data* structures will contain the appropriate data for the specified transfer request, even though the status of the request is not changed. This allows the calling program to determine the present status of the specified request, and to continue scanning

for other requests. In all cases, a nonzero error code is accompanied by valid information in *error_data*. See the section entitled The Error Data Structure for more information.

The list below contains error codes that may be returned by FT$SUB during normal operation. Other error codes may be returned, but these usually indicate an unusual circumstance, such as a disk error, or insufficient disk storage to submit a request. Error codes marked with * indicate operations that failed but still return valid *internal_name* and *request_data* values.

Code	*Meaning*
E$BKEY	Bad key in call
E$EOF	End-of-file
E$BPAR	Bad parameter
F$ARTL	Argument too long
F$ARTS	Argument too short
F$INEX	Invalid external name
F$NERF	No eligible request of this name found
F$NRFD	No request of this name found
F$OMOP	Only one management option allowed
F$TRPR*	Transfer in progress
F$RQHU*	Request already put on hold by user
F$RQHO*	Request already put on hold by operator
F$RQHF*	Request already put on hold by FTS
F$RQAB*	Request already aborted
F$RQWT*	Request waiting
F$RHPR*	Request held by operator
F$QNMD	Queue name may not be modified
F$NCMD	NO_COPY flag may not be modified
F$CPMD	COPY flag may not be modified
F$SSMD	Source site may not be modified
F$DSMD	Destination site may not be modified
F$HDMD	HOLD flag may not be modified
F$SFMD	Source file type may not be modified
F$DFMD	Destination file type may not be modified
F$CLMN	Command lines must be null
F$NWNA	Networks unavailable
Q$NVDB	The FTS database is invalid
Q$QNRD	FTS not ready to use

The Request Data Structure

When the *request data* structure is provided during a call to FT$SUB, the request data structure is filled in by FT$SUB to indicate the status and parameters of the specified request. However, it is filled in only if the returned error code is nonzero or one of the values listed in the section above.

To simplify the process of determining the validity of *request_data*, a *valid* bit is provided in the substructure *request_data.flags*. If this bit is '1'b, the entire request data structure contains valid information. Otherwise, the contents of *request_data* are not valid.

However, if the request data structure was returned from FT$SUB as a result of a call using the F$STAL key (Status of all requests), certain fields in *request_data* are set to null or zero values by FT$SUB before returning to the caller to prevent the FT$SUB caller getting access to other user's confidential information. These fields are

log_file
site(source_ptr)
site(destination_ptr)
user_pswd(source_ptr)
user_pswd(destination_ptr)
tree(source_ptr)
tree(destination_ptr)
file_pswd(source_ptr)
file_pswd(destination_ptr)
account_pswd(source_ptr)
account_pswd(destination_ptr)
device

The *source_ptr* and *destination_ptr* indexes are constant expressions that you might find useful in your program. They are used to access one of the two elements in several different arrays in *request_data* in a mnemonic fashion. Use the following statement in your PL1G program to set up *source_ptr* and *destination_ptr* correctly.

```
%replace source_ptr by 1,
        destination_ptr by 2;
```

The request data structure and its corresponding version number allow future changes to the structure contents, while supporting programs that call FT$SUB by using an earlier version of the structure. This structure has the following declaration, known as version 1.

```
dcl 1 request_databased,        /* Beginning of item entry.        */
      2 inumber bin,            /* Internal use only.              */
      2 iorig bin,              /* Internal use only.              */
      2 istatus bin,            /* Status of item, see below.      */
      2 reserved bin(31),       /* Reserved.       */
      2 userid char(32) var,    /* User name.                      */
      2 frnode char(32) var,    /* Reserved.    */
      2 extnam char(32) var,    /* External name for this request.*/
      2 tempfile char(32) var,  /* Temporary file and internal name
                                   for this request. */
```

```
2 idate char(8),   /* Date this request queued (YYYYMMDD).*/
2 itime bin(31),   /* Time this request queued (seconds after
                      midnight). */
2 netctl bin,      /* Internal use only.                    */
2 version bin,     /* Version number of request block (1). */
2 block_type bin, /* Type of request block.
                      0 = transfer request block. */
2 flags,           /* General flag halfword.                */
   3 valid bit(1),  /* True, '1'b, if request block
                       contains valid info, otherwise
                       set to false, '0'b.                  */
   3 mbz bit(15),   /* Pad flags to one halfword.           */
2 last_date char(8),  /* Date of last connect (YYYYMMDD).
                         Valid only if ntry>0.          */
2 last_time bin(31),  /* Time of last connect (minutes).    */
                         Valid only if ntry>0.          */
2 ntry bin,        /* Number of connection attempts.       */
2 action bin,      /* Transfer action code, see below.     */
2 priority bin,    /* Relative priority within queue       */
2 file_size bin(31), /* File size (bytes).                 */
2 defer,           /* Time before which the request will
                      not be processed.                     */
   3 time bin,
   3 date char(8),
2 runby,           /* Reserved. */
   3 time bin,
   3 date char(8),
2 runevery bin(31),  /* Reserved. */
2 pad bit(5),      /* Pad up to 16 bits.                    */
2 append bit(1),   /* Reserved.            */
2 file_exist bit(2), /* Required existence of output file. */
2 file_type bit(1),  /* Binary = '1'b, text = '0'b.        */
2 copy bit(1),     /* Make copy of file = '1'b.            */
2 delete bit(1),   /* Delete local source file after       */
                   /* transfer = '1'b. */
2 defer_set bit(1),  /* Reserved. */
2 runby_set bit(1),  /* Reserved. */
2 runevery_set bit(1),/* Reserved. */
2 notify(2) bit(1),  /* Source/dest user notify on = '1'b. */
2 log_file char(128) var,  /* User requested log file. */
                           Null if no file specified.    */
2 msg_level bin,   /* Message level for logging (1-4).     */
2 site(2) char(128) var,  /* Source/dest site addresses.  */
2 site_type(2) bin,       /* Site is Prime = 1, Other = 0.  */
2 stream char(32) var,    /* Name of request queue.         */
2 user(2) char(32) var,   /* Source/dest users.             */
2 user_pswd(2) char(32) var,  /* Reserved. */
```

```
      2 tree(2) char(128) var,       /* Source/dest file names.     */
      2 file_pswd(2) char(32) var,   /* Source/dest file passwords.*/
      2 kinship(2) char(32) var,     /* Source/dest file types.
                                        SAM, DAM, SEGSAM, SEGDAM. */
      2 mode(2) bin,         /* Reserved. */
      2 account(2) char(32) var,        /* Reserved. */
      2 account_pswd(2) char(32) var,   /* Reserved. */
      2 device char(32) var,       /* Destination device, or null. */
      2 tid bin,            /* Reserved. */
      2 response_queue char(32) var,   /* Reserved. */
      2 option char(255) var;   /* Reserved. */

%replace request_size by 825; /* Request_data size in halfwords    */
```

Status of Item: The value of *istatus* is set according to the following table.

```
/* Request status values, - permitted values of request.istatus. */

%replace wait_rqstatus by 1;   /* Waiting. */
%replace busy_rqstatus by 2;   /* In progress (transferring). */
%replace user_hold_rqstatus by 3;   /* Held by user. */
%replace operator_hold_rqstatus by 4;   /* Held by FTS operator
                                           (user SYSTEM). */
%replace user_abort_rqstatus by 5;   /* Aborted by user. */
%replace fts_hold_rqstatus by 6;   /* Held by FTS server. */
%replace operator_abort_rqstatus by 7;   /* Aborted by FTS operator
                                            (user SYSTEM). */
```

File Transfer Action Code: The value of *action* is set according to the following table.

```
/* Definition of the possible values for request.action */

%replace null_action by 0;   /* Initial value. */
%replace take_file_action by 1;   /* File is being sent. */
%replace give_file_action by 2;   /* File is being fetched. */
%replace take_jobin_action by 3;   /* Not used. */
%replace give_jobin_action by 4;   /* Not used. */
%replace take_jobout_action by 5;   /* File is being sent to a
                                       device. */
%replace give_jobout_action by 6;   /* Not used. */
```

The Error Data Structure

If a call to FT$SUB results in *code* being set to a nonzero value, you can examine the error data structure, if it is passed to FT$SUB, to pinpoint the incorrect input. This is useful if the error code indicates an error in one of the two command lines passed to FT$SUB, *user_cmdl* and *prog_cmdl*. Therefore, this structure is useful only when FT$SUB is being used either to submit or to modify file transfer requests.

The following table shows error codes that indicate an error in *user_cmdl* or *prog_cmdl*.

Code	*Meaning*
F$ARTL	Argument too long
F$ARTS	Argument too short
F$BDCL	Bad command line format
F$BDDN	Bad device name
F$BDKW	Unknown keyword
F$BDSN	Bad site name format
F$CNOP	Conflicting option
F$CPLS	Copy option only applies to local source file
F$CPMD	COPY flag may not be modified
F$DFAC	Destination file type invalid
F$DFMD	Destination file may not be modified
F$DFNS	You did not specify the destination file
F$DLLS	Delete option only applies to local source file
F$DRNA	Device transfer from remote site not allowed
F$DSMD	You may not modify the destination site
F$DSNC	Destination site is not configured
F$DUIN	Destination user name invalid
F$DUNS	You did not specify destination user when you requested destination notify
F$DUOP	Duplicate option
F$FPTL	Full pathname too long
F$HDMD	HOLD flag may not be modified
F$IDDT	Invalid defer date/time supplied
F$IDFT	Invalid destination file type
F$IFDC	Illegal file or directory conversion
F$INMS	Invalid message level
F$IPRI	Invalid priority supplied
F$ISFT	Invalid source file type
F$MCLP	Missing command line parameter
F$MBNL	Message level specified but request log treename omitted
F$NCLS	NO_COPY option only applies to local source file

F$NCMD	NO_COPY flag may not be modified
F$NDLS	No delete option applies only to local source file
F$NTCF	Not configured
F$OMOP	Only one management option allowed
F$PCAM	CAM files not supported on the remote system
F$PINS	Segment dir. transfer to or from a Rev 1 site is not supported
F$PRAU	Priority x for administrator use only
F$PSFQ	Passworded pathname must be fully qualified
F$RTIS	Remote treename incorrectly specified
F$QNMD	Queue name may not be modified
F$RLST	Request log treename same as source or target treename
F$SDSL	Source or destination site must be local
F$SFAC	Source file type invalid
F$SFMD	Source file type may not be modified
F$SFNE	Source file does not exist
F$SFNS	Source file has not been specified
F$SFTD	Specified and actual source file types differ
F$SSMD	Source site may not be modified
F$SSNC	Source site is not configured
F$SUIN	Source user name invalid
F$SUNS	Source user not specified when source notify requested
F$TDFN	Transfer to a device as well as a file is not allowed
F$TDNS	Transferring a SEG directory to a DEVICE is not supported
F$TFNP	Transferring a file to itself is not possible
F$UNOP	Unknown option

However, to simplify programming, the error data structure itself has a bit that indicates the validity of its data. This is the *valid* bit. It is set to 0 by FT$SUB whenever FT$SUB is unable to fill the rest of the structure with valid data. Otherwise, *valid* is set to 1, and the rest of the structure contains valid data.

Before calling FT$SUB, you should initialize *start_ptr* and *end_ptr* to 0. FT$SUB adds the starting and ending locations in the command line of the invalid option or keyword. After FT$SUB returns a nonzero error code, if *comm_line* in the error data structure is nonzero, you should display the specified command line to the user, and use the *start_ptr* and *end_ptr* values to indicate which part of the command line was in error. If these values were initialized to 0, then they will contain values between 1 and the length of the invalid command line, inclusive.

The error data structure and its corresponding version number are designed to allow future changes to the structure contents, while supporting programs that call FT$SUB by using an earlier version of the structure. This structure has the following declaration, known as version 1.

```
dcl 1 error_databased,
      2 valid bit(1),  /* '1'b if structure info valid, else '0'b. */
      2 mbz bit(15),   /* Bit padding. */
      2 version bin,   /* Version number of buffer, must be 1.     */
      2 errcode bin,   /* Copy of FT$SUB return code.              */
      2 comm_line bin, /* 1 = Display program command line with ptrs.
                          2 = Display user command line with ptrs.
                          0 = Don't display any command line.      */
      2 start_ptr bin, /* If comm_line ^= 0, points to start of
                          area on command line causing the error. */
      2 end_ptr bin,   /* If comm_line ^= 0, points to end of
                        area on command line causing the error.
                        FT$SUB adds to the value supplied to
                        it in start_ptr and end_ptr.      */
      2 text char(160) var, /* Optional explanatory text.          */
      2 proc char(32) var;  /* Name of procedure detecting
                               the error.                          */

%replace error_size by 104;  /* Size of error_data in
                                  16-bit halfwords.  */
```

You can use the *text* and *proc* strings during a call to the system error printing subroutine ERRPR$. They provide additional visual feedback on the error, even if the problem was not in one of the command lines.

Example

Here is a sample program using FT$SUB.

```
C ft$sub_test.f77. Submit a local file to FTS.
C
C This program reads the source filename, destination filename, and
C destination site from the user terminal, and then submits this
C request by calling FT$SUB.
C
      PROGRAM main
C
$INSERT SYSCOM>FT$SUB.INS.FTN
C
      INTEGER*2 empty_string(17),   /* for char*32 var
     +          empty_string1(129), /* for char*255 var
     +          options(21)         /* for char*40 var
```

```
C
      INTEGER*2 int_name(17), int_length  /* for char*32 var
      CHARACTER*32 int_string
      EQUIVALENCE (int_length,int_name(1)), (int_string,int_name(2))
C
      INTEGER*2 cmd_line(41), cmd_length  /* for char*80 var
      CHARACTER*80 cmd_string
      EQUIVALENCE (cmd_length,cmd_line(1)), (cmd_string,cmd_line(2))
C
C Define request block storage.
C
      INTEGER*2 request_block(900)
C
C Define error block storage and structure.
C
      INTEGER*2 error_data(104),    /* Total size
     +      e_valid,
     +      e_text_len,
     +      e_proc_len
      CHARACTER*160 e_text
      CHARACTER*32 e_proc
      EQUIVALENCE (e_valid, error_data(1)),
     +      (e_text_len, error_data(7)),
     +      (e_text, error_data(8)),
     +      (e_proc_len, error_data(88)),
     +      (e_proc, error_data(89))
C
      INTEGER*2 return_code
C
      COMMON /ALAN/empty_string,
     +      empty_string1,
     +      options,
     +      int_name,
     +      cmd_line,
     +      request_block,
     +      error_data
C
C Initialize several FT$SUB parameter strings.
C
      CALL init_str(empty_string, empty_string1, int_name, options)
C
C Now get the user command line
C
      CALL get_line(cmd_string, cmd_length)
C
C Dispatch the request!
C
```

```
      CALL FT$SUB(F$SUBM,
     +      empty_string,
     +      int_name,
     +      cmd_line,
     +      options,
     +      empty_string1,
     +      empty_string,
     +      F$NQRY,
     +      LOC(request_block), 1,
     +      LOC(error_data), 1,
     +      return_code)
C
C If the submission was OK, type out the internal name, otherwise
C dump error message.
C
      CALL TNOUA('Submitted as ', INTS(13))
      CALL TNOU(int_string, int_length)
C
      CALL TNOUA('Submission error code: ', INTS(23))
      CALL TODEC(return_code)
      CALL TONL
      IF (AND(e_valid,:100000) .NE. 0) THEN
         CALL TNOUA(e_text, e_text_len)
         CALL TNOUA(' (', INTS(2))
         CALL TNOUA(e_proc, e_proc_len)
         CALL TNOU(') ', INTS(1))
      ENDIF
C
      CALL EXIT
      END
C GET_LINE
C
      SUBROUTINE get_line(string, length)
C
      CHARACTER*80 string
      INTEGER*2 length
C
C Get the source file, the destination file, and the destination site
C
      PRINT 9000
      READ (1,'(A80)') string
      length = 80
C
9000  FORMAT ('Give srcfile, destfile, -ds destsite')
      RETURN
      END
C INIT_STR
```

```
C
C This routine initializes several FT$SUB parameter strings.
C
      SUBROUTINE init_str(e, el, int, o)
C
      INTEGER*2 e(1), el(1), int(1), o(1)
      INTEGER*2 defopt(20)          /* Max 40 chars
C
      DATA defopt/'-log demo.log                              '/
C
      e(1) = 0                  /* Set null length
      el(1) = 0
      int(1) = 0
C
      o(1) = 13                 /* Actually used
      DO 10 i = 1, 20
10    o(i + 1) = defopt(i)
C
      RETURN
      END;
```

Appendices

A

X.25 Programming Guidelines

This appendix discusses

- PRIMENET's X.25 support
- Optional X.25 fields and the IPCF parameters that control them
- X.24 protocol restrictions
- The Protocol ID and User Data fields
- X.25 facilities
- Called address extension
- Window and packet size negotiation

PRIMENET's X.25 Support

With the release of Rev. 21.0, Prime supports the 1984 X.25 standard, as well as the Connection-Oriented Network Service of the International Organization for Standardization (ISO CO.NS). The major advantage of the new standards for most Prime users is that they allow connections between Prime and non-Prime systems over full-duplex (FDX) and LAN300 links. The following kinds of connections support the new standard:

- Between two Rev. 21.0 Prime systems over a RINGNET, LAN300, FDX, or half-duplex (HDX) line
- Between a Rev. 21.0 Prime system and a non-Prime system that supports 1984 X.25, over a LAN300 or FDX line
- Between a Rev. 21.0 Prime system and a 1984 X.25 PSDN

For a complete description of the X.25 and ISO CO.NS standards, refer to the following documents:

- CCITT X.20-X.32 1984 (Red Book) CCITT X.1-X.29 1980 (Yellow Book)
- ISO/DIS 8348, *Information Processing Systems — Data Communications — Network Service Definition*

- ISO/DIS 8208, *Data Communication — X.25 Packet Level Protocol for Data Terminal Equipment*
- ISO/DP 8878 or ISO/TC97/SC6 N3438, *Use of X.25 to Provide the OSI Connection-Oriented Network Service*
- ISO/DP 8348/DAD2, *Addendum to the Network Service Definition Covering Network Layer Addressing*

Besides Prime-to-non-Prime FDX and LAN connections, PRIMENET supports the following X.25 1984 functionality:

- Incoming calls may be routed to a specific user process on the local system. (Calls may continue to be routed to a port, as in X.25 1980.) Similarly, outgoing calls may be directed to specific processes on remote systems. The mechanism for this routing is the Called Address Extension, described later in this appendix.
- Calling and called applications may negotiate packet and window sizes more flexibly.
- The facilities field can be a maximum 109 bytes long in X.25 1984. The maximum length in X.25 1980 is 63 bytes.
- Interrupt packets, described in the sections on X$TRAN and X$RCV in Chapter 4, IPCF Subroutines, can be a maximum of 32 bytes long in X.25 1984. The maximum length in X.25 1980 is 1 byte.
- In X.25 1984, a maximum of 128 bytes of user data may be included when a Fast Select call is cleared, even if the call was previously accepted. This was possible in X.25 1980 only in response to a call request.
- Facilities can now appear in Clear Request packets.

Optional Fields of X.25 Packets and IPCF Parameters

The long-form IPCF subroutines (XLxxxx) contain parameters that allow you to control the contents of the following optional fields within certain X.25 packets:

- Protocol ID
- User Data
- Facilities

The Fast Select subroutines (X$Fxxx) also allow you to specify user data. Packets that contain protocol ID and/or User Data fields are called *extended* packets. The following table shows which subroutines and parameters allow you to specify these fields and which types of packets can be extended. In the table, *prid* indicates the Protocol ID; *udata* and *clrudat* indicate user data; and *fcty* indicates facilities. These fields are described in more detail later in this appendix.

Subroutine	*Parameters*	*Type of Packet*
XLASGN	prid, udata	
XLCONN	prid, udata, fcty	Call Request
X$FCON	udata	Fast Select Call Request
XLGA$	prid, udata, fcty	Call Connected
XLGCON	prid, udata, fcty	Incoming Call
XLGC$	prid, udata, fcty	
XFGC	udata	
XLACPT	prid, udata, fcty	Call Accept
X$FACP	udata	Fast Select Call Accept
XLCLR	clrudat, fcty	Call Clear
X$FCLR	clrudat	Fast Select Call Clear
XLGI$	prid, udata, fcty	Clear Indication
XLUASN	prid, udata	
XLGVVC	prid, udata, fcty	

X.25 Protocol Restrictions

The X.25 1984 protocol is not supported over links to pre-Rev. 21.0 nodes or links to 1980 PSDNs. Several problems can arise when X.25 1984 features are used over a path containing either of these types of links. The following conditions may cause problems:

- If you include more than 63 bytes of facilities in a call request, accept, or clear, a 1980 PSDN may reject the call or truncate the facilities. A pre-Rev. 21.0 Prime system may also reject the call.
- If you include 1984-only facilities (for example, Called Address Extension) in a call request, accept, or clear, a 1980 PSDN may reject the call. A pre-Rev. 21.0 Prime system will probably accept the call and pass the unfamiliar facilities unchanged. However, if the facilities include a Called Address Extension, a call to a pre-Rev. 21.0 Prime system will probably not work, since the called system will not know to which process the call should be directed.
- If you send an Interrupt packet longer than one byte to a pre-Rev. 21.0 system, all but the first byte of the packet may be lost. If the long Interrupt packet is sent over a Route-through path, the Route-through Server may reset the circuit. (Neither the Rev. 21.0 X$TRAN nor the Rev. 21.0 Route-through Server will allow you to send an Interrupt packet longer than one byte to a 1980 PSDN.)

- If you use an extended clear packet to clear an accepted fast-select circuit over a 1980 X.25 link, the call will be cleared, but any facilities or user data will probably be lost.

The Protocol ID and User Data Fields

User data may be included in the following packets:

- Call Request
- Fast Select Call Request
- Call Accept
- Fast Select Call Accept
- Call Clear
- Fast Select Call Clear

User data is named call user data when it is in a Call Request packet, user data when it is in a Call Accept packet, and clear user data when it is in a Call Clear packet. The maximum size of the Call User Data field is 128 bytes for Fast Select connections, otherwise 16 bytes.

The CCITT X.3, X.28, and X.29 standards regulate the communications protocol used in PSDNs. These standards use the first four bytes of the Call User Data field as a *protocol identifier* field. The remainder of the call user data can be used for call data.

X.25 acknowledges the existence of X.3 and other protocols by setting rules for the two most significant bits of the first byte of the call user data and the called user data. For most user-written applications, these two bits should be set to 1. There is no restriction on these bits in clear user data.

Most IPCF subroutines recognize the Protocol ID field by handling it as a separate parameter (*prid*). PRIMENET uses the 4-byte Protocol ID field to convey port number information in call requests. Thus, this 4-byte field for Prime-to-Prime virtual circuits cannot be used in applications, unless you are using the Called Address Extension facility instead of a port number. In contrast, the protocol identifier argument is not used as part of clear user data in the clear routines, X$FCLR and XLCLR.

When initiating a call request, an application can provide an array to retrieve any returned user data. For connect, the array provides accept and/or clear user data. For accept, the array provides clear user data (if any). The array corresponds to the X.25 User Data field, and thus includes the Protocol ID field. Figure A-1 below shows how the call, called, and clear user data, as defined by X.25, are split into the Protocol ID field and User Data field for the IPCF subroutines. For the Fast Select routines (X$Fxxx), the User Data field is not split into two parts (protocol ID and user data). The figure also shows the retrieve-data array, optionally provided in the call request routines, XLCONN and X$FCON, and the accept routine, XLACPT.

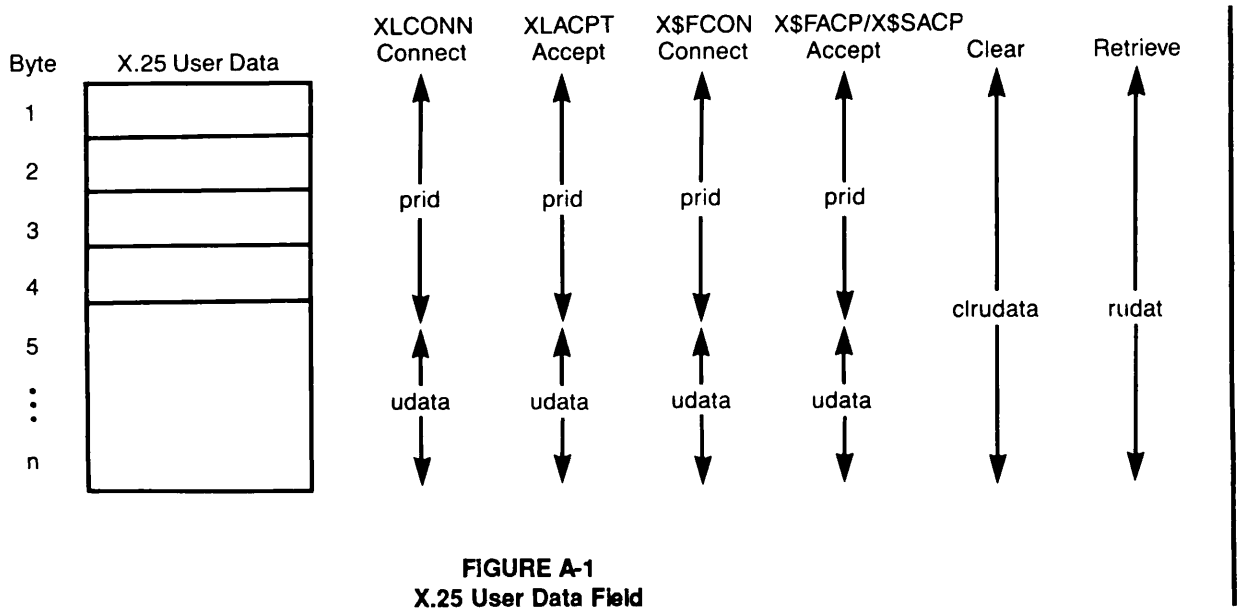

FIGURE A-1
X.25 User Data Field

X.25 Facilities

Facilities fields may be included in Call Request, Call Accept, and Call Clear packets. In X.25 1984, facilities may be up to 109 bytes long; in X.25 1980, the maximum size is 63 bytes.

Over X.25 1980 connections, PRIMENET explicitly supports the following facilities:

- Fast Select
- Window and packet size negotiation
- Acceptance of a call with reverse charging

Over X.25 1984 connections, the following additional facilities are supported:

- Called address extension
- Window and packet size negotiation for Remote Login calls

Note

Facilities that are not explicitly supported are passed on unchanged by PRIMENET.

Called Address Extension

In 1980 X.25 connections, incoming calls are directed to ports, and user processes receive calls by assigning ports. Calls are directed to ports as follows:

- The Called Address in the Incoming Call packet must match one of the X.25 addresses defined for the local node in the network configuration.
- The protocol ID field is checked. If the first byte is hex C0, then the second byte is assumed to be a port number.

In 1984 X.25 connections, ports can still be used, but the Called Address Extension can also be used to route calls to user processes, circumventing the port mechanism. The Called Address Extension is an optional part of the facilities field of Incoming Call packets, and can have up to 41 hexadecimal digits. The Called Address, together with the appended Called Address Extension, comprises the Network Service Access Point (NSAP) address. Over X.25 1984 links, a process that has sufficient privilege can use the XLASGN routine to register interest in incoming calls with any of the following characteristics:

- Called Address matches a configured X.25 address for the local node.
- Called Address starts with a given prefix.
- Called Address ends with a given suffix.
- Called Address Extension starts with a given prefix.
- Port number matches a given number.
- Protocol ID starts with a given prefix.
- User data starts with a given prefix.

Only certain combinations of these criteria can be used; refer to the description of XLASGN in Chapter 4, IPCF Subroutines, for further information.

To place a call to a remote process using the Called Address Extension, place the Called Address Extension in the *fcty* field in your XLCONN call.

Window and Packet Size Negotiation

As mentioned in Chapter 3, IPCF Programming Principles, you can increase throughput by changing a virtual circuit's window and packet sizes. Window and packet sizes can be negotiated between calling and called applications, subject to the following restrictions:

- CONFIG_NET imposes certain limits on window and packet sizes. The maximum supported packet size is configurable to 512, 1024, or 2048 for RINGNET, and is 256 bytes for synchronous lines. Refer to the *PRIMENET Planning and Configuration Guide* for more information.
- Some PSDNs impose limits on window and packet sizes.

Window and packet sizes are negotiated by means of the facilities fields in the Call Request and Call Accept packets. The calling application sets the Call Request facilities by specifying a *fcty* argument in the call to XLCONN. The called application sets the Call Accept facilities by specifying a *fcty* argument in the call to XLACPT. The specific numbers that should be placed into the *fcty* field are listed later in this chapter.

If the calling application uses the XK$FCT key when calling XLCONN, PRIMENET supplies a predefined facilities field for the Call Request packet. This field has one value for direct Prime-to-Prime links and other values for links through PSDNs. The actual value varies with the PSDN. The only risk in using this strategy occurs when you run an international link over multiple PSDNs. If your local PSDN link increases either window or packet size, the international gateway to the next PSDN may reject the packet facility request and clear your call attempt.

If you provide your own facility field, you must be sure to comply with the restrictions listed earlier in this section. For a Prime-to-Prime connection, if you request a packet or window size greater than that supported by PRIMENET over the medium you are using, PRIMENET will reduce these values to the maximum supported by the medium.

A Call Accept packet without a facilities field grants the caller's required window and packet size. If the called application requires different values, it should supply a facilities field in the call to XLACPT. The called application may negotiate only values closer to the X.25 defaults of window size 2 and packet size 128.

Facilities Field Values

The following table defines X.25 international facility formats for determining window and packet sizes. The window and packet size facility elements consist of 3 bytes each. You can include either or both in the facility field, and combine them with other facility elements.

Facility Element	*First Byte*	*Second Byte*	*Third Byte*
Window size	01000011	00000xxx	00000yyy
Packet size	01000010	0000zzzz	0000vvvv

Parameter	*Meaning*
xxx	Window size, binary coded (000 not allowed), for transmission from called node to calling node
yyy	Window size, binary coded (000 not allowed), for transmission from calling node to called node
zzzz	Packet size code (see below) for transmission from called node to calling node
vvvv	Packet size code (see below) for transmission from calling node to called node.

The packet size code is the binary coded logarithm base 2 of the packet size, expressed in bytes.

zzzz/vvvv	0100	0101	0110	0111	1000	1001	1010
Size (bytes)	16	32	64	128	256	512	1024

The description above limits the window size to a maximum of 7 bytes, which is the maximum permitted for normal X.25 sequence numbering, and also the maximum value currently supported by PRIMENET. For example, suppose you wanted a window size of 6 and a packet size of 256 for the calling node's transmissions, and for the called node's transmissions you wanted the normal default values of 2 and 128. The facility field required would be the following 6 bytes:

01000011 00000010 00000110 01000010 00000111 00001000

The encoding of Fast Select and reverse charging is as follows:

First Byte	*Second Byte*
000000001	fr00000c

Parameter	*Meaning*
f	1 = Fast Select 0 = not Fast Select, r is then ignored
r	1 = Fast Select, restricted-response (clear only) 0 = Fast Select, call may be accepted
c	1 = reverse-charged call 0 = caller pays

The encoding for Calling/Called Address extension is shown below. This must be preceeded by a marker:

First Byte	*Second Byte*
00000000	00001111

	First Byte	*Second Byte*	*Third Byte*	*Remaining Bytes*
Calling Address Extension	11001011	nnnnnnnn	ccmmmmmm	m BCD digits, two per byte
Called Address Extension	11001001	nnnnnnnn	ccmmmmmm	m BCD digits, two per byte

Parameters	*Meaning*
cc	00 = Entire OSI NSAP address 01 = Partial OSI NSAP address 10 = Non-OSI value 11 = Reserved You should use the non-OSI value, unless your application is designed to participate in an OSI network and you are party to an address registration scheme.
mmmmmm	No. of BCD digits following this byte. Legal values are 0 through 40.
nnnnnnnn	A value depending on m, the number of BCD digits in the address. The value is computed from 1 + (m + 1)/2 Legal values are 1 through 21.

B

FTS Error Messages

This appendix lists and explains the error messages produced by FTS. The messages are divided into two sections as follows:

- General error messages that any FTS utility may produce.
- Messages produced by the FTR utility. Many of the messages described here are those that a programmer using FT$SUB would need to know.

General Error Messages

The following errors can occur in any of the FTS utilities.

Argument too long. (F$ARTL)

You specified an argument that was longer than the maximum argument length allowed.

Argument too short. (F$ARTS)

You specified an argument that was shorter than the minimum argument length allowed.

FTS not ready for use. (Q$QNRD)

The FTS database has not been initialized with the FTGEN INITIALIZE_FTS command.

No help is available on the subject xxxxxx

You have requested help regarding a topic for which online documentation is unavailable.

The FTS database is invalid. (Q$NVDB)

The FTS database has been corrupted, or an FTGEN INITIALIZE_FTS command has not been performed after FTS installation.

FT$SUB Error Messages

The following messages may occur when you are using FT$SUB.

Bad command line format. (F$BDCL)

The format of the command line is incorrect. For example, you should type FTR LETTER –CANCEL instead of FTR –CANCEL LETTER.

Bad device name. (F$BDDN)

You typed an invalid –DEVICE name. LP is the only correct device name.

Bad site name format. (F$BDSN)

The site name must be a valid one and adhere to the PRIMOS filenaming standard. See the *Prime User's Guide* for more information on naming standards.

CAM files not supported on the remote system. (F$PCNT)

You are trying to transfer a CAM file to a site which is not capable of handling such files (that is, a pre-Rev. 20 site or a Rev. 20 site with disks created by a pre-Rev. 20.0 revision of the MAKE utility).

Command lines must be null. (F$CLMN)

You specified an FTR management option and included extraneous items on the command line. Reissue the command, ensuring that the command line contains only the items specifically documented for the management option you are using.

Conflicting options. (F$CNOP)

The options you specified conflict. For example, you specified a request with both –COPY and –NO_COPY when it must be either one or the other option.

Copy flag may not be modified. (F$CPMD)

You tried to modify the copy flag, which is not allowed. For example, you may have entered the following command:

```
FTR –MODIFY 2 –COPY
```

Copy option only applies to local source file. (F$CPLS)

You specified this option when you were fetching a file from a remote site, which is not allowed. This option is relevant only when you are sending local files.

Delete option only applies to local source file. (F$DLLS)

You specified this option when you were fetching a file from a remote site, which is not allowed. This option is relevant only when sending local files.

Destination file access mode invalid. (F$DFAC)

The destination file type is not one that FTS supports. Only SAM, DAM, SEGSAM, SEGDAM, and CAM files are supported.

Destination file has not been specified. (F$DFNS)

You did not specify a destination pathname.

Destination file may not be modified. (F$DFMD)

You tried to modify the destination file option, which is not allowed.

Destination site is not configured. (F$DSNC)

The destination site has not been configured in the FTS configuration for your site.

Destination site may not be modified. (F$DSMD)

You tried to modify the destination site name, which is not allowed.

Destination user name invalid. (F$DUIN)

You used an invalid destination user ID. The user ID must conform to the PRIMOS standard for user IDs. See the *Prime User's Guide* for more information.

Destination user not specified when destination notify requested. (F$DUNS)

You did not specify a destination user (with –DSTN_USER). You must specify a destination user if you use the FTR –DSTN_NTFY option when sending a file to a remote site.

Device transfer from remote site not allowed. (F$DRNA)

You attempted to fetch a file from a remote site and send it a local device, which is not allowed.

Duplicate option. (F$DUOP)

You duplicated one or more options. For example, the following command duplicates the –SRC_NTFY option, which is not allowed:

```
FTR <ASH>TREE <ELM>BRANCH -SRC_NTFY -DSTN_USER CLARKE -SRC_NTFY
```

Full pathname too long. (F$FPTL)

You exceeded the maximum pathname length of 128 characters.

Hold flag may not be modified. (F$HDMD)

You tried to modify the hold flag, which is not allowed. For example, you may have entered the following command:

```
FTR –MODIFY 3 –HOLD
```

Illegal file or directory conversion. (F$IFDC)

You used the –SRC_FILE_TYPE or –DSTN_FILE_TYPE options in an invalid combination.

Invalid defer date/time supplied. (F$IDDT)

You have specified the date/time value for the –DEFER option to FTR in an invalid format.

Invalid priority supplied. (F$IPRI)

The priority value you have specified is not in the valid range from 1 through 9.

Invalid destination file type. (F$IDFT)

You did not specify a correct destination file type.

Invalid external name. (F$INEX)

The name you specified for the request name is not a valid PRIMOS filename. See the *Prime User's Guide* for the correct filename syntax.

Invalid message level. (F$INMS)

You specified an FTS log file message level other than the following valid ones: NORMAL (1), DETAILED (2), STATISTICS (3), and TRACE (4).

Invalid source file type. (F$ISFT)

You specified an incorrect source file type.

Message level specified but request log treename omitted. (F$MBNL)

You specified the –MSGL_LEVEL option with a specific level (for example, DETAILED), but you did not specify a log filename.

Missing command line parameter. (F$MCLP)

The command line has a required parameter missing. For example, the following did not specify a destination user ID with the –DSTN_USER option:

```
FTR <ELM>TREE <ASH>BURN –SRC_NTFY –DSTN_USER
```

Networks unavailable. (F$NWNA)

You tried to use FTR to submit a request, but the network has been shut down or not configured.

No Copy flag may not be modified. (F$NCMD)

You tried to modify the NO_COPY flag, which is not allowed. For example, you may have entered the following command:

```
FTR –MODIFY 2 –NO_COPY
```

No copy option only applies to local source file. (F$NCLS)

You specified this option when you were fetching a file from a remote site, which is not allowed. This option is relevant only when you send local files.

No delete option only applies to local source file. (F$NDLS)

You specified this option when you were fetching a file from a remote site, which is not allowed. This option is relevant only when you are sending local files.

No eligible request of this name found. (F$NERF)

You attempted to modify, abort, release, hold, or cancel requests with the specified name without success because either they do not exist or they are in an ineligible state. For example, you would receive this error if you tried to hold a request that was already HELD.

No request of this name found. (F$NRFD)

You specified a nonexistent request name when you performed a –DISPLAY or –STATUS of a particular request. Check that you specified the right name, or use the request number in the command.

No requests queued. (F$NRQD)

You tried to list the contents of a queue that is empty.

Not configured. (F$NTCF)

You specified a site, server, or queue that had not been configured with FTGEN.

Only one management option allowed. (F$OMOP)

You specified more than one management option, which is not allowed. For example, FTR –ABORT LETTER –CANCEL is not allowed.

Passworded pathname must be fully qualified. (F$PSFQ)

You did not specify a passworded pathname from the directory down to the filename, and you did not enclose the complete pathname, including the password, in single quotes.

Priority x for administrator use only. (F$PRAU)

You have specified a priority value which is reserved for administrator use only. You should specify a value in the range from 1 through 7.

Queue blocked. (Q$QBLK)

You tried to submit a request to a queue that has been blocked with the FTGEN BLOCK_QUEUE command. The queue must be unblocked with the FTGEN UNBLOCK_QUEUE command so that requests can be accepted.

Queue does not exist. (Q$QNEX)

You tried to submit a request to a request queue that has not been configured with FTGEN.

Queue full. (Q$FULL)

The request queue is full.

Queue name may not be modified. (F$QNMD)

You tried to modify the –QUEUE option, which is not allowed.

Remote treename incorrectly specified. (F$RTIS)

You specified a pathname for the destination site that did not include disk and directory names. You must specify the entire pathname in the command line.

Request already aborting. (F$RQAB)

You tried to use an FTR management option (except –STATUS or –DISPLAY) on an aborting request, which is not allowed.

Request already put on hold by FTS. (F$RQHF)

You tried to hold or abort a request that has already been held by FTS, which is not allowed.

Request already put on hold by operator. (F$RQHO)

You tried to hold or abort a request that has already been held by an operator, which is not allowed.

Request already put on hold by user. (F$RQHU)

You tried to hold or abort a request that has been put on hold already.

Request held by operator. (F$RHPR)

You tried to release an operator-held request.

Request log treename same as source or target treename. (F$RLST)

You specified a log filename that is not different from the source or destination pathname.

Request waiting. (F$RQWT)

You tried to release or abort a waiting request, which is not allowed.

Segment dir. transfer to/from a Rev 1 site is not supported. (F$PINS)

You tried to transfer a SEG file to or from a REV 1 FTS site, which is not allowed.

Source file access mode invalid. (F$SFAC)

The source file type is not one that FTS supports. Only SAM, DAM, SEGSAM, SEGDAM, and CAM files are supported.

Source file does not exist. (F$SFNE)

You tried to transfer a nonexistent file. Check to see that you specified an existing file for the transfer request.

Source file has not been specified. (F$SFNS)

You did not specify a file to be sent or fetched in the transfer request.

Source file type may not be modified. (F$SFMD)

You tried to modify the source file type, which is not allowed.

Source or destination site must be local. (F$SDSL)

You cannot make file transfers between two remote sites. You can transfer requests in loopback either on a local site or between local and remote sites only.

Source site is not configured. (F$SSNC)

You specified a source site that has not been configured with FTGEN.

Source site may not be modified. (F$SSMD)

You tried to modify the source site, which is not allowed.

Source user name invalid. (F$SUIN)

You specified an incorrect source user ID. User IDs must conform to the PRIMOS naming standard. See the *Prime User's Guide* for more information.

Source user not specified when source notify requested. (F$SUNS)

You did not specify the source user (with –SRC_USER) when fetching a file from a remote site. You must specify a source user if you use the –SRC_NTFY option in the command line.

Specified and actual source file types differ. (F$SFTD)

You used the –SRC_FILE_TYPE option, but the file type that you specified differs from the actual source file type.

Transfer in progress. (F$TRPR)

You tried to release, cancel, modify, or hold a transferring request, which is not allowed.

Transfer rejected: Problem with remote file.

The file could not be opened for transfer. Check to be sure you have spelled the filename correctly. Also, check with the remote site to be sure that there is sufficient disk space for the file you are transferring, and that the remote server has access to the destination directory.

Transfer rejected: Will not retry.

This message, which usually appears along with other messages, indicates that an error has occurred and that retrying the transfer will not help until the error is corrected. The request is put on hold.

Transfer to a device as well as a file is not allowed. (F$TDFN)

You cannot specify both a destination file and a destination device (–DEVICE LP) in one transfer request.

Transferring a file to itself is not possible. (F$TFNP)

You used only one filename for two files in a transfer request. The source and destination file cannot be the same.

Transferring a SEG directory to a DEVICE is not supported. (F$TDNS)

You cannot print a SEG file type on a remote line printer.

Unable to create temporary file. (Q$UCTF)

The number of temporary files in the FTSQ* directory may have reached the maximum number as a result of queued requests. The operator should investigate the possibility of any old requests being cancelled. In addition, check to be sure the correct FTS-related access rights are assigned. Check to be sure that the disk containing FTSQ* is not full, and that the quotas on FTSQ* and its disk are set to appropriate values.

Unknown keyword. (F$BDKW)

An argument on the command line is unknown. For example, the following shows an unknown keyword, –FRED. The command line should also include –DSTN_USER (abbreviation is –DS) for the user ID FRED:

```
FTR <ASH>TREE <ELM>BRANCH –FRED
```

Unknown option. (F$UNOP)

You specified an unknown option in the command line. For example, the following command specifies an extra option, –EXTRA, that is not known to FTS:

```
FTR <ELM>TEST <ASH>ANSWERS –DSTN_USER JONES –SRC_NTFY –EXTRA
```

C

Clearing Causes and Diagnostic Codes

The following table lists predefined clearing causes (CC$xxx) and diagnostic codes (CD$xxx) for the second word of the array. The numeric values associated with the codes are provided. For your convenience, the values are shown in hexadecimal, octal, and decimal form. Different communications applications (NETLINK, remote login, etc.) may display these values in different forms.

The clearing causes shown match the masked (not shifted) high-order byte of the second word of the virtual circuit status. A FORTRAN test would be

```
IF (AND(VCSTAT(2),:177400) .EQ. CC$CLR) ...
```

Clearing Cause	*Value in Hexadecimal*	*Value in Octal*	*Value in Decimal*	*Meaning*
CC$BAD	'0300'	1400	768	The call request packet is invalid.
CC$BAR	'0B00'	5400	2816	Access to the requested system has been barred.
CC$BSY	'0100'	400	256	The called system is not accepting connections right now.
CC$CLR	'0000'	0	0	This circuit was explicitly cleared. There might be a diagnostic code from you or PRIMENET.
CC$DWN	'0900'	4400	2304	The system to which this circuit is connected is not currently operating.
CC$GCN	'8500'	205	133	Private network congestion.
CC$GPE	'9300'	223	147	Private network procedural error.

Clearing Cause	*Value in Hexadecimal*	*Value in Octal*	*Value in Decimal*	*Meaning*
CC$GRN	'9900'	231	153	Private network reverse charging.
CC$IAD	'0300'	1400	768	Illegal or unknown address.
CC$LPE	'1300'	11400	4864	Local procedure error. (See CC$RPE.)
CC$NET	'0500'	2400	1280	Temporary packet network congestion.
CC$NOB	'0D00'	6400	3328	The requested system is not obtainable through the packet network.
CC$RPE	'1100'	10400	4352	Remote procedure error. Violation of X.25 protocol through a packet network.
CC$RRC	'1900'	14400	6400	The requested system refuses a collect call.

Diagnostic Code	*Value in Hexadecimal*	*Value in Octal*	*Value in Decimal*	*Meaning*
CD$BSY	'00FD'	375	253	The target system cannot accept any more connections at this time.
CD$CSP	'0040'	100	64	Call setup problem.
CD$DTE	'00A0'	240	160	DTE-specific signals.
CD$DRC	'00A3'	243	163	DTE resource constraint.
CD$DWN	'00FA'	372	250	The system to which this circuit is directed is not currently operating.
CD$FCN	'0041'	101	65	Facility code not allowed.
CD$FPN	'0042'	102	66	Facility parameter not allowed.
CD$GFI	'0028'	050	40	Invalid GFI.
CD$IAD	'00FB'	373	251	A connection request specified an unknown or illegal address.

Diagnostic Code	*Value in Hexadecimal*	*Value in Octal*	*Value in Decimal*	*Meaning*
CD$ICA	'0044'	104	68	Invalid calling address.
CD$IDA	'0043'	103	67	Invalid called address.
CD$IFL	'0045'	105	69	Invalid facility length.
CD$IPR	'0002'	002	2	Invalid P(R). Acknowledgement for packet not in window.
CD$IPS	'0001'	001	1	Invalid P(S). Lost or duplicate packet, or window size mismatch between PSDN and Prime.
CD$IP1	'0014'	024	20	Packet type invalid for state P1.
CD$IP2	'0015'	025	21	Packet type invalid for state P2.
CD$IP3	'0016'	026	22	Packet type invalid for state P3.
CD$IP4	'0017'	027	23	Packet type invalid for state P4.
CD$IP5	'0018'	030	24	Packet type invalid for state P5.
CD$IP7	'001A'	032	26	Packet type invalid for state P7.
CD$IR1	'0011'	021	17	Packet type invalid for state R1.
CD$LOP	'00F6'	366	246	A Route-through call request is looping.
CD$LPE	'00F9'	371	249	Local procedure error. A violation of the X.25 protocol
CD$MEM	'00F4'	364	244	The Route-through Server does not have enough memory for a call to be routed.
CD$MFT	'00EF'	357	239	The segment in which a parameter was stored has been deallocated.
CD$NAI	'0000'	0	0	No additional information.

Diagnostic Code	*Value in Hexadecimal*	*Value in Octal*	*Value in Decimal*	*Meaning*
CD$NRU	'00FC'	374	252	The target system has no more remote processes available at this time. (Used with remote login.)
CD$NSV	'00F8'	370	248	The PRIMENET server process is not running.
CD$PIC	'002A'	052	42	Packet type not compatible with facility.
CD$PNA	'00FF'	377	255	The port to which this call is directed is not assigned in the target system.
CD$PNL	'0020'	040	32	Packet not allowed.
CD$PTI	'0010'	020	16	Packet type invalid.
CD$PTL	'0027'	047	39	Packet too long.
CD$PTS	'0026'	046	38	Packet too short.
CD$RST	'0029'	051	41	Restart with nonzero Logical Channel Number.
CD$RTD	'00F3'	363	243	The Route-through Server is down or inconsistencies exist between different network configuration files.
CD$RTE	'00F7'	367	247	A Route-through protocol error was detected.
CD$SNU	'00FE'	376	254	The target system is not yet available for login. (Used with remote login.)
CD$TCA	'00F2'	362	242	A Call Request has not been answered by the target system, so the circuit was cleared.
CD$TCE	'0090'	220	144	Timer expired or retransmit count surpassed.
CD$TCI	'0032'	062	50	Timer expired for clear indication. Sent to remote system in retransmitted clear request.

Diagnostic Code	*Value in Hexadecimal*	*Value in Octal*	*Value in Decimal*	*Meaning*
CD$TCR	'00F0'	360	240	A clear request has not been confirmed by the target system, so the circuit has been cleared and dropped.
CD$TIC	'0031'	061	49	Call not accepted within 100 seconds.
CD$TIN	'0091'	221	145	Timer expired for interrupt confirmation. Sent to remote system in reset request.
CD$TME	'0030'	060	48	Timer expired. No additional information.
CD$TMO	'00F5'	365	245	The Route-through Server experienced a virtual circuit timeout.
CD$TRE	'0033'	063	51	Timer expired for reset confirmation.
CD$TRT	'0034'	063	52	Timer expired for restart indication.
CD$UIC	'002B'	053	43	Unauthorized interrupt confirmation.
CD$UIN	'002C'	054	44	Unauthorized interrupt.
CD$ULC	'0024'	044	36	Packet on unassigned logical channel.
CD$UNP	'0021'	044	36	Unidentified packet.
CD$URC	'001B'	033	27	Unauthorized reset confirmation.

Prime Network Programming Glossary

This appendix contains a glossary of terms that may be of use to use to you as a programmer writing distributed applications. Terms that are in italics in definitions are defined elsewhere in the glossary.

asynchronous communication

A method of transmitting data in which each character is preceded by a start *bit* and followed by a stop bit. The time interval between the characters may vary.

asynchronous line

A line that carries *asynchronous communication.*

bit

An acronym for binary digit. Eight bits constitute a *byte.*

byte

Eight *bits* of data. A character, for example, is one byte.

CCITT

See Consultative Committee for International Telephony and Telegraphy.

communication line

See communication link.

communication link

The connecting medium between two systems that allows them to transmit and/or receive data. At Rev. 21.0, types of communication links include *LAN300*, *RINGNET*, *synchronous lines* (*full-duplex* and *half-duplex*), and *Packet Switched Data Networks* (PSDNs). Two systems can also be linked by one or more intervening *gateway nodes.*

CONFIG_NET

PRIMENET's global *network configuration* utility (or configurator). Also, the command used to invoke the configurator.

Consultative Committee for International Telephony and Telegraphy (CCITT).

An advisory committee established under the auspices of the United Nations to recommend worldwide standards.

event

A significant system or *network* occurrence such as a cold start, machine check, disk error, or network link problem.

FDX

See full-duplex.

File Transfer Generation (FTGEN)

The *File Transfer Service* (FTS) utility that a *System Administrator* or *Network Administrator* uses to configure the FTS database. FTGEN is described in the *PRIMENET Planning and Configuration Guide.*

File Transfer Manager

See YTSMAN.

File Transfer Operator (FTOP)

The *File Transfer Service* (FTS) utility that an operator uses to manage the FTS system. FTOP is described in the *Operator's Guide to Prime Networks.*

File Transfer Request (FTR)

The *File Transfer Service* (FTS) utility for submitting transfer requests.

File Transfer Service (FTS)

A queued file transfer program that enables files to be transferred between Prime systems in a *network.* FTS comprises the following utilities: *File Transfer Request* (FTR), *File Transfer Generation* (FTGEN), and *File Transfer Operator* (FTOP).

framing

The process of prefixing and suffixing a message with control characters before sending it over a *communication link.* The control characters are said to frame the message.

FTOP

See File Transfer Operator.

FTR

See File Transfer Request.

FTS

See File Transfer Service.

full-duplex

A communication mode in which both systems can send and receive signals at the same time. In a *PRIMENET* environment, full-duplex communication occurs over a permanently configured, dedicated connection (cables or leased telephone lines). Full-duplex lines can link a Prime system to a PSDN or to another Prime system.

gateway

See gateway node.

gateway access

The *access rights* between two systems that communicate through a *gateway node.* For example, consider the configuration shown in Figure GL-1. Systems A and C communicate through gateway node B. The gateway access from node A to node C is IPCF; the gateway access from node C to node A is RLOG. Gateway access is defined during *network* configuration.

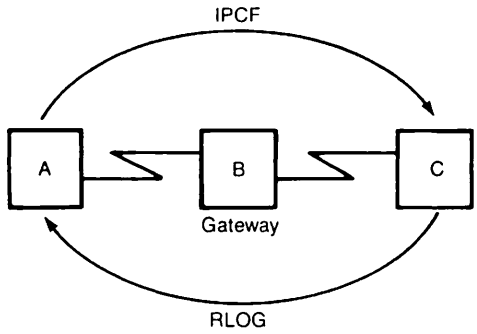

FIGURE GL-1
Gateway Access

gateway node

A system on which the *Route-through Server* is configured. A gateway node can route messages between two other systems. Two systems may communicate via a gateway node even if they are not directly connected.

half-duplex

A communication mode in which data transmission occurs in only one direction at a time. Each system alternates between sending and receiving signals. In a *PRIMENET* environment, half-duplex communication occurs over temporary connections (generally, dialup telephone lines). Half-duplex PRIMENET lines may link two Prime systems, but may not link a Prime system to a non-Prime system or to a *Packet Switched Data Network* (PSDN).

HDX

See half-duplex.

HDX network/HDX subnetwork

The set of all *half-duplex* (HDX) *nodes* and lines in a *network.*

HDX node

A node that has a *half-duplex* line attached to it.

header

The beginning portion of a *packet* that contains information such as the IDs of the target and sending *node* or packet type.

Interprocess Communications Facility (IPCF)

A set of subroutines that permit applications to send and receive messages within a Prime *network,* or to transfer messages between processes on the same system. Also, an *access right* (selected through *CONFIG_NET*) that enables systems to communicate using IPCF subroutines only. IPCF access is the minimum access that you can configure between systems.

International Organization for Standardization (ISO)

Organization responsible for developing Open Systems Interconnection (OSI) model, a 7-layered network architecture.

IPCF

See Interprocess Communications Facility.

ISO

See International Organization for Standardization.

LAN

See Local Area Network.

LAN300

The Prime IEEE 802.3 compliant *Local Area Network* that uses a bus topology. A standard LAN300 is composed of bus segments, each of which can be a maximum of 500 meters in length. Devices are connected to a LAN300 at a *station*, where a controller, *transceiver*, and tap are attached to the bus segment. A station supports a host or a cluster of terminal or serial printers or both. The LAN300 uses the *Carrier Sense Multiple Access with Collision Detection (CSMA/CD) access method.*

LCN

The *X.25* Logical Channel Number of a virtual circuit. When combined with the *LCGN*, forms the X.25 Virtual Circuit Number.

LCGN

The *X.25* Logical Channel Group Number of a virtual circuit. When combined with the *LCN*, forms the X.25 Virtual Circuit Number.

Level 1

PRIMENET's hardware interface. This level (or layer) acts as an intermediary between the physical transmission medium (cable or transmission line) and *Level 2*.

Level 2

PRIMENET's link level. It describes a *protocol* for transferring data between two directly connected systems.

Level 3

PRIMENET's packet level. It creates and controls connections across the *network*, handles error recovery, and controls the flow of data between processes on a pair of communicating systems.

Local Area Network (LAN)

A *network* in which independent computer systems are physically connected and communicate at a high speed over a short distance, such as within a building or building complex. *RINGNET* is the Prime Local Area Network that uses a ring configuration. *LAN300* is the Prime IEEE 802.3 compliant LAN that uses a bus configuration.

local system

A system on a *network* from which a user issues commands to communicate with another *remote* system.

NETLINK

A *PRIMENET* utility that enables a user to gain access to another networked Prime system or a non-Prime system across a *Packet Switched Data Network* (PSDN) if that non-Prime system adheres to the *CCITT PAD protocols (X.3/X.28/X.29).*

NETMAN

See Network Manager Process.

network

A group of independent computer systems that are connected by communication media such as PSDNs, LANs, and *synchronous lines* and that communicate and share resources. A network can consist of systems that are all physically connected and communicate over short distances, as in a LAN, or of systems that use different communication media to communicate over long distances.

The term network is sometimes used synonymously with *subnetwork*; for example, *ring network* or *HDX network* can refer to subsets of a larger network.

Network Administrator

The person responsible for maintaining the proper and continuous operation of a *network.* Tasks include using *CONFIG_NET* to configure the network, ensuring that appropriate security measures are taken, and maintaining the daily network operation. Sometimes the same person serves as Network Administrator and *System Administrator. See also* System Administrator.

network configuration

A description of the systems, services, and communication media that make up a *network.*

network configuration file

The file that contains the network configuration in binary format. The *Network Administrator* creates this file through *CONFIG_NET.* At network startup time, this file is loaded into *PRIMENET* by the *START_NET* command (unless a *cache file* is used).

Network Manager Process (NETMAN)

A process that handles network activity. NETMAN is X.25 Level 2 and Level 3 PRIMENET. NETMAN appears on the STATUS USERS list as nsp (network server process).

Network Process Extension (NPX) facility

An internal PRIMOS facility that provides a remote procedure call mechanism between any two *PRIMENET* systems. *Remote File Access* (RFA) uses NPX.

network protocol

See protocol.

node

An independent computer system that is part of a *PRIMENET network.*

NPX

See Network Process Extension Facility.

packet

A sequence of data and control characters that are arranged in a specific format and transmitted as a unit.

Packet Assembler/Disassembler (PAD)

Provides a number of functions: controlling normal terminal operation, controlling normal *X.25* circuit functions, passing characters from terminal to host over a *virtual circuit*, passing characters to terminal as they are received from host over virtual circuits, handling call clearing, and providing other functions using the *X.3* recommendation. *NETLINK* emulates a PAD.

Packet Switched Data Network (PSDN)

A *network* in which the *X.25* protocol defines communication between X.25-compatible equipment called Data Terminal Equipment (DTE) and processors called Data Circuit Termination Equipment (DCE). To transmit data, PSDNs divide long messages into shorter units with a fixed maximum length (*packets*). Examples of PSDNs include TELENET, UNINET, TYMNET, PSS or an equivalent private network.

packet size

The number of *bytes* in a *packet.*

PAD

See Packet Assembler/Disassembler.

path

The sequence of intervening systems between two given systems in a *network.*

port

An address within a *node* to which an incoming *network* request can be routed. Each node in a *PRIMENET* network has a pool of available ports that a program running under PRIMOS can assign.

PRIMENET

Prime's distributed networking software that offers local and wide-area networking facilities.

PRIMENET address

A numeric address that *PRIMENET* uses internally to identify a *node. CONFIG_NET* generates this address based on the name of the node.

protocol

A set of rules governing communication between two systems in a *network.*

PSDN

See Packet Switched Data Network.

PSDN address

A unique sequence of as many as 15 digits assigned by a PSDN to any *node* connected directly or indirectly to the PSDN. An indirect connection is indicated when a 2-digit subaddress is used.

PSDN gateway

A *communication link* between two different PSDNs. Can be referred to as an *X.25 gateway.*

Remote File Access

See RFA.

remote login

See RLOG.

remote system

A system that can communicate with the *local system* through a *network.*

remote user

A user on a *remote system.*

RFA

Remote File Access. A *PRIMENET* intersystem service or *access right* (selected through *CONFIG_NET*) that enables a user to access files on a *remote system* as though the files were on the *local system.*

ring

See ring network.

ring network

A type of *Local Area Network* (LAN). Prime's ring LAN, *RINGNET,* is a token-passing ring network.

RINGNET

One of Prime's LANs for Prime-to-Prime communications. A RINGNET *network* is composed of Prime systems that are connected by cable in a ring configuration. Each system is logically connected to all other systems on the ring. RINGNET uses a token-passing *protocol* to control communication around the ring.

ring node ID

A number from 1 through 247 that identifies, and is unique to, a particular *node* on a *RINGNET network.*

RLOG

Remote login. A *PRIMENET* service that enables users to log in to a *remote system* from a local terminal without logging in to the *local system* first. The local and remote systems must be connected directly or connected through one or more gateway *nodes*. The *Network Administrator* must assign remote login access rights (selected through *CONFIG_NET*) during network configuration. These access rights are checked only on the remote system.

Route-through

The message-routing operation performed on a *gateway node* that connects two systems or *networks.*

Route-through Server (RT_SERVER)

The *server* that performs *Route-through* and enables a system to act as a *gateway node* for communication between *nodes* not directly connected through a ring, PDSN, or *synchronous line.*

server

A cooperating set of processes available to perform one or more functions. FTS servers service local request queues and incoming requests from *remote systems.*

slave process

A process on a *local system* that handles a request that a user on another system makes to access files or to attach to a directory on the local system. A slave acts for a single *remote user* until the remote user releases the slave process. The number of slave processes available (configured by the *System Administrator*) depends on whether the local system is using the *Route-through Server* or the *File Transfer Service* (FTS) or both.

synchronous communication

Transmission in which data, characters, and *bits* are transmitted at a fixed rate. The transmitting and receiving systems are synchronized, thus eliminating the need for start and stop bits.

Generally, synchronous communication offers more efficient line usage, better error checking, and higher speeds of transmission than *asynchronous communication.*

synchronous line

A line that carries *synchronous communication.*

System Administrator

The person responsible for maintaining the proper and continuous operation of a system. The System Administrator's duties can include network-related tasks such as setting *PRIMENET*-related ACL rights, and setting up the *File Transfer Service* (FTS). At times, the same person serves as System Administrator and *Network Administrator. See also* Network Administrator.

timeout

The condition that occurs when a transmitting *node* sees neither a token nor a *packet* within a certain time period.

virtual circuit

A logical *network* connection that enables transmission of data between two processes. IPCF subroutines are used to establish virtual circuits in *PRIMENET.* PRIMENET supports 255 virtual circuits.

window size

The maximum number of *frames* or *packets* that can be sent before an acknowledgment must be received. Window size is configurable only for *Packet Switched Data Network* (PSDN) links.

X.3

A *CCITT* recommendation entitled "Packet Assembly/Disassembly Facility in a Public Data Network". X.3 outlines the procedures for *packet* assembly/disassembly for asynchronous transmissions.

X.25

A *CCITT* recommendation entitled "Interface between Data Terminal Equipment (DTE) and Data Circuit Terminating Equipment (DCE) for Terminals Operating in the Packet Mode and Connected to Public Data Networks by Dedicated Circuit". The X.25 recommendation and the X.25 *protocol,* based on the recommendation, define communication between X.25-compatible equipment called Data Terminal Equipment (DTE) and processors called Data Circuit Termination Equipment (DCE) in a PSDN.

X.28

A *CCITT* recommendation entitled "DTE/DCE Interface for a Start-Stop Mode Data Terminal Equipment Accessing the Packet Assembly/Disassembly Facility (PAD) in a Public Data Network Situated in the Same Country". The X.28 recommendation describes the interfacing procedures that enable an asynchronous terminal to be connected to a PAD.

X.29

A *CCITT* recommendation entitled "Procedures for the Exchange of Control Information and User Data Between Assembly/Disassembly Facility (PAD) and a Packet Mode DTE or Another PAD". The X.29 recommendation describes the interfacing procedures that enable a PAD to communicate with an *X.25 network.*

X.121

A *CCITT* recommendation entitled "International Numbering Plan for Public Data Networks". The X.121 recommendation describes an addressing scheme, supported by *NETLINK,* to uniquely identify computer systems within PSDNs.

Index

Index

O

P

R

S

T

U

V

W

X

Surveys

READER RESPONSE FORM

Programmer's Guide to Prime Networks
DOC10113-1LA

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate this document for overall usefulness?

   ☐ *excellent* ☐ *very good* ☐ *good* ☐ *fair* ☐ *poor*

2. What features of this manual did you find most useful?

3. What faults or errors in this manual gave you problems?

4. How does this manual compare to equivalent manuals produced by other computer companies?

   ☐ *Much better* ☐ *Slightly better* ☐ *About the same*
   ☐ *Much worse* ☐ *Slightly worse* ☐ *Can't judge*

5. Which other companies' manuals have you read?

Name:
Position:
Company:
Address:

Postal Code:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

Prime™

Attention: Technical Publications
Bldg 10
Prime Park, Natick, Ma. 01760