BEA MessageQ

LU6.2 Services for OpenVMS User's Guide

BEA MessageQ LU6.2 Services for OpenVMS V4.0A
Document Edition 4.0A
February 1999

BEA MessageQ LU6.2 Services for OpenVMS User’s Guide

Document Edition	Date	Software Version
4.0A	February 1999	BEA MessageQ LU6.2 Services for OpenVMS, Version 4.0A

Contents

Preface

Purpose of This Document

This document describes the BEA MessageQ LU6.2 Services for OpenVMS product. It also provides instructions for developing applications using this software and for configuring the LU6.2 Port Server.

Who Should Read This Document

This document is intended for system administrators, network administrators, and developers who are interested in enabling communications between BEA MessageQ and IBM applications.

How This Document Is Organized

The *BEA MessageQ LU6.2 Services for OpenVMS User's Guide* is organized as follows:

- Chapter 1, "Introducing BEA MessageQ LU6.2 Services," provides an overview of the BEA MessageQ LU6.2 Services for OpenVMS, including basic concepts and terms.
- Chapter 2, "Developing Applications Using BEA MessageQ LU6.2 Services," provides an overview of how to use BEA MessageQ LU6.2 Services for OpenVMS to develop programs that communicate between IBM mainframes and VAX or Alpha systems running OpenVMS.

- Chapter 3, “Configuring the LU6.2 Port Server,” describes how to configure, start up, and manage the LU6.2 Port Server.
- Chapter 4, “Port Server Messages,” describes the messages used by the LU6.2 Port Server: port server control messages and port server connection messages (BEA MessageQ messages).
- Chapter 5, “LU6.2 Port Server Application Programming Interface,” presents a sample application programming interface (API) for LU6.2 Services for OpenVMS.
- Chapter 6, “LU6.2 User Callback Services,” introduces the LU6.2 User Callback Services (UCB) and contains detailed descriptions of all LU6.2 User Callback APPC messages alphabetized by message type.
- Appendix A, “LU6.2 User Callback Interface Logical Names and Error Codes,” describes the LU6.2 user callback logical names and error codes.
- Appendix B, “Notes on IMS,” provides information on the restrictions present when using APPC verbs with the IMS LU6.1 Adapter.
- Appendix C, “Examples of BEA MessageQ LU6.2 Inbound and Outbound Applications,” provides sample Inbound and Outbound applications that exchange data with an APPC application in an SNA network.
- Appendix D, “Examples of CICS Inbound and Outbound Applications,” provides sample CICS Inbound and Outbound applications.

How to Use This Document

This document, *BEA MessageQ LU6.2 Services for OpenVMS User’s Guide*, is designed primarily as an online, hypertext document. If you are reading this on paper, note that to get full use from this document you should install and access it as an online document via a Web browser.

The following sections explain how to view this document online, and how to print a copy of this document.

Opening the Document in a Web Browser

To access the online version of this document, open the following HTML file in a Web browser:

`/beadir/doc/bmq/lu62_40a/usergde/index.htm`

Note: The online documentation requires a Web browser that supports HTML version 3.0. We recommend Netscape Navigator version 4.0 or Microsoft Internet Explorer version 4.0 or later.

Figure 1 shows the online document with the clickable navigation bar and table of contents.

Figure 1 Online Document Displayed in a Netscape Web Browser

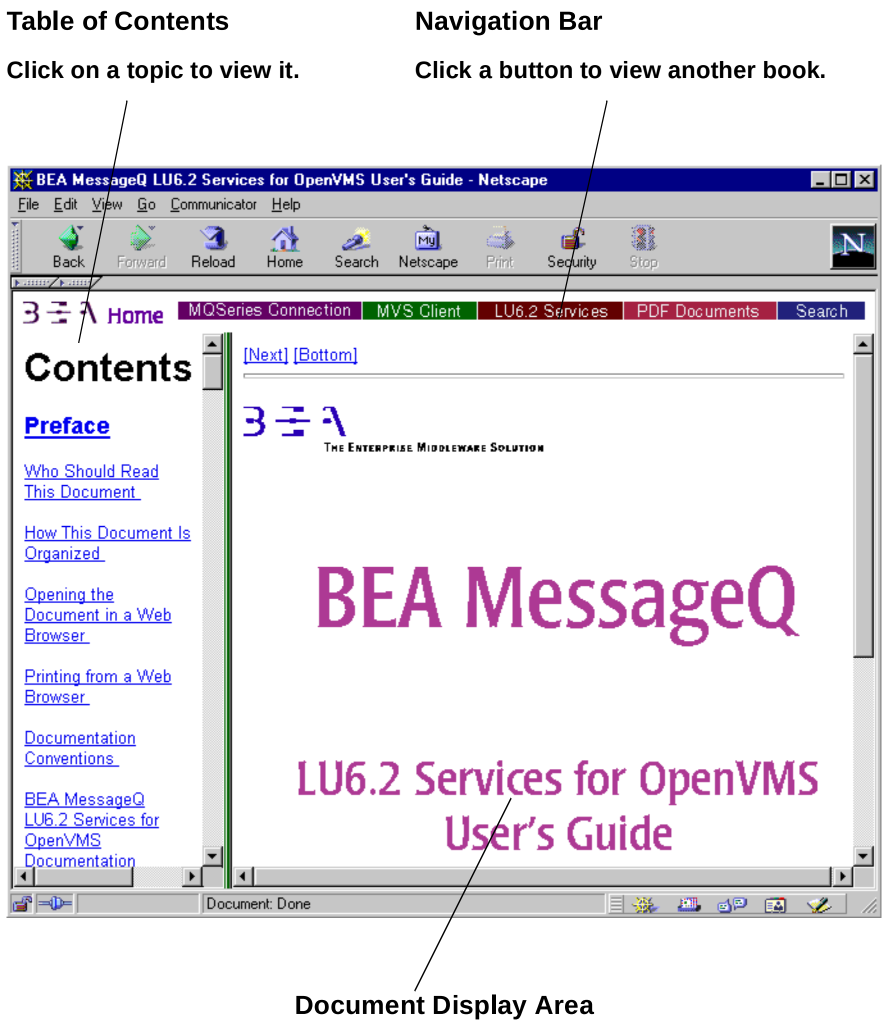


Table of Contents
Click on a topic to view it.
Navigation Bar
Click a button to view another book.
BEA MessageQ LU6.2 Services for OpenVMS User's Guide - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Home
MQSeries Connection
MVS Client
LU6.2 Services
PDF Documents
Search
Contents
Preface
Who Should Read This Document
How This Document Is Organized
Opening the Document in a Web Browser
Printing from a Web Browser
Documentation Conventions
BEA MessageQ LU6.2 Services for OpenVMS Documentation
[Next] [Bottom]
The Enterprise Middleware Solution
BEA MessageQ
LU6.2 Services for OpenVMS User's Guide
Document: Done
Document Display Area

Printing from a Web Browser

You can print a copy of this document, one file at a time, from the Web browser. Before you print, make sure that the chapter or appendix you want is displayed and *selected* in your browser. (To select a chapter or appendix, click anywhere inside the chapter or appendix you want to print. If your browser offers a Print Preview feature, you can use the feature to verify which chapter or appendix you are about to print.)

The BEA MessageQ Online Documentation CD also includes Adobe Acrobat PDF files of all of the online documents. You can use the Adobe Acrobat Reader to print all or a portion of each document.

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary in the *BEA MessageQ Introduction to Message Queuing*.
Ctrl+Tab	Indicates that you must press two or more keys sequentially.
italics	Indicate emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include stdio`
	`pams_attach_q`
	`\bmq\lu62_40a\include`
	`.htm`
	`bmq.doc`
	`BITMAP`
	`float`

Convention	Item
`monospace boldface text`	Identifies significant words in code. *Example*: `put_msg(msg_ptr,` **`class`**`, type)`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 PATH OR
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.

`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed.
	Example:
	`int32 pams_get_msg (`*`msg_area`*`,` *`priority`* `...`
	`[-`*`sel_filter`*`] [psb] [`*`show_buffer`*`]...)`

`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

`...`	Indicates one of the following in a command line:
	♦ That an argument can be repeated several times in a command line
	♦ That the statement omits additional optional arguments
	♦ That you can enter additional parameters, values, or other information
	The ellipsis itself should never be typed.
	Example:
	`int32 pams_get_msg (`*`msg_area`*`,` *`priority`* `...`
	`[-`*`sel_filter`*`] [psb] [`*`show_buffer`*`]...)`

. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

Related Documentation

The following sections list the documentation provided with the BEA MessageQ software, and other publications related to messaging-oriented middleware technology.

BEA MessageQ LU6.2 Services for OpenVMS Documentation

The BEA MessageQ LU6.2 Services for OpenVMS information set consists of the following documents:

BEA MessageQ LU6.2 Services for OpenVMS User's Guide

BEA MessageQ LU6.2 Services for OpenVMS Installation Guide

BEA MessageQ LU6.2 Services for OpenVMS Release Notes

Note: The BEA MessageQ Online Documentation CD also includes Adobe Acrobat PDF files of all of the online documents. You can use the Adobe Acrobat Reader to print all or a portion of each document.

BEA Publications

The following BEA publications are also available:

BEA MessageQ Introduction to Message Queuing

BEA MessageQ Programmer's Guide

BEA MessageQ Installation and Configuration for OpenVMS

BEA MessageQ Client for OpenVMS User's Guide

BEA MessageQ for OpenVMS Release Notes for Version 4.0A

Contact Information

The following sections provide information about how to obtain support for the documentation and software.

Documentation Support

If you have questions or comments on the documentation, you can contact the BEA Information Engineering Group by e-mail at **docsupport@beasys.com**. (For information about how to contact Customer Support, refer to the following section.)

Customer Support

If you have any questions about this version of BEA MessageQ LU6.2 Services for OpenVMS, or if you have problems installing and running BEA MessageQ LU6.2 Services for OpenVMS, contact BEA Customer Support through BEA WebSupport at `www.beasys.com`. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

CHAPTER

1 Introducing BEA MessageQ LU6.2 Services

The BEA MessageQ LU6.2 Services for OpenVMS product allows users to communicate with IBM application programs using Advanced Program-to-Program Communications (APPC) over System Network Architecture (SNA) LU6.2 sessions. APPC/LU6.2 communications are *connection-oriented* and *half-duplex*. This means that before two partners can exchange messages, they must first establish a connection (connection-oriented), and that one partner is sending when the other is receiving (half-duplex).

This chapter includes the following topics:

- ♦ Basic Terms and Concepts
- ♦ SNA APPC/LU6.2 Fundamentals

Basic Terms and Concepts

This section provides general information about BEA MessageQ applications, APPC, the Port Server, and other application components.

The BEA MessageQ LU6.2 Services Product

The BEA MessageQ system is a connectionless, stateless communications system. This means that a connection need not be established before programs can communicate with each other. It also means that communicating programs do not care what state their partner is in.

Using BEA MessageQ LU6.2 Services is like using the postal mail system—you can send a letter to anyone whose address you know, whether or not the addressee is home and regardless of the addressee's desire to speak to you. Addressees can check their mailboxes at their convenience.

The BEA MessageQ system supports network-independent addressing and makes use of an application programming interface (API) common to all platforms that support the MessageQ system.

SNA LU6.2 Sessions

SNA is connection-oriented systems networking architecture. Connections are established between network-addressable units. Applications are interested in *logical units* (LUs) that are the “end users” of a network. The connection between LUs is an SNA session.

The LU that requests a session is called the *primary logical unit* or PLU. The LU that accepts the session is called the *secondary logical unit* or SLU. The capabilities of an LU are defined by the LU type. Only one type of LU supports APPC: LU type 6.2 (also known as LU6.2).

An *LU6.2 session* is the type of SNA connection used by APPC applications for communications with each other. The session is like the telephone connection received when you dial another number—if the other number does not answer, you cannot speak.

Without an SNA session, APPC conversations cannot take place. An LU is the software equivalent of a telephone—it has a specific address in the network, and other LUs have to know that address in order to establish sessions with it.

Advanced Program-to-Program Communications (APPC)

A program that wants to communicate with other programs must first establish a connection to each potential partner, and the communicating programs must be aware of each other's state (for example, ready to receive, ready to send, and shutting down). This requirement is similar to the requirements of the telephone system: before you can talk to someone, you must call someone up; the call recipient must answer the phone, and agree to listen to you; after you have finished speaking, the other party can speak to you; and so on.

BEA MessageQ LU6.2 Services for OpenVMS applications use Advanced Program-to-Program Communications (APPC) to establish connections to partners and communicate state information before initiating conversations. APPC is a connection-oriented, half-duplex, state-oriented communications system. It uses the following concepts:

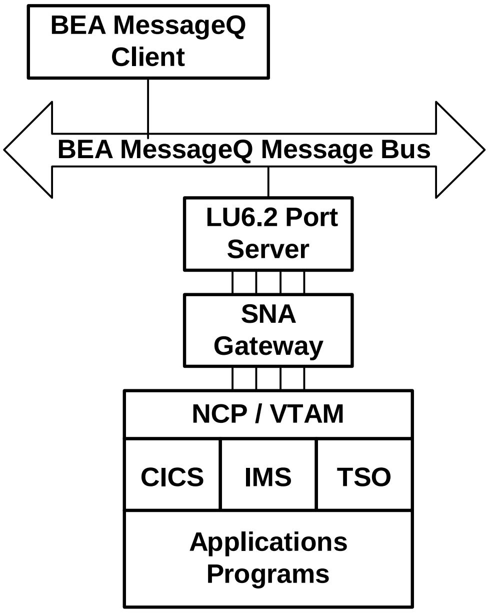
- ♦ Conversation—a structured exchange of messages between two partners, conducted over a previously established LU6.2 session.
- ♦ Allocation—the act of establishing a conversation.
- ♦ Deallocation—the act of terminating an existing conversation.
- ♦ Sync-level—the highest degree of synchronization permitted on a given conversation. Sync-level parameters are 0 (None), 1 (Confirm), and 2 (Syncpoint). The BEA MessageQ LU6.2 Services for OpenVMS product uses only sync-levels 0 and 1.

  APPC provides a set of functions called *APPC verbs* to manage conversations. (See Chapter 6 for more information on APPC verbs.)

BEA MessageQ LU6.2 Services Application Components

Figure 1-1 shows the components of a typical BEA MessageQ LU6.2 Services application.

Figure 1-1 BEA MessageQ LU6.2 Components



The components are:

- The *BEA MessageQ client* that communicates with the BEA MessageQ LU6.2 Port Server by sending and receiving messages over the BEA MessageQ message bus. By sending specific, predefined message types, the BEA MessageQ client can ask the Port Server to:
  - Establish a connection to an IBM LU6.2 application (a Connect request)
  - Send traffic received from incoming connections to a specific BEA MessageQ client (a Register Target request)
  - Send or receive data over a previously established connection (a data message)
  - Manage a previously established connection (a Change Direction request or a Connection Terminated notice)
- The *BEA MessageQ LU6.2 Port Server* that establishes and maintains SNA sessions for use by BEA MessageQ clients and applications residing on remote IBM systems.

The LU6.2 Port Server is a *connection point manager:* a software process that understands the "language" of network applications located in different networks on opposite sides of a "connection point," the place where the two networks attach to each other.

A Port Server manages communication resources on behalf of client processes (resource sharing), reduces application complexity by "hiding" the details of resource management, and optionally provides other value-added services (like multithreading).

The SNA sessions are created using the services of the SNA Gateway (either a DECnet/SNA Gateway, a Domains Gateway, or a Peer Server). SNA sessions are assigned to BEA MessageQ clients on a first-come, first-served basis when the clients send Connect requests.

- ♦ The *SNA Gateway* that performs the lower-level protocol and message format translations required to connect a DECnet network with an SNA network.
- ♦ *NCP* and *VTAM*, software components crucial to the operation of the networked SNA systems. SNA Gateways, communications lines, and LUs must be properly defined to NCP and VTAM before communication with the IBM applications is possible.
- ♦ *CICS*, *TSO,* and *IMS*, application subsystems that run on the MVS operating system and support the use of APPC over LU6.2 sessions.

  BEA MessageQ LU6.2 Services are most frequently used to communicate with application programs that run under these application subsystems.

SNA APPC/LU6.2 Fundamentals

The information that follows is a short summary of SNA APPC LU6.2 fundamentals. For more detailed information, refer to *SNA Transaction Programmer's Reference Manual for LU Type 6.2*.

Logical Unit Type 6.2 Overview

LU6.2 is a general-purpose architecture that enables IBM products to communicate with each other. The LU6.2 architecture defines a set of protocols. To communicate with each other, products must implement LU6.2 according to these protocols. There are two general implementations of LU6.2:

- The “open-box” protocol provides a programming interface to allow customized solutions.
- The “closed-box” protocol provides no programming interface but does offer turnkey solutions.

Transaction programs (TPs) are programs that can process a specific set of input data, trigger specific job executions, or produce specific output data. Distributed transactions within an SNA network communicate by exchanging information during a conversation, which is a temporary logical path established between two cooperating TPs. This path is treated as a shared resource between TPs.

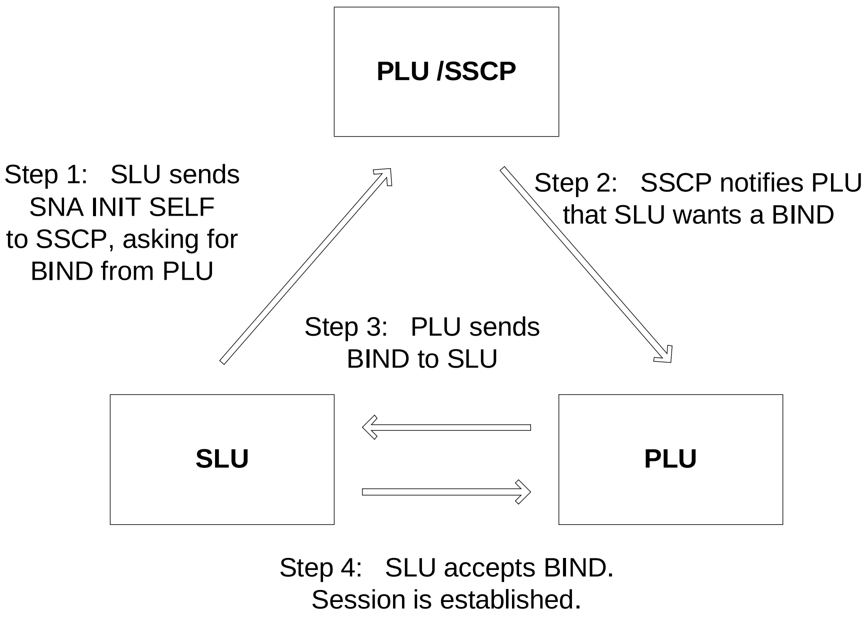
IBM's architectural definition for LU6.2 provides a set of procedures called *verbs* that is used to design distributed transactions. CICS ISC implements the verb functions with `EXEC CICS` commands and OpenVMS TPs implement the verb functions with OpenVMS procedures.

The APPC verb set consists of function verbs that are implemented as BEA MessageQ messages. The logic of how these verbs are used (verb flow) is the same regardless of whether you are programming in an OpenVMS environment or an IBM environment. Refer to Chapter 6, “LU6.2 User Callback Services,” for detailed information on *User Callback Services* and APPC verbs.

Inbound and Outbound Conversations

Conversation allocations may be initiated either by the OpenVMS TP (inbound) or by the IBM TP (outbound). Figure 1-2 illustrates a typical SNA conversation session. The LU6.2 type of SNA LUs can be configured as independent or dependent. The BEA MessageQ LU6.2 Services for OpenVMS product uses dependent LUs.

Figure 1-2 SNA Session for Dependent LUs


PLU /SSCP
Step 1: SLU sends SNA INIT SELF to SSCP, asking for BIND from PLU
Step 2: SSCP notifies PLU that SLU wants a BIND
Step 3: PLU sends BIND to SLU
SLU
PLU
Step 4: SLU accepts BIND. Session is established.

Inbound allocation causes the gateway to transmit an `INIT SELF` to the IBM SSCP that builds a suggested `BIND` and passes it to the application subsystem (which may modify it). The application passes the `BIND` back to the gateway. If the gateway accepts the `BIND`, the session is established and the OpenVMS and IBM TPs are in a "session."

Outbound allocation causes the gateway to wait in the active-listening mode for the IBM TP to send a `BIND` followed by an `ATTACH`. If the gateway accepts this `ATTACH`, the session is established and the OpenVMS and IBM TPs are in a session. They are also in a state of conversation (also called "between brackets").

Contention for session resources occurs when both partners attempt to begin conversation simultaneously on the same session. The contention is resolved according to the polarity agreed upon when the session was established. The contention winner (first speaker) always receives the session resources, and the contention loser (bidder) always has to wait. The contention winner/loser is negotiated in the `BIND`.

Using the LU6.2 Port Server for Applications Connections

Exchanging information between the connectionless, stateless environment of the BEA MessageQ system and the connection-oriented, state-oriented environment of APPC (a task similar to that of connecting systems as different as the postal mail and the telephone) requires the services of an intermediary that understands both. The intermediary provided by BEA MessageQ LU6.2 Services is the LU6.2 Port Server.

The LU6.2 Port Server understands how to:

- Get connections to APPC applications when asked to do so by BEA MessageQ applications (inbound connections)
- Convert data messages from one network into the form required by the other network
- Accept connections from APPC applications and deliver the incoming data to BEA MessageQ applications (outbound connections)
- Detect errors from one network and deliver the proper error notifications to the other network

In addition, the LU6.2 Port Server performs numerous “housekeeping” tasks (such as error recovery, automatic restart of communication links, and error and trace logging) that are desirable in distributed applications.

Refer to Chapter 3, “Configuring the LU6.2 Port Server,” for more information on the LU6.2 Port Server.

The BEA MessageQ LU6.2 Services product uses the Stream Output Facility for logging and tracing. This facility provides time stamps on both the logging and tracing output as well as dynamic tracing. See the *BEA MessageQ Installation and Configuration Guide for OpenVMS* for more information on dynamic tracing and the BEA MessageQ for OpenVMS Event Logger Utility.

Writing Your Own Port Server

When installed, the BEA MessageQ LU6.2 Services software provides a typical LU6.2 Port Server that uses seven predefined messages to simplify setting and managing the application connections of the BEA MessageQ clients to remote partners.

However, if the standard LU6.2 Port Server programming interface does not meet the needs of an application, does not offer a function required by all client applications, or does not meet some other unusual requirements, a specialized Port Server may be developed using the LU6.2 User Callback Services.

The BEA MessageQ LU6.2 Services provide you with the option of writing your own Port Server using 21 BEA MessageQ messages that map to the APPC verb set.

Note: Refer to Chapter 3, “Configuring the LU6.2 Port Server,” and Chapter 4, “Port Server Messages,” for LU6.2 Port Server information. Refer to Chapter 6, “LU6.2 User Callback Services,” for LU6.2 user callback information.

2 Developing Applications Using BEA MessageQ LU6.2 Services

This chapter provides an overview of how to use BEA MessageQ LU6.2 Services for OpenVMS to develop programs that communicate between IBM mainframes and VAX or Alpha systems running OpenVMS.

This chapter describes:

- Applications Development Overview
- Structure of BEA MessageQ LU6.2 Services Applications
- Development Checklist

Applications Development Overview

BEA MessageQ LU6.2 Services can be used to develop a wide range of distributed applications that integrate BEA MessageQ applications and LU6.2 APPC clients.

BEA MessageQ LU6.2 Services support three types of applications:

- Inbound applications
- Outbound applications
- Hybrid applications

Inbound Applications

Inbound applications initiate APPC conversations with partner programs in the SNA network based on events that occur in the BEA MessageQ network (for example, user input at a terminal or workstation, receipt of a message from another BEA MessageQ client program, and so on). Inbound applications are typically used to trigger application actions in the SNA network based on events and data generated in the BEA MessageQ network.

Outbound Applications

Outbound applications accept APPC conversations initiated by partner programs in the SNA network based on events that occur in the SNA network (such as user input at a terminal or workstation, receipt of a message from other APPC client programs, and the like). Outbound applications are typically used to trigger application actions in the BEA MessageQ network based on events and data generated in the SNA network.

Hybrid Applications

Hybrid applications both initiate APPC conversations with partner programs in the SNA network and accept APPC conversations initiated by partner programs in the SNA network. Hybrid applications are typically used to route application traffic among BEA MessageQ and APPC clients based on application-specific criteria.

The LU6.2 Services port server is a very general form of a hybrid application: it both initiates inbound APPC conversations and accepts outbound conversations.

Target Registration

A BEA MessageQ client that is to receive messages on outbound sessions must be registered with the LU6.2 Port Server before a connection can be established. A BEA MessageQ client is registered by sending a message to the LU6.2 Port Server. This message contains the following information:

- The name of the target (as known to the LU6.2 Port Server) to be used by the IBM client to establish communication with the BEA MessageQ client
- The BEA MessageQ group ID and queue number of the BEA MessageQ client

After registering the target, the system returns the `REGISTER_TARGET` message to the BEA MessageQ client.

A BEA MessageQ client can register itself or it can be registered by another application. Each target may be registered by only one BEA MessageQ client application. Applications are automatically deregistered when they exit.

Note: A permanent outbound target is permanently registered with the BEA MessageQ group ID and queue number provided on the target definition. BEA MessageQ clients that receive output from permanent outbound targets do not need to register.

Structure of BEA MessageQ LU6.2 Services Applications

BEA MessageQ LU6.2 Services applications are BEA MessageQ applications that use the services of the BEA MessageQ LU6.2 Services Port Server to conduct APPC conversations with partner programs running in the SNA network.

In its simplest form, such an application will attach a queue, by calling `pams_attach_q`, and conduct a dialog with the port server, by calling `pams_put_msg` and `pams_get_msg` (or `pams_get_msgw`), to exchange the seven predefined port server message types with the port server.

Simple Linear Conversations

It is possible to write an application that performs this dialog in a linear manner. In its simplest form, an application may conduct an LU6.2 conversation as follows:

```
pams_attach_q
pams_put_msg(connect_req)
pams_get_msgw(connect_accept)
pams_put_msg(data_message+change_direction)
pams_get_msgw(data_message)
pams_get_msgw(change_direction)
pams_put_msg(connection_terminated
pams_exit()
```

However, this approach assumes that the message to be received by a `pams_get_msg` is the message that the application expects. Because the BEA MessageQ system is a distributed queuing system, and because many things can happen in a distributed application, the message that arrives might not be the message expected by the application logic at that point in the conversation.

A better approach is to design the application as a simple *state machine* that performs initial application housekeeping and receives messages. Otherwise, extensive exception processing must be added to the logic to handle unexpected message types, which leads to more complex applications and a possible increase in logic errors.

State Machines

A simple state machine application performs initial application housekeeping, enters a loop in which a `pams_get_msgw` is issued to receive a message, and processes the message based on its type (see Figure 2-1).

In the routing that deals with the particular message type received, the application checks the current state to see if the message is a valid one, processes the message, sets a new state, and returns to the top of the loop.

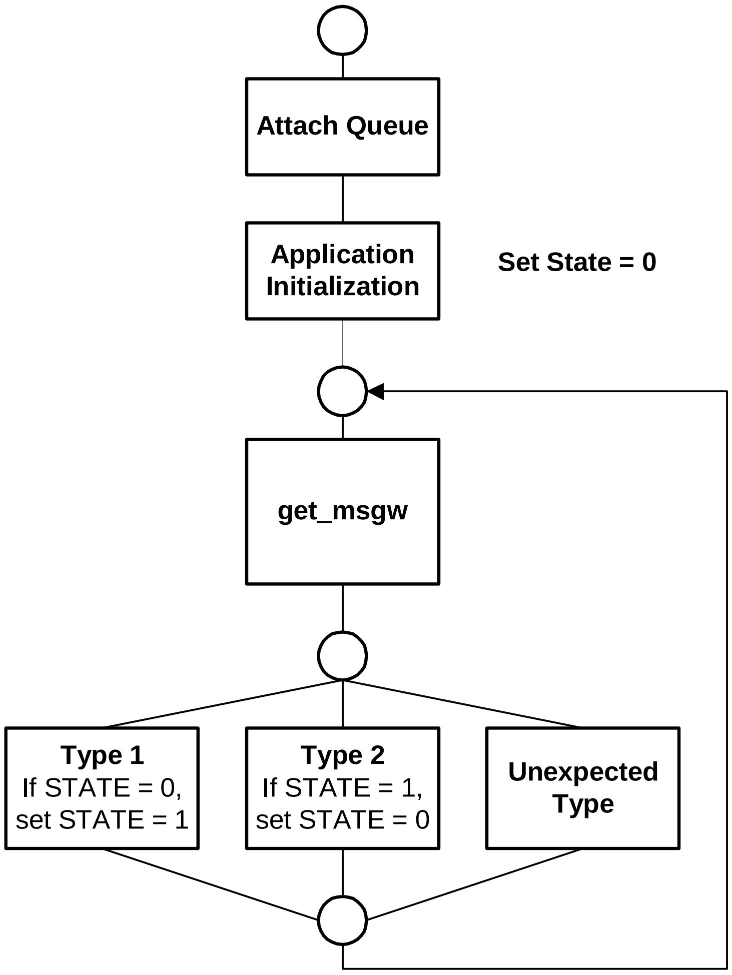
For the application loop described in Figure 2-1, assume that the application must receive a Type 1 message before it can receive a Type 2 message, and that all types other than 1 and 2 are invalid. A simple state machine implements this scheme as follows:

The application begins at `STATE=0`.

If a Type 1 message arrives and `STATE=0`, the message is valid and the new `STATE` is 1.

If a Type 2 message arrives and `STATE=1`, the message is valid, the new `STATE` is 0, and the process starts over.

Figure 2-1 Application Loop


Attach Queue
Application Initialization
Set State = 0
get_msgw
Type 1
If STATE = 0, set STATE = 1
Type 2
If STATE = 1, set STATE = 0
Unexpected Type

Overview of State/Event/Action Table

State machines can be simply documented using a state/event/action table. Table 2-1 describes the preceding application and shows each possible state in column 1, all events that can occur in that state in column 2, the action to be taken when that event occurs in column 3, and the new state in column 4.

Table 2-1 Sample State/Event/Action Table

State	Event	Action	New State
`STATE=0`	Type 1 msgs	process message	`STATE=1`
	Other msgs	report error	`STATE=0`
`STATE=1`	Type 2 msgs	process message	`STATE=0`
	Other msgs	report error	`STATE=1`

In some cases, the new state is the same as the original state; this allows the application to deal with unexpected events. In `STATE 0`, for example, receiving anything other than a Type 1 message leaves the application in `STATE 0`, so the next message received is subject to the same rules. This keeps the application from processing any mesages until a Type 1 message has been received.

The following sections provide basic State/Event/Action tables for inbound and outbound LU6.2 applications. The *events* listed in the Event column are the messages used to communicate with the port server (refer to Chapter 3). These tables were used to build the example programs listed in Appendix F.

Inbound State/Event/Action Listing

Table 2-2 describes an application that asks for a connection to an APPC partner program, sends it a message, waits for a response, and disconnects the conversation. This State/Event/Action table handles unexpected events and unexpected message types.

Table 2-2 Inbound State/Event/Action Table

State	Event	Action	New State
`connecting`	N/A	send connect	`wait_connect`
`wait_connect`	`connect_accept`	send data message and change direction	`wait_response`
	`connect_reject`	log error	`exiting`
	other	log error	`wait_connect`
`wait_response`	`data_message`	process response	`wait_complete`
	`change_direction`	log error and send abort message	`exiting`
	other	log error	`wait_response`
`wait_complete`	`change_direction`	send connection terminated (normal)	`exiting`
	`data_message`	log error and send abort message	`exiting`
	other	log error	`wait_complete`
`exiting`	N/A	call `pams_exit()`	`application done`

Outbound State/Event/Action Listing

Table 2-3 describes an application that registers to accept connections from a remote APPC partner program and then waits for data. After receiving a data message, it waits to become the sender, sends a response, and waits for a disconnect from the remote partner program.

Table 2-3 Outbound State/Event/Action Table

State	Event	Action	New State
`registering`	N/A	send register target message	`wait_register`
`wait_register`	`register_target`	N/A	`wait_data`
	`connection_terminated`	log error	`exiting`
	other	log error	`wait_register`
`wait_data`	`data_message`	process message	`wait_to_send`
	`change_direction`	log error and send abort message	`exiting`
	other	log error	`wait_data`
`wait_to_send`	`change_direction`	send data message plus `change_direction`	`wait_disconnect`
	`data_message`	log error and send abort message	`exiting`
	other	log error	`wait_to_send`
`wait_disconnect`	`connection_terminated`	N/A	`exiting`
	other	log error	`wait_disconnect`
`exiting`	N/A	call `pams_exit()`	application done

Development Checklist

When developing a distributed application, make sure that your process includes the following development steps:

1. Define the application boundaries.

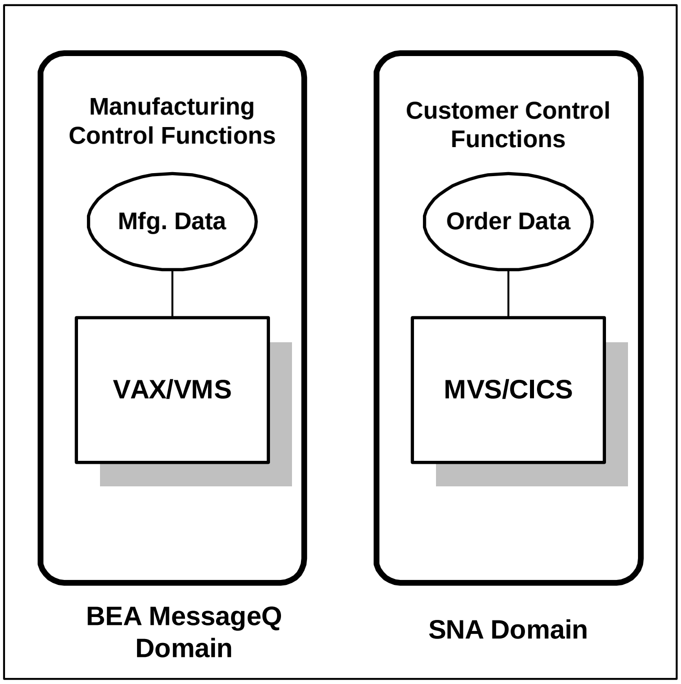
2. Identify the communicating partners.
3. Design the application conversations.
4. Develop the application.
5. Define the communications environment.
6. Test the application.

Step 1: Define the Application Boundaries

The first step in developing a distributed application, especially one that will run in a heterogeneous network, is to determine the *application boundaries*. This process consists of analyzing the functions that must be performed and the data that those functions will act upon, and then identifying the location (domain) in the network where those data and functions “naturally” reside.

For example, if the application is intended to integrate customer order entry with manufacturing control, you might determine that the customer order database is stored in DB2 under CICS on an MVS system, and the shop floor control database is stored in Rdb on an OpenVMS system. Functions that manipulate the customer order data will “naturally” reside on the MVS system and functions that manipulate the shop floor control data will “naturally” reside on the OpenVMS system. Figure 2-2 describes application domains.

Figure 2-2 Integrated Application Domains



Step 2: Identify the Communicating Partners

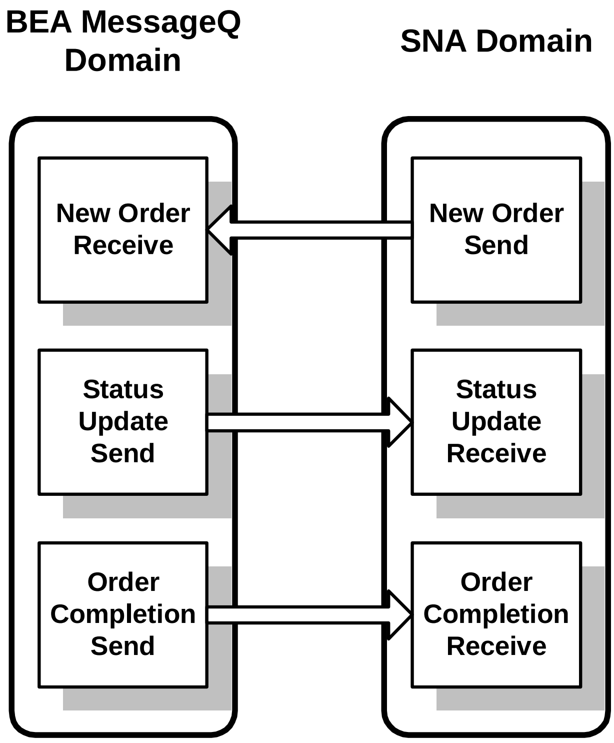
After the functions and data have been associated with network locations (the BEA MessageQ part of the network and the SNA part of the network, respectively), the functions must be mapped onto the processes that will implement them. As shown in Figure 2-2, you can assume that the customer order functions and manufacturing control functions have already been implemented in the existing application systems. In this case, you are concerned with identifying the new processes that will implement the new communications functions. Assume you must add the following functions:

- **New Order Transfer**: When the Customer Order system accepts a new order, the order is to be transferred immediately to the manufacturing control system for execution.

- **Order Status Update**: As the order is moved through the manufacturing process, a status update is to be delivered to the Customer Order system indicating:
  - Last manufacturing step completed
  - Scheduled start and end time of the next operation
  - Updated estimated time of delivery of the finished order
- **Order Completion**: When the order is completed and ready for shipment, an *order complete* status must be delivered to the Customer Order system.

Each of these three functions requires two communicating partners---one in the BEA MessageQ domain and one in the SNA domain (see Figure 2-3).

Figure 2-3 Communication Partners in Application



Now you can see what new processes must be added to the applications running in each domain, who the communicating partners will be, and how the communications flow will be initiated.

In this example, there is one outbound conversation, initiated by the New Order Send function, and two inbound conversations, initiated by the Status Update Send and Order Completion Send functions, respectively.

Step 3: Design the Application Conversations

For each pair of communicating partners, you must design the application conversation. This is the actual exchange of messages between communicating partners.

To design the application dialog, you must know:

- Who will initiate the conversation
- The format of each message
- How and when the roles of sender and receiver will be exchanged
- Who will terminate the conversation
- How errors will be handled

For this example, assume that the rules are very simple (which is usually true):

- The party initiating the conversation is responsible for terminating it.
- All errors are fatal; the conversation is terminated immediately when an error occurs.
- All conversations consist of one or more messages sent by the initiating party.
- When the initiating party is finished sending, it becomes the receiver.
- When the accepting party sees the initiating party become a receiver, it sends an acknowledgment, and switches itself back to a receiver.
- When the initiating party receives acknowledgment and regains control of the conversation (in other words, becomes the sender), it terminates the conversation.

The application conversation between New Order Send and New Order Receive looks like this:

New Order Send	New Order Receive
Initiate conversation	Accept new conversation
Send New Order message	Receive New Order message
Become receiver	Receive `OK_TO_SEND`
Receive acknowledgment	Send acknowledgment
Receive `OK_TO_SEND`	Become receiver
Terminate conversation	Accept termination

Step 4: Develop the Application

After the application conversations and message formats have been defined, the normal processes of application development (detail design, coding, and unit testing) can take place.

Note: Refer to Chapter 3, “Configuring the LU6.2 Port Server,” Chapter 4, “Port Server Messages,” and Chapter 6, “LU6.2 User Callback Services,” for information on Port Server and User Callback messages.

Note: Refer to the *BEA MessageQ Programmer’s Guide* for BEA MessageQ programming information.

Step 5: Define the Communications Environment

Before integration testing can occur, the communications environment must be defined. The full set of definitions that must be in place varies, based on the specific hardware, operating systems, and application subsystems involved.

Assuming a typical configuration consisting of a channel-attached SNA Gateway and MVS with VTAM and CICS, the following definitions must be available:

- ♦ Physical devices (the gateway) must be defined to VTAM.
- ♦ LU6.2 logical units that the gateway will provide must be defined to VTAM.

- LU6.2 “terminals” must be defined to CICS and mapped onto the LUs defined to VTAM.

  Note: Most sites use the CICS Resource Definition Online task to manage this function.

- CICS programs must be assigned Transaction Program Names (TPNs).
- Access names must be defined to identify the specific groups of gateway LUs that BEA MessageQ LU6.2 Services will use.
- BEA MessageQ programs that will accept outbound conversations must be assigned TPNs.
- An LU configuration file must be created that defines the gateway node names, access names, and specific LUs that LU6.2 Services will use (defined earlier in this step).
- A target configuration file must be created that defines the “target” names used by BEA MessageQ applications in connecting the LU6.2 Port Server and maps them onto TPNs defined in steps 4 and 6.

Note: Refer to Chapter 3, “Configuring the LU6.2 Port Server,” for more information on LU and target configuration files and the use of the LU6.2 Port Server.

Step 6: Test the Application

With the communications environment defined, you can now begin testing your application. If your application implements a state machine that is documented with a state/event/action table, developing a test plan that will validate correct behavior in each state is relatively straightforward.

Refer to Appendix C, “Examples of BEA MessageQ LU6.2 Inbound and Outbound Applications,” for samples of inbound and outbound applications.

Developing a Sample Application

The following are the specifications of a sample application. For the purposes of the example, assume that:

- The gateway node name is `SNAGWY`.
- The access name is `ORDERS`.
- This application is assigned LU numbers 1, 2, and 3.
- The two inbound CICS programs are given the following TPNs:
  - `ORDU`—Order Update Receive
  - `ORDC`—Order Completion Receive
- One outbound BEA MessageQ program is given the following TPN: `NEWORDER` —New Order Receive

Using the development process described in this section, this example produces the two BEA MessageQ LU6.2 Port Server initialization files: `ORDERS.LU` (inbound or resources file) and `ORDERS.TGT` (outbound or target file). Refer to Chapter 3, “Configuring the LU6.2 Port Server,” for information about recalling and editing these initialization files.

Listing 2-1 shows these two initialization files.

Listing 2-1 Sample Resources and Target Initialization Files

```
ORDERS.LU

!

!  One LU for use by OUTBOUND Conversations
    !
    !Resource   Gateway   Access   LU  Type
    CICSOUT     SNAGWY    ORDERS   1   2
    !
    ! Two LUs for use by INBOUND Conversations
    !
    !Resource   Gateway   Access   LU   Type
    CICSIN      SNAGWY    ORDERS   2    1
    CICSIN      SNAGWY    ORDERS   3    1
```

```
ORDERS.TGT
!
    ! One Target definition for OUTBOUND Conversations
    !
    !Target TPN     Resource     Type      Comm    Deallocate
    !                                      Type    Type
    NEWORDER        NEWORDER     CICSOUT   2       2        1
    !
    ! Two Target Definitions for INBOUND Coinversations
    !
    !Target TPN     Resource     Type      Comm    Deallocate
    !                                      Type    Type
    UPDATE          ORDU         CICSIN    1       2        1
    COMPLETE        ORDC         CICSIN    1       2        1
```

3 Configuring the LU6.2 Port Server

This chapter describes how to configure, start up, and manage the LU6.2 Port Server.

Note: Before you configure or use your LU6.2 Port Server, make sure that the BEA MessageQ LU6.2 Services software is properly installed and operational on your system. Refer to *Installing BEA MessageQ LU6.2 Services for OpenVMS* for installation information.

Specific topics covered in this chapter include:

- Port Server Functions
- Port Server Limits of Operation
- Configuring the Port Server
- Configuring Inbound and Outbound Connections
- Defining Logical Names
- Managing the LU6.2 Port Server

Port Server Functions

The LU6.2 Port Server is a connection-point management software tool that provides network applications (in this case, LU6.2 type) connection and namespace mapping services. The LU6.2 Port Server uses predefined messages (verbs) to simplify setting and managing applications' connections of the BEA MessageQ clients to remote partners.

To provide these functions, the LU6.2 Port Server must know:

- What SNA resources are available
- What names exist in both BEA MessageQ and SNA namespaces

Therefore, before you can use the LU6.2 Port Server, you need to set up the context (targets and resources) in which it operates. This means that you must determine and prepare the information that the LU6.2 Port Server requires to configure itself when it is initialized.

To summarize, to configure a port server, you must configure the following:

- IBM system(s)
- SNA gateway(s)
- Targets
- Resources
- Startup options

Port Server Limits of Operation

Use of the LU6.2 Port Server is restricted by the limits described in Table 3-1. You can configure the LU6.2 Port Server only within these limits.

Table 3-1 Port Server Operational Limits

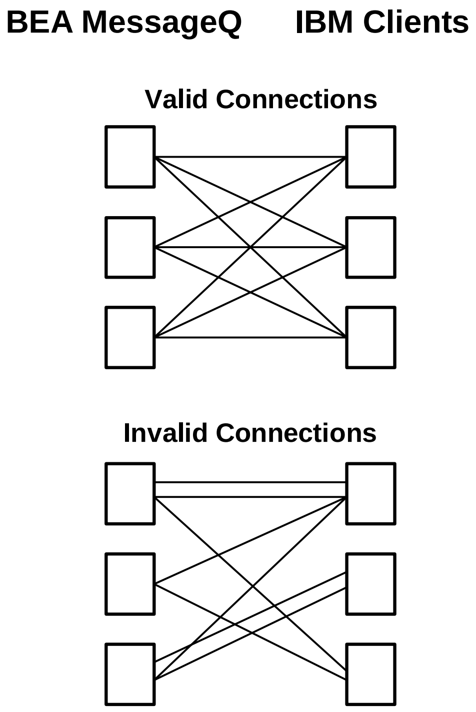
Limit	Description
Targets	Targets are defined by a user as part of the LU6.2 Port Server configuration process. A maximum of 512 targets can be defined for any LU6.2 Port Server.
SNA Logical Units	A maximum of 256 SNA LUs can be defined for any LU6.2 Port Server.
Concurrent Sessions	A maximum of 256 concurrent LU6.2 sessions are supported for any LU6.2 Port Server. The available pool of LUs is divided into inbound and outbound groups; the sum of the active sessions in each group cannot exceed 256 at any one time.
Message Size	The maximum data message size is 31982 bytes. This limit is imposed by the BEA MessageQ maximum message size of 32000 bytes. An 18-byte header is generated internally by LU6.2 Services for OpenVMS, leaving 31982 bytes for user data.
Target Sync Level	The LU6.2 Port Server supports both `SYNC_LEVEL 0` (`NONE`) and `SYNC_LEVEL 1` (`CONFIRM`). Support for `SYNC_LEVEL` is controlled globally by the `DMQLU62$SELECT_SYNC` and `DMQLU62$DISABLE_CONFIRM` logical names, and at the individual target level through the `SYNC_LEVEL` option on an extended target definition. If the logical name `DMQLU62$SELECT_SYNC` is not defined, defining the logical name `DMQLU62$DISABLE_CONFIRM` disables `SYNC_LEVEL 1` support. `SYNC_LEVEL 1` is supported as follows: A Change Direction request from a BEA MessageQ client results in a `PREPARE_TO_RECEIVE` verb being issued at `SYNC_LEVEL 1`. A Terminate Connection request from a BEA MessageQ client results in a `DEALLOCATE` verb being issued at `SYNC_LEVEL 1`. The LU6.2 Port Server automatically and unconditionally issues a `CONFIRMED` message in response to any `CONFIRM` verb issued by an IBM client.

Multiple Connections for BEA MessageQ Clients	BEA MessageQ clients are allowed multiple active connections to IBM clients. However, a BEA MessageQ Client can have only *one* active connection to any *one* IBM client (see Figure 3-1). The LU6.2 Port Server provides context information when each connection is established, enabling the BEA MessageQ client to distinguish connections from each other. It is the responsibility of the BEA MessageQ client to present the correct context information to the LU6.2 Port Server when using a previously established connection.
Multiple Connections for IBM Clients	IBM clients can initiate connections to multiple BEA MessageQ clients. However, any IBM Client can have only *one* active connection to any *one* target (see Figure 3-1). Because BEA MessageQ clients can register themselves with multiple target names, an IBM client can have multiple connections to a single BEA MessageQ client.
Security	Support for inbound conversation security is provided to those VTAM application programs that support this APPC feature, such as CICS. BEA MessageQ clients can present a user name, password, and profile when obtaining a connection through the LU6.2 Port Server. The LU6.2 Port Server presents these values to the VTAM application program when allocating the conversation on behalf of the BEA MessageQ client.

Security File	Security support for inbound connection requests is provided through a security file. The file specifies which permanent processes are allowed to initiate an inbound connect request. You can create a security file with any text editor. Each record must have a group and queue number. A logical name in the LNM process table, `DMQLU62$SECURITY_FILE`, must be defined as the full path name of the security file. The file is read at Port Server startup, and only processes defined in this file are allowed to initiate inbound conversations. If no logical name is defined, no security checking will occur. A sample security file follows:
	
	`! Dmq LU62 Security File`
	`! Group Queue`
	`    3     4`
	`    3     5`
	`  305    12`

Figure 3-1 describes valid and invalid connections between the BEA MessageQ (SNA) and IBM (CICS) clients and partners. The invalid connections are the multiple connections (two or more) between two network partners.

Figure 3-1 Valid and Invalid BEA MessageQ and IBM Multiple Connections



Configuring the Port Server

When initialized, the LU6.2 Port Server builds two tables—`LU_CONFIG` and `TARGET_CONFIG`. These tables must contain the following information:

- `LU_CONFIG`—SNA Resources

  Each resource definition identifies an SNA Gateway node name, access name, and session, as well as the `RESOURCE NAME` to which the definition belongs.

The table of SNA resources is used at run time to maintain context information about each active connection that requires an SNA LU.

- Valid Destinations (`TARGET_CONFIG`)

  Target definitions translate BEA MessageQ names into IBM names. Each target definition identifies the `RESOURCE NAME` used to establish a connection to that target.

  The table of valid destinations is used at run time to establish connections and to find entries in the table of SNA resources to track the connections as they are established.

These tables are built by reading two text files, known as *configuration files*, which are identified by the following logical names:

- `DMQLU62$SERVER_LU_CONFIG`, which contains entries for the table of SNA resources, `LU_CONFIG.TXT`
- `DMQLU62$SERVER_TARGET_CONFIG`, which contains entries for the table of valid destinations, `TARGET_CONFIG.TXT`

The following two sections describe how to prepare the configuration files that the LU6.2 Port Server uses to build the LU and TARGET configuration tables.

Building the LU Configuration File

The LU configuration file defines the SNA resources required by the LU6.2 port server:

- Resource name (`LU_SYSTEM_ID`)
- Gateway node (`LU_GATEWAY`)
- Gateway access name (`LU_ACCESS`)
- Gateway session (`LU_SESSION`)
- Resource type (`LU_TYPE`)
- Translation option
- Sync-level

To create or edit the LU configuration file, type

```
$ edit DMQLU62$SERVER_LU_CONFIG
```

Note: In the LU configuration file, data items are delimited by spaces or tab characters; any text following data items is treated as comments; and lines beginning with an exclamation point (!) or an asterisk (*) are treated as comments.

Listing 3-1 presents the LU configuration file format.

Listing 3-1 LU Configuration File Format

```
!                  LU CONFIGURATION FILE
!                  ======================
!
!    LOGICAL NAME: DMQLU62$SERVER_LU_CONFIG
!    FUNCTION:     DEFINES ALL SNA RESOURCES FOR PORT SERVER
!    FORMAT:       FREE-FORM POSITIONAL, WHITESPACE DELIMITED
!                  LEADING WHITESPACE REMOVED BEFORE PROCESSING
!                  LINES BEGINNING WITH "!" OR "*" ARE COMMENTS
!
!    FIELDS: LU_SYSTEM_ID      UP TO 8 CHAR
!            LU_GATEWAY        UP TO 6 CHAR
!            LU_ACCESS         UP TO 8 CHAR
!            LU_SESSION        3 NUMERIC, 0 = ANY
!            LU_TYPE           2 NUMERIC (1,2 DEFINED)
!
!ALL TEXT FOLLOWING THE POSITIONAL FIELDS IS A COMMENT
!
! 4 LU DEFINITIONS INBOUND TO CICS ADDRESS SPACE 1
!
!LU_SYSTEM_ID LU_GATEWAY LU_ACCESS LU_SESSION LU_TYPE
   CICS01      SNAGW1    DECLU62   0         1      LU FOR INBOUND USE TO CICS01
   CICS01      SNAGW1    DECLU62   0         1      LU FOR INBOUND USE TO CICS01
   CICS01      SNAGW1    DECLU62   0         1      LU FOR INBOUND USE TO CICS01
   CICS01      SNAGW1    DECLU62   0         1      LU FOR INBOUND USE TO CICS01
 !
! 2 INBOUND LUS FOR CICS 2
!
   CICS02      SNAGW1    DECLU62B  0         1      LU FOR INBOUND USE TO CICS02
   CICS02      SNAGW1    DECLU62B  0         1      LU FOR INBOUND USE TO CICS02
 !
! 3 LU DEFINITIONS OUTBOUND FROM CICS ADDRESS SPACE 1
!
  CICSOUT1    SNAGW1    DECLU62   200       2      LU FOR OUTBOUND USE FROM CICS01
  CICSOUT1    SNAGW1    DECLU62   201       2      LU FOR OUTBOUND USE FROM CICS01
  CICSOUT1    SNAGW1    DECLU62   202       2      LU FOR OUTBOUND USE FROM CICS01
 !
! ONE OUTBOUND TRANSPARENT LU FOR IMS TO USE
!
  IMSTRANS    SNAGW1    DECLU62   210       3      TRANSPARENT OUTBOUND LU FOR IMS

! END
```

The data items required to define each resource are described in Table 3-2.

Table 3-2 LU Configuration File Data Items

Data Item	Description
`LU_SYSTEM_ID`	The system ID that uniquely identifies all LUs with common characteristics. It is used to locate an entry in the LU configuration file when a connection request is received for a given destination (target). The entry in the `TARGET_CONFIG` configuration file for this destination uses the `LU_SYSTEM_ID` to identify the `LU_CONFIG` entries that can be used to create and manage connections to this target. See “Building the Target Configuration File” for more information. The `LU_SYSTEM_ID` can be used to reserve blocks of LUs for use by particular processes. The degree of port contention can be controlled by adjusting the number of entries in the block. To eliminate contention, make the size of the block equal to the number of concurrent connections required.
`LU_GATEWAY`	The SNA Gateway through which this LU is accessed.
`LU_ACCESS`	The SNA Gateway access name by which this LU is accessed.
`LU_SESSION`	The SNA Gateway session number. The session number is required for `LU_TYPE 2 OUTBOUND` and `LU_TYPE 3 OUTBOUND TRANSPARENT` connections (described next) because the specific session number is required to activate the specified sessions for use by VTAM. We recommend using the `LU_TYPE 1 INBOUND` session number connection to facilitate problem diagnosis.

LU_TYPE	The type of connection for which the LU is being reserved. Three types of connections are available: 1 = INBOUND The LU is used for communication with IBM transactions on demand, when requested by an OpenVMS process. 2 = OUTBOUND The LU is used for communication with OpenVMS transactions on demand when requested by an IBM process. 3 = OUTBOUND TRANSPARENT The LU is used for communication with OpenVMS transactions on demand when requested by an IBM process. Received data is not translated from EBCDIC data format to ASCII data format. No LU can be used for both inbound and outbound traffic. This restriction prevents contention for resources between the LU6.2 port server and VTAM.

Building the Target Configuration File

The Target configuration file defines all valid destinations known to the LU6.2 Port Server. The file translates the target name that is known to the sender program into a name that is known by the receiver program. This isolates sender programs from changes in the receiver program system.

Target definitions include:

- Target name (for BEA MessageQ destinations) (TARGET_NAME)
- TP name (for IBM) (TARGET_TPN)
- Resource name (TARGET_SYSTEM_ID)
- Target type (TARGET_TYPE)
- Communication type (COMMUNICATION_TYPE)
- Deallocation rule (DEALLOCATE_TYPE)
- Delivery mode

To create or edit this file, type

```
$ edit DMQLU62$SERVER_TARGET_CONFIG
```

Note: In the target configuration file, data items are delimited by spaces or tab characters; any text following data items is treated as comments; and lines beginning with an exclamation point (!) or an asterisk (*) are treated as comments.

Listing 3-2 presents the target configuration file format.

Listing 3-2 TARGET Configuration File Format

```
!                  TARGET CONFIGURATION FILE
!                  =========================
!
!    LOGICAL NAME: DMQLU62$SERVER_TARGET_CONFIG
!    FUNCTION:     DEFINES ALL SNA RESOURCES FOR PORT SERVER
!    FORMAT:       FREE-FORM POSITIONAL, WHITESPACE DELIMITED
!                  LEADING WHITESPACE REMOVED BEFORE PROCESSING
!                  LINES BEGINNING WITH "!" OR "*" ARE COMMENTS
!
!    FIELDS: TARGET_NAME         UP TO 8 CHAR
!            TARGET_TPN          UP TO 8 CHAR
!            TARGET_SYSTEM_ID    UP TO 8 CHAR
!            TARGET_TYPE         1 NUMERIC (1,2,3,4 DEFINED)
!            COMMUNICATION_TYPE  1 NUMERIC (1,2 DEFINED)
!            DEALLOCATE_TYPE     1 NUMERIC (1,2 DEFINED)
!
!    CICS TRANSACTION PROGRAM NAMES FOR INBOUND DUPLEX TRANSACTIONS TO CICS 01
!
! TARGET_NAME TARGET_TPN  TARGET_SYS_ID TARGET_TYPE COMM_TYPE DEALLOC_TYPE
!
  TRANS1      TRN1       CICS01       1           2         2    INBOUND TRANS 1
  TRANS2      TRN2       CICS01       1           2         2    INBOUND TRANS 2
! !    THE SAME TRANSACTIONS ARE ALSO ON CICS 2
!
  TRANS1A     TRN1       CICS02       1           2         2    INBOUND TRANS 1

  TRANS2A     TRN2       CICS02       1           2         2     INBOUND TRANS 2
!
!    TP NAMES TO BE ALLOCATED BY CICS
!
  VX01       VX01       VAX01        2                          OUTBOUND TRANS 1
  VX02       VX02       VAX01        2                          OUTBOUND TRANS 2
```

```
!
!    INBOUND EXTENDED TARGET
!
!=====================+ !TRANSLATION!=!SYNC LEVEL!
 TRANSIN    TRN3       CICS02       3          2       2   0 1 No Translation

!
!    OUTBOUND EXTENDED TARGET
!
!=====================  !TRANSLATION!SEND OPT!PERM!GRP!QUEUE!
  OUTPERM     VX03        VAX01        4            1        1    0 1 1 1 10
! END
```

The data items required to define each destination are described in Table 3-3.

Table 3-3 Target Configuration File Data Items

Data Item	Description
`TARGET_NAME`	The target name as it is known to BEA MessageQ application programs. For IBM targets, this is the name that OpenVMS application programs provide in the `CONNECT_REQUEST` message. For OpenVMS targets, this is the name that OpenVMS application programs provide in the `REGISTER_TARGET` message. **Note:** The `CONNECT_REQUEST` and `REGISTER_TARGET` messages are described in Port Server Connection Messages.
`TARGET_TPN`	The actual TPN known to the IBM system. The specific use varies as a function of `TARGET_TYPE`. For inbound targets, the `TARGET_TPN` is the IBM TPN as it is known to the VTAM application (CICS/VS or IMS/VS). For outbound targets, the `TARGET_TPN` is the TPN used by the IBM application program to allocate conversations with the LU6.2 Port Server.
`TARGET_SYSTEM_ID`	The `LU_SYSTEM_ID` used as a key when searching the LU configuration file for a valid entry to use in tracking the progress of the connection. For IBM targets, the `LU_NAME` in the selected LU configuration entry is used in the `LU62_ALLOCATE` message. For OpenVMS targets, the `LU_SYSTEM_ID` entry in the LU configuration file is used to store context information, but it does not contribute any information used in establishing the connection.

`TARGET_TYPE`	The method used to establish connections to the target. Valid values are: ♦ `1 = INBOUND` Inbound targets reside on IBM systems and are activated when application programs request connections to them. Inbound targets are nontransparent, which means that the LU6.2 Port Server translates messages sent to inbound targets from ASCII data format to EBCDIC data format. ♦ `2 = OUTBOUND` Outbound targets reside on any system connected to the BEA MessageQ message queuing bus. Outbound application programs must be registered with the LU6.2 Port Server for the target names that they support *before* conversation allocation from IBM application programs is accepted for those targets. Outbound targets are nontransparent, which means that the LU6.2 Port Server translates messages sent to outbound targets from the EBCDIC data format to ASCII data format.
`TARGET_TYPE` (cont.)	♦ `3 = INBOUND EXTENDED` Inbound extended targets reside on the IBM system and are activated when OpenVMS application programs request connections to them. If `TARGET_TYPE` is `INBOUND EXTENDED`, you must also indicate the `TRANSLATE OPTION` and `SYNC LEVEL` values. **Note:** You must specify these values *after* you specify the `TARGET_NAME`, `TARGET_TPN`, `TARGET_SYSTEM_ID`, `TARGET_TYPE`, `COMMUNICATION_TYPE`, and `DEALLOCATE_TYPE`. This means that you must specify the major `TARGET_TABLE` configuration file data items first, then you specify any additional `TARGET_TYPE` data items required. (See Listing 3-2 for clarification.) The `TRANSLATE OPTION` controls whether data translation service is provided for this target. Valid values are: 0—Do not translate (transparent) 1—Translate (nontransparent) The `SYNC_LEVEL` determines the synchronization level permitted on conversations with this target. Valid values are: 0—`SYNC_LEVEL=NONE` 1—`SYNC_LEVEL=CONFIRM`

TARGET_TYPE (cont.)	♦ 4 = OUTBOUND EXTENDED Outbound extended targets reside on any system connected to the BEA MessageQ message queuing bus. Outbound application programs must be registered with the LU6.2 Port Server for the target names that they support *before* conversation allocation from IBM application programs is accepted for those targets. If the TARGET_TYPE is OUTBOUND EXTENDED, you must also indicate the TRANSLATE OPTION, the SEND OPTION, and whether it is a PERMANENT TARGET. If it is a PERMANENT TARGET, then you must indicate the PERMANENT GROUP and the PERMANENT QUEUE. **Note:** You must specify these values *after* you specify the TARGET_NAME, TARGET_TPN, TARGET_SYSTEM_ID, TARGET_TYPE, COMMUNICATION_TYPE, and DEALLOCATE_TYPE. This means that you must specify the major TARGET_TABLE configuration file data items first, then you specify any additional TARGET_TYPE data items required. See Listing 3-2 for clarification. The TRANSLATE OPTION controls whether data translation service is provided for this target. Valid values are: 0—Do not translate (transparent) 1—Translate (nontransparent)

TARGET_TYPE (cont.)	The SEND OPTION controls the DELIVERY MODE parameter used on pams_put_msg calls that write messages to the message queue currently registered for the target. Valid values are: 0 = PDEL_MODE_NN_MEM, PDEL_UMA_RISC 1 = PDEL_MODE_WF_MEM, PDEL_UMA_RISC 2 = PDEL_MODE_WF_DQF, PDEL_UMA_SAF The PERMANENT TARGET value indicates whether this is a permanent outbound target. Valid values are: 0—This is not a permanent target 1—This is a permanent target **Note:** If the PERMANENT TARGET value is set to 1, you must specify PERMANENT GROUP and PERMANENT QUEUE. The PERMANENT GROUP specifies the BEA MessageQ group ID permanently registered for this target. The PERMANENT QUEUE value specifies the BEA MessageQ queue number permanently registered for this target.
COMMUNICATION_TYPE	The type of communication supported by the target. Valid values are: 1 = SIMPLEX The initiator of the conversation is always the sender program. The sender program and the receiver program cannot exchange roles. 2 = DUPLEX The initiator of the conversation is the initial sender program. The initial sender program can become the receiver program by exchanging roles with the remote partner program.
DEALLOCATE_TYPE	The partner that is allowed to deallocate the conversation. Note that *only the partner in send state* can issue a normal deallocate, regardless of the deallocate type. Valid values are: 1 = INITIATOR-ONLY Only the partner that *initiated* the conversation is allowed to deallocate the conversation. 2 = OPEN Either partner can deallocate the conversation when in the send state.

Configuring Inbound and Outbound Connections

The following tables provide the data that you must enter to properly configure inbound and outbound connections.

Configuring Inbound Connections

Use the following data to configure inbound connections (targets) supported by the LU6.2 Port Server. Table 3-4 lists the data items that you specify to configure inbound targets properly.

Table 3-4 Data Items for Configuring Inbound Targets

Data Item	Value
`TARGET_NAME`	The target name presented by the OpenVMS end client in the `CONNECT_REQUEST` message.
`TARGET_TPN`	The TPN of the IBM program as known to the VTAM application (CICS/VS or IMS/VS).
`TARGET_SYSTEM_ID`	The value of an `LU_SYSTEM_ID` in the `LU_TABLE` configuration file for an LU_TYPE 1 INBOUND LU as known to the VTAM application (CICS/VS or IMS/VS).
`TARGET_TYPE`	1 = `INBOUND`.
`COMMUNICATION_TYPE`	Specify 1 (simplex) if the BEA MessageQ client does not receive messages from the IBM system. Specify 2 (duplex) if the BEA MessageQ client receives messages from the IBM system.
`DEALLOCATE_TYPE`	Specify 1 if the IBM client does not issue `LU62_DEALLOCATE` messages. Specify 2 if `COMMUNICATION_TYPE` = 2 and the IBM client issues `LU62_DEALLOCATE` messages.

Configuring Outbound Connections

Use the following data to configure outbound connections supported by the LU6.2 Port Server. Table 3-5 lists the data items that you specify to configure outbound connections (targets) properly.

Table 3-5 Data Items for Configuring Outbound Targets

Data Item	Value
`TARGET_NAME`	The `TARGET_NAME` presented by the OpenVMS end client in the `REGISTER_TARGET` message.
`TARGET_TPN`	The OpenVMS TPN as it is known to the IBM end client.
`TARGET_SYSTEM_ID`	The value of an `LU_SYSTEM_ID` in the LU_TABLE configuration file for an LU_TYPE 1 INBOUND LU known to the VTAM application (CICS/VS or IMS/VS).
`TARGET_TYPE`	2—`OUTBOUND`.
`COMMUNICATION_TYPE`	Specify 1 (`SIMPLEX`) if the BEA MessageQ client does not send messages to the IBM system. Specify 2 (`DUPLEX`) if the BEA MessageQ client sends messages to the IBM system.
`DEALLOCATE_TYPE`	Specify 1 if the BEA MessageQ client does not issue `LU62_DEALLOCATE` messages. Specify 2 if `COMMUNICATION_TYPE` = 2 and the BEA MessageQ client issues `LU62_DEALLOCATE` messages.

Defining Logical Names

Two sets of logical names are provided for LU6.2 Services for OpenVMS. One set is provided for convenience in using and managing LU6.2 Services for OpenVMS by allowing easy access to the LU6.2 on-disk structure; the second set is used to set startup options in the LU6.2 Port Server.

Logical Names for the On-Disk Structure

The logical names used to access the LU6.2 Services for OpenVMS on-disk structure are as follows:

- `DMQLU62$SERVER_EXE`

  The device and directory specification of the directory containing the `.EXE`, `.COM`, and `.UID` files for the LU6.2 Port Server.

- `DMQLU62$SERVER_LIB`

  The device and directory specification of the directory that contains the message structure definitions for programs written in C, Pascal, PL/1, BASIC, FORTRAN, BLISS, and MACRO. This directory also contains the object files necessary to relink the LU6.2 Port Server.

- `DMQLU62$SERVER_SRC`

  The directory specifications of the C source for the application programming interface (API) shell are given in Chapter 5, "LU6.2 Port Server Application Programming Interface."

- `DMQLU62$SERVER_EXAMPLES`

  The device and directory specification of the directory containing example programs.

- `DMQLU62$SERVER_DOC`

  The device and directory specification of the directory that contains the LU6.2 Port Server documentation.

Logical Names for Port Server Control

The logical names used to control the LU6.2 Port Server are:

- `DMQLU62$SERVER_LU_CONFIG`

  The file specification of the file containing the LU definitions (`LU_CONFIG.TXT`)

- `DMQLU62$SERVER_TARGET_CONFIG`

The file specification of the file containing the target definitions (`TARGET_CONFIG.TXT`)

- `DMQLU62$SERVER_PAMS_PROCESS`

  The BEA MessageQ queue number to use in the `pams_attach_q` call. If not specified, the default is queue number 63.

- `DMQLU62$SERVER_UCB_ADDR`

  The BEA MessageQ queue number to use when sending messages to the LU6.2 user callback. If not specified, the default is queue number 62.

- `DMQLU62$SERVER_BROADCAST_STREAM`

  The BEA MessageQ broadcast stream to use when sending event messages. If not specified, the default is 4801.

- `DMQLU62$SERVER_RECONNECT_TIMER`

  The time to wait, in seconds, before sending a new `LU62_ACTIVATE` message for any LUs on which the prior activation attempt failed. The default is 900 (15 minutes); the minimum value is 60.

- `DMQLU62$SERVER_LOG_INFO`

  Define this logical name as any arbitrary value to enable logging for all successful conversation connects and disconnects, as well as the following operations:

  - Connect Request
  - Security Check
  - Connected
  - Connect Accept
  - Disconnect

  For this logical name to take effect, define it in the LNM table for the BEA MessageQ Bus and Group in which the Port Server is running.

  Unsuccessful conversation requests are always logged, regardless of whether or not this logical name is defined.

  Disabling log information reduces the size of the log files for Port Servers that have many conversation requests.

- `DMQLU62$SECURITY_FILE`

  This logical name is used to define the full path name of the security file. If this logical name is not defined, no security checking will occur.

- `DMQLU62$SERVER_MULTI_CONNECT`

  Define this logical name as any arbitrary value to allow multiple connections to the same INBOUND IBM target from a single BEA MessageQ program.

- `DMQLU62$SERVER_IMS_ADAPTER`

  Define this logical name as any arbitrary value to inform the LU6.2 port server that the IBM clients are being accessed using the IMS LU6.1 Adapter for LU6.2 applications.

  The LU6.2 Port Server changes its error-handling procedures to comply with the following restrictions imposed by the Adapter. Specifically, it does not issue:

  - The `SEND_ERROR` APPC verb
  - The `DEALLOCATE(ABEND_PROGRAM)` verb

Note: When using the IMS LU6.1 Adapter, you must provide for application-level data integrity checks because the normal facilities for obtaining confirmation (`SYNC_LEVEL` 1) and signaling errors are not available due to restrictions imposed by the IMS LU6.1 Adapter.

Refer to Appendix B, “Notes on IMS,” for more information on the IMS LU6.1 Adapter.

In addition to the logical names specifically checked by the LU6.2 Port Server, there are four logical names that affect the behavior of the LU6.2 User Callback, upon which the LU6.2 Port Server is based. These logical names are:

- `DMQLU62$BUFFER_SIZE`
- `DMQLU62$BUFFER_COUNT`
- `DMQLU62$SELECT_SYNC`
- `DMQLU62$DISABLE_CONFIRM`

These logical names are described in Appendix A, “LU6.2 User Callback Interface Logical Names and Error Codes.”

Managing the LU6.2 Port Server

This section describes the DCL command procedures and utility programs that are provided to start up, stop, and manage the LU6.2 Port Server.

Starting Port Servers

Run the `DMQLU62_SERVER_STARTUP.COM` procedure to start LU6.2 Port Servers. This command procedure is stored in `DMQLU62$SERVER_EXE` and performs the following functions:

- ♦ Defines the LU6.2 Port Server logical names
- ♦ Establishes the LU6.2 Port Server utilities as OpenVMS foreign commands
- ♦ Starts the LU6.2 Port Server as a detached process
- ♦ Runs the Event Watcher utility (described in "Watching Events") so you can observe the result of the LU6.2 Port Server initialization

The command format is:

```
@DMQLU62_SERVER_STARTUP Y que_id ps_id lu_config_file target_config_file
```

The command procedure takes the following arguments, all of which are optional:

- ♦ `Y` or `N`—Specify `Y` to start the servers. Specify `N` to set up logicals without starting the servers.
- ♦ *`que_id`*—The BEA MessageQ queue number to be assigned to the port server
- ♦ *`ps_id`*—The BEA MessageQ broadcast stream to be assigned to the port server for Event Messages
- ♦ *`lu_config_file`*—The `LU_CONFIG` configuration file to be used
- ♦ *`target_config_file`*—The `TARGET_CONFIG` configuration file to be used

Watching Events

Use the `DMQLU62_EVENT_WATCH` utility to watch the result of the LU6.2 Port Server initialization.

`DMQLU62_EVENT_WATCH.EXE` listens to the BEA MessageQ broadcast stream defined by the logical name `DMQLU62$SERVER_BROADCAST_STREAM` (if the logical name is not defined, it defaults to address 4801) and displays the event messages received on that stream.

The utility is stored in `DMQLU62$SERVER_EXE` and is defined as the foreign command `DMQLU62_EVENT_WATCH` by the `DMQLU62_SERVER_STARTUP.COM` procedure. The utility accepts a Stream MOT (Multipoint Outbound Target) address as the P1 command-line parameter. This allows multiple `EVENT_WATCH` programs in a single group to monitor different log stream MOTS. Therefore, when multiple Port Servers are run in a single group, each may be monitored by a separate `EVENT_WATCH` utility. The event watch display screen border shows the name of the log stream MOT on which it is listening. Pressing any key terminates the event watch utility after a 5-second delay.

The `DMQLU62_SERVER_STARTUP.COM` procedure starts the Event Watch program with a foreign command, and passes the defined MOT address as the P1 parameter.

When the Event Watch program is run interactively, it may be started by a run command or a foreign command. If started with a run command, the program translates the `DMQLU62$SERVER_BROADCAST_STREAM` if present or defaults to 4801 if absent. When started by a foreign command, the P1 parameter is used as the log stream MOT.

Defining Logical Names with DMQLU62_SERVER_LOGICALS.COM

Use the `DMQLU62_SERVER_LOGICALS.COM` procedure to define LU6.2 Port Server logical names.

The command format is:

```
@DMQLU62_SERVER_LOGICALS  device install_dir V40-VAX que_id ps_id lu_config_file
target_config_file
```

The command procedure takes the following arguments:

- `device`—The device on which the LU6.2 option is installed
- `install_dir`—The directory on which the LU6.2 option is installed
- `V40-VAX`—The version and architecture of the LU6.2 option that is installed
- `que_id`The BEA MessageQ queue number to be assigned to the port server
- `ps_id`——The BEA MessageQ broadcast stream to be assigned to the port server
- `lu_config_file` —The `LU_CONFIG` configuration file to be used
- `target_config_file`—The `TARGET_CONFIG` configuration file to be used

Stopping LU6.2 Port Servers

Run `DMQLU62_SERVER_STOP` to stop LU6.2 Port Servers. The `DMQLU62_SERVER_STOP` utility program sends a `SHUTDOWN` message to a designated BEA MessageQ address.

The utility is stored in `DMQLU62$SERVER_EXE` and is defined as the foreign command `DMQLU62_SERVER_STOP` by the `DMQLU62_SERVER_STARTUP.COM` procedure. The utility takes two arguments: the BEA MessageQ group ID and the queue number of the LU6.2 Port Server to be stopped.

For example:

```
$ DMQLU62_SERVER_STOP  5 63
```

where `5` = *`group_ID`* and `63` = *`port_server_queue_number`*.

4 Port Server Messages

BEA MessageQ clients establish and manage LU6.2 connections through the LU6.2 Port Server using predefined messages. There are two types of predefined messages used by the LU6.2 Port Server: port server control messages and port server connection messages (BEA MessageQ messages).

These messages are sent via the BEA MessageQ API function `pams_put_msg`. Message class and type definitions, which are used as arguments to `pams_put_msg`, are provided in the BEA MessageQ Class and Type file (`DMQ$TYPCLS.TXT`) at installation. Refer to Appendix B in the *BEA MessageQ Installation and Configuration Guide for OpenVMS* for a sample of `DMQ$TYPCLS.TXT`.

This chapter discusses the following topics:

- Port Server Control Messages
- Port Server Connection Messages
- Example of Port Server Messages Used for Client Communications

Port Server Control Messages

This section describes messages that control the LU6.2 Port Server.

Message	Description
`ADD_LU`	Dynamically adds an LU definition while the LU6.2 Port Server is running
`ADD_TARGET`	Dynamically adds a target definition while the LU6.2 Port Server is running

`SHUTDOWN`	Instructs the LU6.2 Port Server to exit

To send these control messages, use the `pams_put_msg` with the target of the port server and the message type of `message`.

ADD_LU

The ADD_LU message dynamically adds an LU definition while the LU6.2 port server is running. The ADD_LU message is formatted as a valid configuration file entry for LUs and TYPE 1 targets. See Chapter 3, "Configuring the LU6.2 Port Server," for more information about configuration files.

Listing 4-1 shows the C message structure for the ADD_LU service.

Listing 4-1 C Message Structure for ADD_LU

```
typedef struct _add_lu {
char   sysid[9];
char   gateway[7];
char   lu_access[9];
char   lu_sess[4];
char   lu_type[3];
int16  lu_session_dir;
} add_lu;

int16  msg_size;

...

class = MSG_CLAS_APPC;
type  = MSG_TYPE_LU62_ADD_LU;
msg_size = sizeof(struct add_lu);

dmq_status = pams_put_msg(
       &add_lu,
       &priority,
       &server_queue,
       &class,
       &type,
       &delivery,
       &msg_size,
       &timeout,
       &put_psb,
       &uma,
       (q_address *) 0,
       (int32 *) 0,
       (char *) 0,
       (char *) 0 );
```

ADD_TARGET

The ADD_TARGET message dynamically adds a TYPE 1 inbound target definition while the LU6.2 port server is running. The ADD_TARGET message is formatted as a valid configuration file entry for LUs and targets. See Chapter 3, “Configuring the LU6.2 Port Server,” for more information about configuration files.

Listing 4-2 shows the C message structure for the ADD_TARGET service.

Listing 4-2 C Message Structure for ADD_TARGET

```
typedef struct _add_tgt {

char   targ_name[9];
char   targ_tpn[9];
char   targ_sysid[9]
int16  comm_type;
int16  dealloc;
} add_tgt;

...

class = MSG_CLAS_APPC;
type  = MSG_TYPE_LU62_ADD_TGT;
msg_size = sizeof(struct add_tgt);

dmq_status = pams_put_msg(
       &add_tgt,
       &priority,
       &server_queue,
       &class,
       &type,
       &delivery,
       &msg_size,
       &timeout,
       &put_psb,
       &uma,
       (q_address *) 0,
       (int32 *) 0,
       (char *) 0,
       (char *) 0 );
```

SHUTDOWN

The SHUTDOWN message instructs the LU6.2 port server to exit. The SHUTDOWN message has no content; the BEA MessageQ message type is sufficient to convey the information.

Listing 4-3 shows the C message structure for the SHUTDOWN service.

Listing 4-3 C Message Structure for SHUTDOWN

```
char  msg_buf[1024];
int16 msg_size;

...

class = MSG_CLAS_APPC;
type  = MSG_TYPE_LU62_SHUTDOWN;
msg_size = 0;

dmq_status = pams_put_msg(
       &msg_buf,
       &priority,
       &server_queue,
       &class,
       &type,
       &delivery,
       &msg_size,
       &timeout,
       &put_psb,
       &uma,
       (q_address *) 0,
       (int32 *) 0,
       (char *) 0,
       (char *) 0 );
```

Port Server Connection Messages

This section describes messages either received from or sent to BEA MessageQ clients by the LU6.2 Port Server. These seven predefined messages allow BEA MessageQ clients to use the LU6.2 Port Server to establish and manage LU6.2 connections to remote partners using the port server as the standard API.

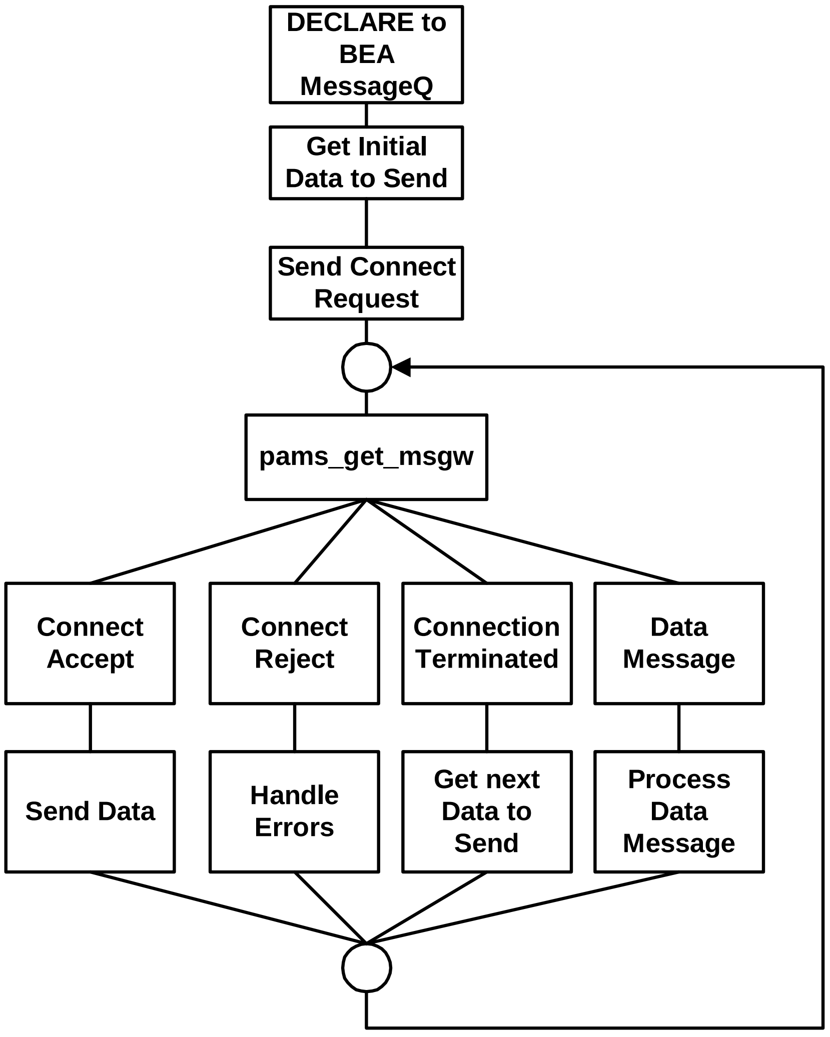
Table 4-1 lists the seven predefined port server messages.

Table 4-1 Summary of LU6.2 Port Server Messages

This message	Is used to . . .
`CHANGE_DIRECTION`	Indicate change of direction of connection. It may mean that the remote IBM client has become the receiver program, and that the BEA MessageQ client is now the sender program, or vice versa.
`CONNECT_ACCEPT`	Indicate that the requested connection has been established
`CONNECT_REJECT`	Indicate that the requested connection could not be established
`CONNECT_REQUEST`	Request a connection for a BEA MessageQ client to a remote LU6.2 partner
`CONNECTION_TERMINATED`	When sent to a BEA MessageQ client, indicate that the remote IBM client has terminated the connection. When sent by a BEA MessageQ client, request termination of the connection.
`DATA_MESSAGE`	When sent to a BEA MessageQ client, carry a data message received from the remote partner. When sent by a Message client, carry a data message to be transmitted to the remote partner.
`REGISTER_TARGET`	Map to a BEA MessageQ client a target name (including group ID and queue number) for registration purposes

Figure 4-1 shows a typical program structure that uses BEA MessageQ messages (verbs) to establish and manage data connections.

Figure 4-1 Typical Program Structure



Note: If the message field value is shorter than the required field length, it is necessary to enter null terminators (hex 0s).

The following sections describe each port server message and its format.

CHANGE_DIRECTION

The CHANGE_DIRECTION message indicates a change in the direction of the connection.

When the CHANGE_DIRECTION message is sent to the BEA MessageQ client, it indicates that the remote IBM client has become the receiver program, and that the BEA MessageQ client is now the sender program.

When the CHANGE_DIRECTION message is sent by the BEA MessageQ client, it indicates that the remote IBM client has become the sender program, and that the BEA MessageQ client is now the receiver.

Listing 4-4 shows the C message structure for the CHANGE_DIRECTION service.

Listing 4-4 C Message Structure for CHANGE_DIRECTION

```
typedef struct _change_direction {
       int32 connection_index;
       } change_direction;
```

MESSAGE DATA FIELDS

Field	**Data Type**	**Description**
CONNECTION_INDEX	word	Context value that uniquely identifies the connection that has changed direction

CONNECT_ACCEPT

The CONNECT_ACCEPT message is sent to the BEA MessageQ client to indicate that the requested connection has been established. This message contains a word (16-bit) context variable used by the LU6.2 port server to identify the connection. The context variable value must be stored by the BEA MessageQ client and provided in any subsequent message sent over the connection.

Listing 4-5 shows the C message structure for the CONNECT_ACCEPT service.

Listing 4-5 C Message Structure for CONNECT_ACCEPT

```
typedef struct _connect_accept {
       int16 connection_index;
       char target_name [8];
       } connect_accept;
```

MESSAGE DATA FIELDS

Field	Data Type	Description
CONNECTION_INDEX	word	Context value that uniquely identifies the connection
TARGET_NAME	text 8 char	Name of the target connected

CONNECT_REJECT

The CONNECT_REJECT message is sent to the BEA MessageQ client to indicate that the requested connection could not be established. The reason for the rejection is indicated in the body of the message.

Listing 4-6 shows the C message structure for the CONNECT_REJECT service.

Listing 4-6 C Message Structure for CONNECT_REJECT

```
typedef struct _connect_reject {
       char target_name [8];
       int32 reject_reason;
       } connect_reject;
```

MESSAGE DATA FIELDS

Field	Data Type	Description
TARGET_NAME	text 8 char	Name of the target rejected
REJECT_REASON	int32	Reason for the connect reject

CONNECT REJECT REASON CODES

- PAMSLU62_ALREADYCON
- PAMSLU62_BADSYSID
- PAMSLU62_BADTARGNAME
- PAMSLU62_BUSY
- PAMSLU62_WRONGTYPE

CONNECT_REQUEST

The CONNECT_REQUEST message is sent by the BEA MessageQ client to request a connection to a remote LU6.2 partner. This message contains the target name of the remote partner and, optionally, can contain security information to be presented to the VTAM application program when the conversation is allocated by the LU6.2 Port Server.

Listing 4-7 shows the C message structure for the CONNECT_REQUEST service.

Listing 4-7 C Message Structure for CONNECT_REQUEST

```
typedef struct _connect_request {
      char target_name [8];
      char username [10];
      char password [10];
      char profile [10];
      } connect_request;
```

MESSAGE DATA FIELDS

Field	**Data Type**	**Description**
TARGET_NAME	text 8 char	Name of the target for connection
USER_NAME	text 10 char	User name for security authentication
PASSWORD	text 10 char	Password for security authentication
PROFILE	text 10 char	Security profile for security authentication

CONNECTION_TERMINATED

When sent to a BEA MessageQ client, the `CONNECTION_TERMINATED` message indicates that the remote IBM client has terminated the connection. This message contains a field indicating the termination status (normal or abnormal).

When sent by a BEA MessageQ client, the `CONNECTION_TERMINATED` message requests termination of the connection. This message can be used to terminate the connection normally when the BEA MessageQ client is the sender program and no data messages are being sent. This message can also be used to terminate the connection abnormally, regardless of the current state (send or receive).

Listing 4-8 shows the C message structure for the `CONNECTION_TERMINATED` service.

Listing 4-8 C Message Structure for CONNECTION_TERMINATED

```
typedef struct _connection_terminated {
       int16 connection_index;
       int16 terminate_type;
       int32 terminate_reason;
       } connection_terminated;
```

MESSAGE DATA FIELDS

Field	**Data Type**	**Description**
`CONNECTION_INDEX`	word	Context value that uniquely identifies the connection to which this message is to be applied
`TERMINATE_TYPE`	word	Specifies the type of termination. Valid values are: 1—Disconnect (normal) 2—Disconnect (error)
`TERMINATE_REASON`	int32	Reason for termination This field is filled in by the LU6.2 port server when a connection is abnormally terminated by the IBM system. The field is ignored on messages sent *to* the LU6.2 port server.

DATA_MESSAGE

When sent to a BEA MessageQ client, the `DATA_MESSAGE` message contains the text of a data message received from the remote partner. The LU6.2 Port Server translates the data message from EBCDIC to ASCII before sending it, provided that translation is requested in the target definition.

When sent by a BEA MessageQ client, the `DATA_MESSAGE` message contains the text of a data message to be transmitted to the remote partner. The LU6.2 Port Server translates the data message from ASCII to EBCDIC before transmitting it, provided that translation is requested in the target definition. Control fields in `DATA_MESSAGE` allow the BEA MessageQ client application program to:

- Indicate that this is the last message
- Request a direction change and become the receiver program
- Terminate the connection, normally or abnormally

Listing 4-9 shows the C message structure for the `DATA_MESSAGE` service.

Listing 4-9 C Message Structure for DATA_MESSAGE

```
typedef struct _data_message {
       int16 last_message;
       int16 change_direction;
       int16 disconnect;
       int16 connection_index;
       char data [31982];
       } data_message;
```

MESSAGE DATA FIELDS

Field	Data Type	Description
`LAST_MESSAGE`	word	Indicates that the current message is the last in the current set. Valid values are: 0—False 1—True When set, the LU6.2 port server issues an explicit FLUSH.

CHANGE_DIRECTION	word	Indicates that the BEA MessageQ client wants to be the receiver program. Valid values are: 0—False 1—True When set, the LU6.2 Port Server issues a PREPARE_TO_RECEIVE message.
DISCONNECT	word	Indicates that the BEA MessageQ client wants to terminate the connection. Valid values are: 0—False 1—Disconnect (normal) 2—Disconnect (error)
CONNECTION_INDEX	word	Context value that uniquely identifies the connection to which this message is to be applied.
DATA	text	0 to 31982 bytes of data to be sent to the remote LU6.2 partner program.

REGISTER_TARGET

When sent by a BEA MessageQ client, the REGISTER_TARGET message maps a target name. This message contains the target name for the registration request and the BEA MessageQ queue address (group ID and queue number) of the application program to be registered.

Listing 4-10 shows the C message structure for the REGISTER_TARGET service.

Listing 4-10 C Message Structure for REGISTER_TARGET

```
typedef struct _register_target {
       char target_name [8];
       int16 target_group;
       int16 target_process;
       } register_target;
```

MESSAGE DATA FIELDS

Field	Data Type	Description
TARGET_NAME	text 8 char	Name of the target to register
TARGET_GROUP	word	BEA MessageQ group ID of the application program to register
TARGET_PROCESS	word	BEA MessageQ queue number of the application program to register

Example of Port Server Messages Used for Client Communications

Listing 4-11 shows port server messages used in a program to support LU6.2 client communications.

Listing 4-11 LU6.2 Port Server Program

```
typedef struct _connect_request {
       char target_name [8];
       char username [10];
       char password [10];
       char profile [10];
       } connect_request;

...

strncpy(connect_request.target_name, "MY_TARGET", 8);
strncpy(connect_request.username, "MY_USERNAME", 10);
strncpy(connect_request.password, "TOPSECRET", 10);
strncpy(connect_request.profile, "THEPROFILE", 10);


class = MSG_CLAS_APPC;
type  = MSG_TYPE_CONNECT_REQUEST;
msg_ptr = &connect_request;

do{
       status = put_msg(msg_ptr, class,type);
       if(status)
              return(status);

status = get_msg(msg_ptr, class, type);

}while(status == CONTINUE);

status
put_msg(msg_ptr, class, type)
char   *msg_ptr;
int16  class;
int16  type;
```

```
       {

...

dmq_status = pams_put_msg(
       msg_ptr,
       &priority,
       &server_queue,
       &class,
       &type,
       &delivery,
       &msg_size,
       &timeout,
       &put_psb,
       &uma,
       (q_address *) 0,
       (int32 *) 0,
       (char *) 0,
       (char *) 0 );
return(dmq_status);
}

status
put_msg(msg_ptr, class, type)
char   *msg_ptr;
int16  class;
int16  type;
       {

...

dmq_status = pams_get_msg(
       msg_ptr,
       &priority,
       &source,
       &class,
       &type,
       &msg_area_len,
       &size,
       (int32 *) sel_filter,
       (struct PSB *) 0,
       (struct show_buffer *) 0,
       (int32 *) 0,
       (int32 *) 0,
       (int32 *) 0,
       (char *) 0 );
       if(dmq_status)
return(dmq_status);
switch(type)  {


case MSG_TYPE_CONNECT_ACCEPT:
       send_data();
       break;

case MSG_TYPE_CONNECT_REJECT:
       error_routine();
       break;

case MSG_TYPE_CHANGE_DIRECTION:
       change_state();
       break;

case MSG_TYPE_DATA_MESSAGE:
       process_data_msg();
       break;

case MSG_TYPE_CONNECTION_TERMINATED:
       handle_termination();
       break;

default
       break;
}
return();
}

...
```

CHAPTER

5 LU6.2 Port Server Application Programming Interface

Users can simplify the development of BEA MessageQ application programs that use the LU6.2 Port Server by hiding the details of BEA MessageQ LU6.2 Services for OpenVMS within a shell of procedure calls.

This chapter presents a sample application programming interface (API) for LU6.2 Services for OpenVMS. The sample API consists of the following four procedure calls:

- `PORT_CONNECT`—Establishes a connection to the specified inbound target.
- `PORT_RECV`—Receives a message through any existing connection or through a new connection resulting from a previous registration.
- `PORT_REGISTER`—Specifies the register to receive output directed at the specified outbound target.
- `PORT_SEND`—Sends a message through a previously established connection.

The source code for these procedures is found in the following file:

```
DMQLU62$SERVER_SRC:PORT_FUN.C
```

Note: The procedure calls are portable to non-OpenVMS platforms with minimal code changes.

This chapter provides information, organized in the format of reference manual entries, about the following procedure calls:

- `PORT_CONNECT`
- `PORT_RECV`
- `PORT_REGISTER`
- `PORT_SEND`

PORT_CONNECT

This procedure establishes a connection to a specified inbound target.

Syntax

```
COND_VALUE=PORT_CONNECT(target_name, ... port_queue)
```

Arguments

Argument	Data Type	Mechanism	Prototype	Access
target_name	char	reference	char *	passed
connection_index	word	reference	short *	returned
port_group	word	value	short	passed
port_queue	word	value	short	passed

Argument definitions

target_name
The name of the target to which to establish the connection.

connection_index
The unique identifier of the requested connection.

port_group
The BEA MessageQ group ID of the LU6.2 Port Server that connects to the specified target.

port_queue
The BEA MessageQ queue number of the LU6.2 Port Server that connects to the specified target.

DESCRIPTION

This procedure sends a `CONNECT_REQUEST` message for the specified target to the designated generic port server and waits for a response.

RETURNS

Argument	Data Type	Mechanism	Prototype	Access
cond_value	longword	value	long	returned

RETURN VALUES

Return Code	Description
`SS$_NORMAL`	A connection is successfully completed.
`PAMSLU62_ALREADYCON`	A connection has already been established to the named target.
`PAMSLU62_BADSYSID`	The port server target definition specifies an undefined value for `SYS_ID`.

`PAMSLU62_BADTARGNAME`	The named target has not been defined to the port server.
`PAMSLU62_BUSY`	All paths to the named target are currently in use.
`PAMSLU62_WRONGTYPE`	The named target is defined as OUTBOUND.
`PAMS_`*`xxxxxxxx`*	This indicates any PAMS status code returned by `pams_put_msg` or `pams_get_msgw`.

Example The following is an example of the `PORT_CONNECT` procedure call.

```
#include stdio
#include "port_fun.h"
#include "p_entry.h"

#define TRUE 1
#define FALSE 0


main()
{
       int32 p_status;
       int32 req;
       long status;
       q_address used;
       short send_connection;

       req = 0;

       p_status = pams_attach_q(&req,&used);
       if (!(p_status & 1)) return(p_status);

       status = port_connect("MY_TARGET",
              &send_connection,
              3,
              63);

       if (!(status & 1))
       {
       p_status = pams_exit();
       return(p_status);
       }
.
.
.
}
```

PORT_RECV

This procedure receives a data message from a remote IBM application program through a previously established connection or through a new connection resulting from a previous `PORT_REGISTER` procedure call.

Syntax `COND_VALUE=PORT_RECV(`*`message, buf_size, ... port_queue`*`)`

Arguments

Argument	Data Type	Mechanism	Prototype	Access
message	char	reference	char *	returned
buf_size	word	value	short	passed
msg_size	word	reference	short *	returned
connection_index	word	reference	short *	returned
change_dir	word	reference	short *	returned
disconnect	word	reference	short *	returned
abort	word	reference	short *	returned
port_group	word	reference	short *	returned
port_queue	word	reference	short *	returned

Argument definitions

message

The user buffer to contain the message received from the IBM application program.

buf_size

The size of the user buffer to contain the received message. Messages too large to fit in the buffer are truncated to the buffer size.

msg_size

The size of the returned message, or the buffer size, if the message is larger than the buffer.

connection_index

The unique identifier of the connection on which the message was received.

change_dir

Indicates that the direction of the connection was reversed.

disconnect

When set to a non-zero value, indicates that the remote IBM client has terminated the conversation normally.

abort

When set to a non-zero value, indicates that the remote IBM client has terminated the conversation abnormally.

port_group

The BEA MessageQ group ID of the generic port server that sent the message.

port_queue

The BEA MessageQ queue number of the generic port server that sent the message.

Description

This procedure waits to receive a message from a remote IBM application program. The message can be received through an established connection or through a connection initiated by the remote IBM application program using a PORT_REGISTER procedure call. The connection on which the message arrives is identified by the *connection_index* argument. By setting the *change_dir*, *disconnect*, or *abort* flags, the direction of the message flow can be changed, buffers at the generic port server can be flushed, or the connection can be terminated (normally or abnormally).

Returns

Argument	**Data Type**	**Mechanism**	**Prototype**	**Access**
cond_value	longword	value	long	returned

Return values

Return Code	**Description**
SS$_NORMAL	Indicates successful completion.
PAMSLU62_BADINDEX	The *connection_index* provided is invalid.
PAMSLU62_ CONABORTDATA	The connection has been aborted by the port server due to a nontranslatable ASCII character in the body of the message.
PAMSLU62_CONABORTSTATE	The connection has been aborted by the port server due to a violation of the selected application protocol.
PAMSLU62_NOCONNECT	No connection has been established.
PAMS_*xxxxxxxx*	Any PAMS status code returned by pams_put_msg.

Example The following is an example of the PORT_RECV procedure call.

```
#include stdio
#include signal
#include "port_fun.h"
#include "p_entry.h"
#include "p_return.h"

#define TRUE 1
#define FALSE 0

main()
{
       int32 p_status;
       long  status;
       char reply[1024];
       short get_disc = 0;
       short buf_siz;
       .
       .
       .
       buf_size = sizeof(reply);

       while (!get_disc)
       {
       status = port_recv(reply,
       buf_siz,
       &msg_siz,
       &recv_connection,
       &get_cdi,
       &get_disc,
       &get_abort,
       &source_group,
       &source_process);

        if (!((status & 1)||(status == PAMS__TIMEOUT)))
       {
       p_status = pams_exit();
       return(p_status);
       }
              .
              .
              .
       }
```

PORT_REGISTER

This procedure specifies the queue to receive the output directed to a specified target by a remote IBM application program.

Syntax

```
COND_VALUE=PORT_REGISTER(target_name, ... reg_queue)
```

Arguments

Argument	Data Type	Mechanism	Prototype	Access
target_name	char	reference	char *	passed
port_group	word	value	short	passed
port_queue	word	value	short	passed
reg_group	word	value	short	passed
reg_queue	word	value	short	passed

Argument definitions

target_name
The name of the target with which to register.

port_group
The BEA MessageQ group ID of the generic port server that connects to the specified target.

port_queue
The BEA MessageQ queue number of the generic port server that connects to the specified target.

reg_group
The BEA MessageQ group ID of the process to register.

reg_queue
The BEA MessageQ queue number of the process to register.

Description

This procedure sends a `REGISTER_TARGET` message for the specified target to the designated generic port server and waits for a response.

Returns

Argument	Data Type	Mechanism	Prototype	Access
cond_value	longword	value	long	returned

Return values

Return Code	Description
`SS$_NORMAL`	The procedure is successfully completed.

PAMSLU62_ALREADYREG	The process has already registered with the named target.
PAMSLU62_BADSYSID	The port server target definition specifies an undefined value for SYS_ID.
PAMSLU62_BADTARGNAME	The named target has not been defined to the port server.
PAMS__*xxxxxxxx*	This indicates any PAMS status code returned by pams_put_msg(w).

Example The following is an example of the PORT_REGISTER procedure call.

```
#include stdio
#include "port_fun.h"
#include "p_entry.h"
#include "p_return.h"

#define TRUE 1
#define FALSE 0

main()
{
       int32 p_status;
       long  status;
       int32 req;
       q_address used;

       req = 0;
       p_status = pams_attach_q(&req,&used);
       if (!(p_status & 1)) return(p_status);

       status = port_register("MY_TARGET",
        3,
        63,
        used.au.group,
        used.au.process);

       if (!(status & 1))
{
p_status = pams_exit();
return(p_status);
}

.
.
.
}
```

PORT_SEND

This procedure sends a data message to a remote IBM application program through a previously established connection. The maximum size of a data message is 31982 bytes.

Syntax

```
COND_VALUE=PORT_SEND(message, connection_index, ... port_queue)
```

Arguments

Argument	Data Type	Mechanism	Prototype	Access
message	char	reference	char *	passed
connection_index	word	value	short	passed
change_dir	word	value	short	passed
last	word	value	short	passed
disconnect	word	value	short	passed
abort	word	value	short	passed
port_group	word	value	short	passed
port_queue	word	value	short	passed

Argument definitions

message

The message text (up to 31982 bytes) to be sent to the IBM application program.

connection_index

The unique identifier of the connection returned by the previous `PORT_CONNECT` procedure call.

change_dir

Changes direction of message flow from Send to Receive when set to a nonzero value.

last

Indicates the last message in a series. When set to a non-zero value, this argument causes the port server to transmit any untransmitted traffic on the indicated connection. To deallocate the LU6.2 conversation, use the *disconnect* argument.

disconnect

Indicates the last message in a series. When set to a non-zero value, this argument causes the port server to transmit any untransmitted traffic on the indicated connection and deallocates the LU6.2 conversation normally.

abort

Indicates an error. When set to a non-zero value, this argument causes the port server to deallocate the LU6.2 conversation abnormally.

port_group

The BEA MessageQ group ID of the generic port server that connects to the specified target.

port_queue

The BEA MessageQ queue number of the generic port server that connects to the specified target.

Description

This procedure sends a message to the remote IBM application program through a previously established connection identified by the *connection_index* argument. By setting the *change_dir*, *last*, *disconnect*, or *abort* flags, the direction of message flow can be changed, buffers at the LU6.2 Port Server can be flushed, or the connection can be terminated (normally or abnormally).

Returns

Argument	Data Type	Mechanism	Prototype	Access
cond_value	longword	value	long	returned

Return values

Return Code	Description
SS$_NORMAL	The procedure is successfully completed.
PAMSLU62_BADINDEX	The *connection_index* provided is invalid.
PAMSLU62_CONABORTDATA	The connection has been aborted by the port server due to a nontranslatable ASCII character in the body of the message.
PAMSLU62_ CONABORTSTATE	The connection has been aborted by the port server due to a violation of the selected application protocol.
PAMSLU62_NOCONNECT	No connection has been established.
PAMS__*xxxxxxxx*	Indicates any PAMS status code returned by `pams_put_msg`.

Example The following is an example of the PORT_SEND procedure call.

```
#include stdio
#include signal
#include "port_fun.h"
#include "p_entry.h"

#define TRUE 1
#define FALSE 0

main()
{
      int32 p_status;
      long status;
      short send_connection;
      .
      .
      .

      status = port_send(argv[2],
      send_connection,
      TRUE,           /* change direction     */
      FALSE,          /* do not FLUSH         */
      FALSE,          /* do not disconnect    */
      FALSE,          /* do not abort         */
       3,
       63);

      if (!(status & 1))
 {
 p_status = pams_exit();
 return(p_status);
 }

    .
    .
    .

}
```

CHAPTER

6 LU6.2 User Callback Services

This chapter introduces the LU6.2 User Callback Services (UCB) and contains detailed descriptions of all LU6.2 User Callback APPC messages alphabetized by message type. Each description lists the message type code name, the operating environment in which the message is available for use, and a detailed explanation of how to define the message area and supply required arguments to send messages using the BEA MessageQ API or scripts.

Specifically, this chapter addresses the following topics:

- LU6.2 User Callback Overview
- Using the LU6.2 User Callback Interface
- APPC User Callback Messages

LU6.2 User Callback Overview

The simplest way to establish and maintain a connection between SNA and CICS applications is through the LU6.2 Port Server. The Port Server uses predefined messages to help you set up and manage the application connections between BEA MessageQ clients and remote partners.

If the standard LU6.2 port server programming interface does not meet all application needs, a specialized port server can be developed using the LU6.2 User Callback Services.

The LU6.2 User Callback allows you to engage in APPC conversations between any OpenVMS application program and one or more CICS transaction programs using BEA MessageQ `pams_put_msg` and `pams_get_msg` callable services. The remote CICS transaction program appears to the OpenVMS program as a source and recipient of BEA MessageQ messages.

Note: Unlike applications that use the LU6.2 Port Server, all applications using the LU6.2 User Callback Services must reside on an OpenVMS platform.

The LU6.2 User Callback uses a set of predefined messages to define and delete LUs, establish LU6.2 conversations, send and receive data, request and send confirmations, and process errors.

The APPC verb set consists of 21 BEA MessageQ messages. The verb flow logic is the same, regardless of whether you are programming in an OpenVMS or IBM environment.

Some of the messages are both sent by the `pams_put_msg` call and received by the `pams_get_msg` call.

Each message sent or received by the LU6.2 User Callback is prefixed by a header. Some user callback messages contain no fields other than the header fields. In this case, the type of message is sufficient to cause the desired action.

Conversations can be initiated either by an OpenVMS program, which is called *inbound conversation allocation*; or by a CICS transaction program, which is called *outbound conversation allocation*. Each user program can have a maximum of 256 conversations active at any time.

Note: To use the LU6.2 User Callback interface, the OpenVMS programmer should have general knowledge of APPC and the CICS programmer should have specific knowledge of CICS APPC. Because the OpenVMS programmer uses the familiar BEA MessageQ interface to perform the APPC functions, no knowledge of the DECnet/SNA OpenVMS APPC/LU6.2 Programming Interface is required.

The tool kit provided with the BEA MessageQ LU6.2 Services contains the object library (`DMQLU62_LIB.OLB`), `PAMSLU62_MSG` message structures, and the `DMQLU62_TEST.C` example program.

The following messages are issued by the LU6.2 User Callback:

- `LU62_ACTIVATE`

- `LU62_ALLOCATE`
- `LU62_CONFIRMED`
- `LU62_CONFIRM_RECV`
- `LU62_CONFIRM_REQ`
- `LU62_CONFIRM_SEND`
- `LU62_CONNECTED`
- `LU62_DEALLOCATE`
- `LU62_DEALLOCATED`
- `LU62_DEFINE_LU`
- `LU62_DEFINE_TP`
- `LU62_DELETE_LU`
- `LU62_ERROR`
- `LU62_INIT`
- `LU62_OK_TO_SEND`
- `LU62_RECV_DATA`
- `LU62_REQ_CONFIRM`
- `LU62_REQUEST_TO_SEND`
- `LU62_SEND_CONFIRM`
- `LU62_SEND_DATA`
- `LU62_SEND_ERROR`

Using the LU6.2 User Callback Interface

The LU6.2 User Callback interface is initialized by sending an `LU62_INIT` message. Remote LUs are defined by sending one or more `LU62_DEFINE_LU` messages. The User Callback returns the `LU62_DEFINE_LU` message if the define operation is successful. It returns an `LU62_ERROR` message if the define operation is not successful. A data field in the `LU62_DEFINE_LU` message indicates whether the LU is to be used for inbound or outbound conversations.

Multithreading Services

BEA MessageQ LU6.2 supports the development of multithread servers (for example, the LU6.2 port server). Multithreading is based on the use of context information in the special LU6.2 message header.

The `LU62_REQUESTER` and `LU62_CONV_ID` fields in the LU6.2 message header allow a process to handle multiple concurrent LU6.2 conversations. The `LU62_CONV_ID` uniquely identifies each active conversation. The `LU62_REQUESTER` identifies the originator of an `LU62_DEFINE_LU`, `LU62_DEFINE_TP`, `LU62_ALLOCATE`, and `LU62_ACTIVATE`. If a session terminates abnormally while a conversation is not active, the `LU62_REQUESTER` value is returned in the `LU62_ERROR` message.

Inbound Conversations

An inbound (to CICS) conversation is requested using the `LU62_ALLOCATE` message. The User Callback returns one of the following messages:

- An `LU62_ERROR` message, if the `LU62_ALLOCATE` message is not successful.
- An `LU62_ALLOCATE` message, if the `LU62_ALLOCATE` message is successful.

  The `LU62_ALLOCATE` message returns a unique conversation ID for this conversation in the `LU62_CONV_ID` field. This value is returned by the User Callback for each conversation allocated. The OpenVMS programmer must keep track of the conversation ID (`LU62_CONV_ID`) for each active conversation.

After a successful `LU62_ALLOCATE` message, the remaining message types can be used to conduct the conversation.

Outbound Conversations

An outbound conversation is requested by CICS transaction programs. For a CICS transaction program to allocate a conversation with an OpenVMS transaction program, the OpenVMS transaction program must perform the following operations:

- Send an `LU62_DEFINE_LU` message to define each remote LU that supports an outbound conversation

- Send an `LU62_DEFINE_TP` message to define each TPN that is requested by a remote CICS transaction program
- Send an `LU62_ACTIVATE` message to explicitly activate an SNA session for each LU that supports an outbound conversation

When a remote CICS transaction program allocates a conversation with one of the TPNs, the OpenVMS transaction program receives an `LU62_CONNECTED` message. The local LU name of the LU that received the connection is returned in the `LU62_CONNECTED_LU_NAME` field.

The `LU62_CONNECTED` message also returns a unique conversation ID for this conversation in the `LU62_CONV_ID` field. This value is returned by the User Callback for each conversation allocated. The OpenVMS programmer must keep track of the `LU62_CONV_ID` for each active conversation.

Following receipt of an `LU62_CONNECTED` message, the remaining message types can be used to conduct the conversation.

Example of User Callback Message Flow

Table 6-1 shows a typical message exchange between a BEA MessageQ client and the LU6.2 User Callback. The "Messages Sent to User Callback:" column lists the messages that a BEA MessageQ client sends to the User Callback. The "Messages Received from User Callback:" column lists the messages that the User Callback sends back to the BEA MessageQ client in response to messages received from the client.

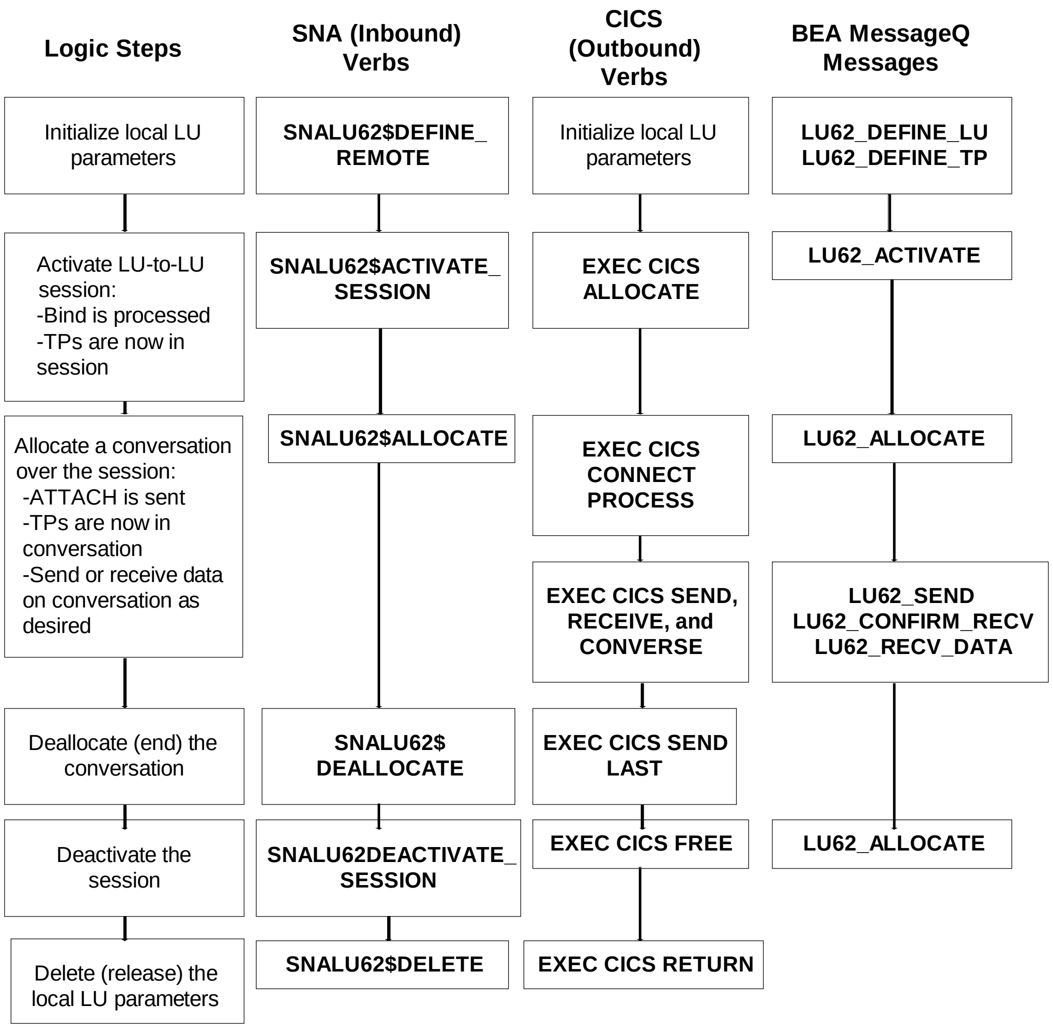
Table 6-1 BEA MessageQ Client—User Callback Message Exchange

Messages Sent to User Callback	Messages Received from User Callback
`LU62_INIT`	
`LU62_DEFINE_LU`	
	`LU62_DEFINE_LU`
`LU62_ALLOCATE`	
	`LU62_ALLOCATE`
`LU62_SEND_DATA` . .	
`LU62_SEND_DATA`	
`LU62_CONFIRM_RECV`	
	`LU62_CONFIRMED`
	`LU62_RECV_DATA` . .
	`LU62_RECV_DATA`
	`LU62_CONFIRM_REQ`
`LU62_DEALLOCATE`	
	`LU62_DEALLOCATED`
—DONE—	

APPC User Callback Messages

The following sections describe APPC User Callback messages (verbs) individually. As an example, Figure 6-1 shows the session logic and corresponding OpenVMS (SNA), IBM (CICS), and BEA MessageQ messages (verbs) used in designing a distributed transaction.

Figure 6-1 BEA MessageQ LU6.2 Session—Typical Verb Sequence


Logic Steps
SNA (Inbound) Verbs
CICS (Outbound) Verbs
BEA MessageQ Messages
Initialize local LU parameters
SNALU62$DEFINE_ REMOTE
Initialize local LU parameters
LU62_DEFINE_LU LU62_DEFINE_TP
Activate LU-to-LU session: -Bind is processed -TPs are now in session
SNALU62$ACTIVATE_ SESSION
EXEC CICS ALLOCATE
LU62_ACTIVATE
Allocate a conversation over the session: -ATTACH is sent -TPs are now in conversation -Send or receive data on conversation as desired
SNALU62$ALLOCATE
EXEC CICS CONNECT PROCESS
LU62_ALLOCATE
EXEC CICS SEND, RECEIVE, and CONVERSE
LU62_SEND LU62_CONFIRM_RECV LU62_RECV_DATA
Deallocate (end) the conversation
SNALU62$ DEALLOCATE
EXEC CICS SEND LAST
Deactivate the session
SNALU62DEACTIVATE_ SESSION
EXEC CICS FREE
LU62_ALLOCATE
Delete (release) the local LU parameters
SNALU62$DELETE
EXEC CICS RETURN

LU62_ACTIVATE

When sent to the CICS (IBM) partner, the `LU62_ACTIVATE` message explicitly activates an SNA session on the LU indicated in the `LU62_ACTIVATE_LOCAL_LU` field.

Note: The behavior of the `LU62_ACTIVATE` message depends on the value of the `LU62_DEFINE_INIT_TYPE` field provided when the LU specified in the message was defined.

For LUs defined with `LU62_DEFINE_INIT_TYPE` = `1`, the `LU62_ACTIVATE` message enables the LU for outbound (from CICS) session activation. The LU is reserved at the gateway, but an SNA session is not started until a transaction program from the remote system requests one.

For all other values of `LU62_DEFINE_INIT_TYPE`, the `LU62_ACTIVATE` message causes the creation of an SNA session on the specified LU. After a session is activated, it is available for use by either partner.

When received from the CICS (IBM) partner, the `LU62_ACTIVATE` message indicates that a previous request to activate a session has been completed by the interface.

C Message Structure

```
struct lu62_activate_struct {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
char lu62_activate_local_lu [8];
char lu62_activate_polarity;
} ;
```

Message Data Fields

Header Fields		
Field	**Data Type**	**Description**

`LU62_REQUESTER`	longword	Identifies the originator of an `LU62_ACTIVATE` request. This field is returned to the user with the status of the request. A copy of the request is returned to the user if the request is successful. The `LU62_ERROR` message is returned to the user if the request is unsuccessful. This value is ignored by the User Callback Interface. This field is intended for use in building applications that handle multiple concurrent conversations. In the event of an abnormal session termination while a conversation is not active, this value is returned on the `LU62_ERROR` message that contains the `PAMSLU62_SESSFAILED` error code (described in Appendix A, "LU6.2 User Callback Interface Logical Names and Error Codes.").

Message Fields

Field	**Data Type**	**Description**
`LU62_ACTIVATE_LOCAL_LU`	text 8 char	The LU name identifying the session to be activated. The `LU62_DEFINE_SESSION` field must have been explicitly defined in the `LU62_DEFINE_LU` message that defined the LU name.
`LU62_ACTIVATE_POLARITY`	byte	Indicates the contention status for the conversation. Valid values are: 0—OpenVMS application is the winner in a contention situation 1—OpenVMS application must bid for access in a contention situation **Note:** If a session is activated with the `LU62_ACTIVATE` message, subsequent `LU62_ALLOCATE` messages must specify the same value for polarity. In other words, the values of `LU62_ACTIVATE_POLARITY` and `LU62_ALLOCATE_POLARITY` must be the same.

Arguments

Argument	pams_put_msg Format	pams_get_msg Format
Target	UCB	Client
Source	Client	UCB
Class	`MSG_CLASS_APPC`	`UCB_CLASS_APPC`
Type	`MSG_TYPE_LU62_ACTIVATE`	`MSG_TYPE_LU62_ACTIVATE`

LU62_ALLOCATE

When sent to the CICS (IBM) partner, the LU62_ALLOCATE message requests an APPC conversation with the CICS transaction program. The conversation uses LUs defined in the 8-byte LU62_ALLOCATE_LOCAL_LU field and in the 8-byte LU62_TPN field, respectively.

The TPN must be in EBCDIC data format. To translate to this format, use LIB$ASC_TO_EBC.

The TPN must be accessible through the specified LU, which means that the local LU name must already be successfully defined using the LU62_DEFINE_LU message.

When received from the CICS (IBM) partner, the LU62_ALLOCATE message indicates that a previous request to allocate conversation completed successfully. The LU62_CONV_ID field contains the unique conversation ID used to manage this conversation. The LU62_REQUESTER field returns whatever value was placed there in the LU62_ALLOCATE message sent to the User Callback. This provides a way to manage multiple, concurrent conversations.

C Message Structure

```
struct lu62_alloc {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
char lu62_allocate_local_lu [8];
char lu62_allocate_username [10];
char lu62_allocate_password [10];
char lu62_allocate_profile [10];
char lu62_allocate_sync_level;
char lu62_allocate_polarity;
} ;
```

Message Data Fields

Header Fields

Field	Data Type	Description

LU62_REQUESTER	longword	Identifies the originator of an LU62_ALLOCATE request. This field is returned to the user with the status of the request. A copy of the request is returned to the user if the request is successful. The LU62_ERROR message is returned to the user if the request is unsuccessful. This value is ignored by the User Callback interface. This field is intended for use in building applications that handle multiple concurrent conversations. In the event of an abnormal session termination while a conversation is not active, this value is returned on the LU62_ERROR message that contains the PAMSLU62_SESSFAILED error code. (Error codes are described in Appendix A, “LU6.2 User Callback Interface Logical Names and Error Codes.”)
LU62_TPN	text 8	Identifies the TPN with whom a user wants a conversation. This value must be in EBCDIC data format.

Message Fields

Field	Data Type	Description
LU62_ALLOCATE_LOCAL_LU	text 8	The LU name identifying the session to be used for the requested conversation.
LU62_ALLOCATE_USERNAME	text 10	The ASCII value of the user ID to be presented to the remote application for authorization.
LU62_ALLOCATE_PASSWORD	text 10	The ASCII value of the password to be presented to the remote application for authorization.
LU62_ALLOCATE_PROFILE	text 10	The ASCII value of the profile to be presented to the remote application for authorization.
LU62_ALLOCATE_SYNC_LEVEL	byte	Indicates the permitted sync-level on the conversation if DMQLU62$SELECT_SYNC is defined. Valid values are: 0—SYNC_LEVEL=NONE 1—SYNC_LEVEL=CONFIRM

`LU62_ALLOCATE_POLARITY`	byte	Indicates the status for the conversation. Valid values are: 0—OpenVMS application is the winner in a contention situation 1—OpenVMS application must bid for access in a contention situation **Note:** If a session is activated with the `LU62_ACTIVATE` message, then subsequent `LU62_ALLOCATE` messages must specify the same value for polarity. In other words, the value of `LU62_ALLOCATE_POLARITY` and `LU62_ACTIVATE_POLARITY` must be the same.

Arguments

Argument	**pams_put_msg Format**	**pams_get_msg Format**
Target	UCB	Client
Source	Client	UCB
Class	`MSG_CLAS_APPC`	`UCB_CLAS_APPC`
Type	`MSG_TYPE_LU62_ALLOCATE`	`MSG_TYPE_LU62_ALLOCATE`

LU62_CONFIRMED

The `LU62_CONFIRMED` message indicates that the remote partner on the conversation specified in the `LU62_CONV_ID` field has issued a `CONFIRM` in response to a `CONFIRM` request from the OpenVMS program. The `LU62_CONFIRMED` message has the header field only.

C Message Structure

```
struct lu62_confirmed_struct {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
       } ;
```

Message Data Fields

Header Fields

Field	Data Type	Description
`LU62_CONV_ID`	longword	Identifies the conversation involved in the request.

Arguments

None

LU62_CONFIRM_RECV

The LU62_CONFIRM_RECV message sends a PREPARE_TO_RECEIVE, TYPE=SYNC_LEVEL indicator on the conversation specified in the message.

This message is typically used to reverse the direction of the conversation. On successful issue of this message and receipt of an LU62_CONFIRMED message from the User Callback, an OpenVMS program can receive data from the CICS transaction program.

Note: The behavior of this message is affected by disabling SYNC_LEVEL=CONFIRM with the logical DMQLU62$DISABLE_CONFIRM. The logical must be set before program activation.

If SYNC_LEVEL=CONFIRM is disabled, then the LU62_CONFIRMED message indicates that the OpenVMS program is now in a receive state.

C Message Structure

```
struct lu62_confirm_recv_struct {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
       } ;
```

Message Data Fields

Header Fields

Field	Data Type	Description
LU62_CONV_ID	longword	Identifies the conversation involved in the request.

Arguments None

LU62_CONFIRM_REQ

The LU62_CONFIRM_REQ message indicates that the remote partner on the conversation specified in the LU62_CONV_ID field has issued a CONFIRM. The LU62_CONFIRM_REQ message has the header field only.

C Message Structure

```
struct lu62_confirm_req_struct {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
       } ;
```

Message Data Fields

Header Fields

Field	**Data Type**	**Description**
LU62_CONV_ID	longword	Identifies the conversation involved in the request.

Arguments None

LU62_CONFIRM_SEND

The LU62_CONFIRM_SEND message indicates that the remote partner on the conversation specified in the LU62_CONV_ID field has issued a PREPARE_TO_RECEIVE, TYPE=SYNC_LEVEL, with the current SYNC_LEVEL set to CONFIRM.

The OpenVMS program can now issue a CONFIRM by sending an LU62_CONFIRM message on the specified conversation. Sending the CONFIRM places the conversation in the send state: the OpenVMS user program can then send data (using the LU62_SEND_DATA message) on this conversation.

C Message Structure

```
struct lu62_send_confirm_struct {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
       } ;
```

Message Data Fields

Header Fields

Field	**Data Type**	**Description**
LU62_CONV_ID	longword	Identifies the conversation involved in the request.

Arguments

None

LU62_CONNECTED

When received, the `LU62_CONNECTED` message indicates that a remote LU6.2 partner has allocated a conversation with one of the TPNs defined with a previously issued `LU62_DEFINE_TP` message. The `LU62_REQUESTER` field in the header of the `LU62_CONNECT` message is provided in the `LU62_DEFINE_TP` message for the TPN that was attached. The `LU62_CONNECT_LOCAL_LU` field contains the local LU name for the LU that has received the connection.

C Message Structure

```
struct lu62_connected_struct {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
              char lu62_connected_lu_name [8];
       } ;
```

Message Data Fields

Header Fields		
Field	**Data Type**	**Description**
`LU62_CONV_ID`	longword	Identifies the conversation involved in the request.
Message Fields		
Field	**Data Type**	**Description**
`LU62_CONNECTED_LU_NAME`	text 8	The ASCII value of the name of the LU that received the connection.

Arguments

Argument	pams_put_msg Format	pams_get_msg Format
Target	NA	Client
Source	NA	UCB
Class	NA	APPC
Type	NA	`MSG_TYPE_LU62_CONNECTED`

LU62_DEALLOCATE

The LU62_DEALLOCATE message sends a DEALLOCATE to a remote partner in the conversation.

C Message Structure

```
struct lu62_deallocate_struct {
      struct  {
            int32 lu62_requester;
            int32 lu62_conv_id;
            char lu62_tpn [8];
            int16 lu62_msg_len;
            } lu62_header_struct;
            int16 lu62_abend_flag;
      } ;
```

Message Data Fields

Header Fields		
Field	**Data Type**	**Description**
LU62_CONV_ID	longword	Identifies the conversation involved in the request.
Message Fields		
Field	**Data Type**	**Description**
LU62_ABEND_FLAG	word	If LU62_ABEND_FLAG= -1, this message sends a DEALLOCATE, TYPE=ABEND_PROG on the conversation specified in the LU62_CONV_ID field in the message. This causes the conversation to terminate abnormally. All other values for LU62_ABEND_FLAG cause this message to send a DEALLOCATE, TYPE=SYNC_LEVEL.

Arguments

Argument	**pams_put_msg Format**	**pams_get_msg Format**
Target	Client	NA
Source	UCB	NA
Class	APPC	NA
Type	MSG_TYPE_LU62_DEALLOCATED	NA

LU62_DEALLOCATED

The `LU62_DEALLOCATED` message indicates that the remote partner on the conversation specified in the `LU62_CONV_ID` field has deallocated normally. The `LU62_DEALLOCATED` message has the header field only.

C Message Structure

```
struct lu62_deallocated_struct {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
       } ;
```

Message Data Fields

Header Fields		
Field	**Data Type**	**Description**
`LU62_CONV_ID`	longword	Identifies the conversation involved in the request.

Arguments None.

See Also `LU62_DEALLOCATE`

LU62_DEFINE_LU

When sent to the CICS (IBM) partner, the LU62_DEFINE_LU message defines a remote SNA resource. Resources must be defined using the LU62_DEFINE_LU message *before* they can be used to allocate conversations.

When received from the CICS (IBM) partner, the LU62_DEFINE_LU message indicates that a previous request to define the LU, specified in the 8-byte LU62_DEFINE_LOCAL_LU field, completed successfully.

C Message Structure

```
struct lu62_define_struct {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
       char lu62_define_local_lu [8];
       char lu62_define_lu_password [8];
       char lu62_define_gateway [6];
       char lu62_define_accname [8];
       char lu62_define_circuit [5];
       int16 lu62_define_session;
       char lu62_define_applid [8];
       char lu62_define_logmode [8];
       char lu62_define_user_data [128];
       int16 lu62_define_init_type;
       } ;
```

Message Data Fields

Header Fields		
Field	**Data Type**	**Description**

`LU62_REQUESTER`	longword	This field identifies the originator of an `LU62_DEFINE_LU` request. The field is returned to the user with the status of the request. A copy of the request is returned to the user if the request is successful. The `LU62_ERROR` message is returned to the user if the request is unsuccessful. This value is ignored by the User Callback Interface. This field is intended for use in building applications that handle multiple concurrent conversations. In the event of an abnormal session termination while a conversation is not active, this value is returned on the `LU62_ERROR` message that contains the `PAMSLU62_SESSFAILED` error code. (Error codes are described in Appendix A, "LU6.2 User Callback Interface Logical Names and Error Codes.")
`LU62_CONV_ID`	longword	This field identifies the conversation involved in the request.

Message Fields

Field	**Data Type**	**Description**
`LU62_DEFINE_LOCAL_LU`	text 8	The ASCII value of the name specified by the user for the specified remote LU.
`LU62_DEFINE_LU_PASSWORD`	text 8	The ASCII value of the password for the remote LU.
`LU62_DEFINE_GATEWAY`	text 6	The ASCII value of the DECnet/SNA gateway through which a specified LU is to be accessed.

`LU62_DEFINE_ACCNAME`	text 8	The ASCII value of the specified gateway that defines the remote LU to be accessed. If this field is specified, the following fields can be optional:
		`LU62_DEFINE_CIRCUIT`
		`LU62_DEFINE_SESSION`
		`LU62_DEFINE_APPLID`
		`LU62_DEFINE_LOGMODE`
		`LU62_DEFINE_USER_DATA`
		The values specified in these fields are site-specific and must be obtained from the person responsible for DECnet/SNA gateway administration at the user site.

`LU62_DEFINE_CIRCUIT`	text 5	The ASCII value of the circuit on the specified gateway that provides the physical connection over which the remote LU is to be accessed.

`LU62_DEFINE_SESSION`	word	The DECnet/SNA gateway session address number (1-255) when accessing the specified remote LU. This field is required for all LUs that are explicitly activated by an `LU62_ACTIVATE` message.
`LU62_DEFINE_APPLID`	text 8	The ASCII value of the VTAM application that owns the specified remote LU.
`LU62_DEFINE_LOGMODE`	text 8	The ASCII value of the VTAM LOGON MODE table entry that accesses the specified remote LU.
`LU62_DEFINE_USER_DATA`	text 128	Up to 128 bytes of variable data to be passed when accessing the remote LU. This data is not interpreted by the User Callback; it must be presented in the format expected by the remote application.
`LU62_DEFINE_INIT_TYPE`	word	A short integer indicating whether the LU is used for inbound or outbound session activation. Valid values are: 0—INBOUND (to CICS) 1—OUTBOUND (from CICS)

Arguments

Argument	**pams_put_msg Format**	**pams_get_msg Format**
Target	UCB	Client
Source	Client	UCB
Class	`MSG_CLAS_APPC`	`UCB_CLAS_APPC`
Type	`MSG_TYPE_LU62_DEFINE_LU`	`MSG_TYPE_LU62_DEFINE_LU`

LU62_DEFINE_TP

When sent to the CICS (IBM) partner, the `LU62_DEFINE_TP` message defines a TPN for use in accepting outbound conversation allocation requests from CICS transaction programs. The value entered in the `LU62_REQUESTER` field is returned in the `LU62_CONNECTED` message (in the same field) when a remote CICS transaction program is allocated using this TPN. This provides a way to distinguish between allocated TPNs when multiple TPNs are defined.

When received from the CICS (IBM) partner, the `LU62_DEFINE_TP` message indicates that a previous request to define the TPN has completed successfully.

C Message Structure

```
struct lu62_define_tp_struct {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
       char lu62_define_tp_tpn [8];
       } ;
```

Message Data Fields

Header Fields

Field	**Data Type**	**Description**
`LU62_REQUESTER`	longword	This field identifies the originator of an `LU62_DEFINE_TP` request. The field is returned to the user with the status of the request. If the request is successful, the user receives a copy of the request. If the request is unsuccessful, the `LU62_ERROR` message is returned to the user. This value is ignored by the User Callback interface. This field is intended for use in building applications that handle multiple concurrent conversations. If the event is terminated abnormally while a conversation is not active, this value is returned in the `LU62_ERROR` message that contains the `PAMSLU62_SESSFAILED` error code. (Error codes are described in Appendix A, “LU6.2 User Callback Interface Logical Names and Error Codes.”)

Message Fields

Field	**Data Type**	**Description**
`LU62_DEFINE_TP_TPN`	text 8	The TPN

Arguments

Argument	pams_put_msg Format	pams_get_msg Format
Target	UCB	Client
Source	Client	UCB
Class	`MSG_CLAS_APPC`	`UCB_CLAS_APPC`
Type	`MSG_TYPE_LU62_DEFINE_TP`	`MSG_TYPE_LU62_DEFINE_TP`

LU62_DELETE_LU

When sent to the CICS (IBM) partner, the `LU62_DELETE_LU` message deletes the LU identified in the 8-byte `LU62_DELETE_LOCAL_LU` field. This message terminates any SNA session active on the specified LU and unbinds the SNA session, freeing up the associated gateway.

If the message is successful, the `LU62_DELETE_LU` message is returned. If the message is unsuccessful, the `LU62_ERROR` message is returned.

It is not necessary to send an `LU62_DELETE_LU` message unless you want to explicitly unbind the SNA session. A previously established session is available to the user program for reuse in establishing conversations with the specified LU following a successful `LU62_DEALLOCATED` message.

Note: An `LU62_DELETE_LU` message sent while a conversation is active on the LU specified for deletion causes the DECnet/SNA APPC/LU6.2 interface to enter a Wait state until the remote transaction program deallocates or unbinds the SNA session. This blocks the user process until the `LU62_DELETE_LU` operation is complete.

When received from the CICS (IBM) partner, the `LU62_DELETE_LU` message indicates that the specified LU has been successfully deleted.

C Message Structure

```
struct lu62_delete_struct {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
       char lu62_delete_local_lu [8];
       } ;
```

Message Data Fields

Header Fields		
Field	**Data Type**	**Description**
`LU62_CONV_ID`	longword	Identifies the conversation involved in the request.
Message Fields		
Field	**Data Type**	**Description**

`LU62_DELETE_LOCAL_LU`	text 8	The ASCII value of the name specified by the user for the specified remote LU.

Arguments

Argument	**pams_put_msg Format**	**pams_get_msg Format**
Target	UCB	Client
Source	Client	UCB
Class	`MSG_CLAS_APPC`	`UCB_CLAS_APPC`
Type	`MSG_TYPE_LU62_DELETE_LU`	`MSG_TYPE_LU62_DELETE_LU`

LU62_ERROR

The LU62_ERROR message indicates that the remote partner on the conversation specified in the LU62_CONV_ID field has signaled an error or that the User CallbackUser Callback encountered a fatal error on the specified conversation and deallocated. The LU62_ERROR field contains the returned status value. The exact circumstances can be determined from the LU62_ERROR_CODE field values.

C Message Structure

```
struct lu62_error_struct {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
       int32 lu62_error_code;
       int32 lu62_error_vector [16];
       } ;
```

Message Data Fields

Header Fields		
Field	**Data Type**	**Description**
`LU62_CONV_ID`	longword	Identifies the conversation involved in the request.

Message Fields		
Field	**Data Type**	**Description**
`LU62_ERROR_CODE`	longword	Contains the primary error received from the User Callback on an `LU62_ERROR` message. The error codes are: ♦ `SNALU62$_PRERTR`, `SNALU62$_PRERNTR`, and `SNALU62$_PRERPU` Each of these values indicates that the remote process has signaled an error. The receipt of these `LU62_ERROR_CODE` values changes the current state to `RECEIVE`. ♦ `PAMSLU62_TRUNCATED` This message indicates that the buffer is too small to contain the received message. This value does not change the current state. All other errors are treated as fatal by the User Callback and result in the immediate deallocation of the conversation on which the error was received.
`LU62_ERROR_VECTOR`	longword	Contains the secondary error information, if any. This is a 16-element longword array. The `LU62_ERROR_VECTOR` array contains the error vector returned (if applicable). This can be processed by the user or displayed directly with the `SYS$PUTMSG` system service.

Arguments

Argument	**pams_put_msg Format**	**pams_get_msg Format**
Target	NA	Client
Source	NA	UCB
Class	NA	`UCB_CLAS_APPC`
Type	NA	`MSG_TYPE_LU62_ALLOCATE`

LU62_INIT

The `LU62_INIT` message initializes the LU6.2 User Callback. Logical names for tracing and buffer allocation are translated when the `LU62_INIT` message is processed. The `LU62_INIT` message has the header field only.

C Message Structure

```
struct lu62_init  {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
       };
```

Message Data Fields

Header Fields

Field	**Data Type**	**Description**
`LU62_CONV_ID`	longword	Identifies the conversation involved in the request.

Arguments None.

LU62_OK_TO_SEND

The `LU62_OK_TO_SEND` message indicates that the remote partner on the conversation specified in the `LU62_CONV_ID` field has entered the receive state in response to a `REQUEST_TO_SEND` from the OpenVMS transaction program. The OpenVMS transaction is now in the send state and can send the data.

C Message Structure

```
struct lu62_ok_to_send_struct {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
       } ;
```

Message Data Fields

Header Fields

Field	Data Type	Description
`LU62_CONV_ID`	longword	Identifies the conversation involved in the request.

Arguments

None

LU62_RECV_DATA

The LU62_RECV_DATA message contains a data block received on the conversation specified in the LU62_CONV_ID field.

C Message Structure

```
struct lu62_recv_data_struct {
      struct  {
            int32 lu62_requester;
            int32 lu62_conv_id;
            char lu62_tpn [8];
            int16 lu62_msg_len;
            } lu62_header_struct;
      char lu62_data_message [31982];
      } ;
```

Message Data Fields

Header Fields		
Field	**Data Type**	**Description**
LU62_CONV_ID	longword	Identifies the conversation involved in the request.
LU62_MSG_LEN	word	Contains the length of the data block that was received.
Message Fields		
Field	**Data Type**	**Description**
LU62_DATA_MESSAGE	text 1-31982	Contains the data block received. The length of the block is contained in the LU62_MSG_LEN field in the message header. The message size is limited to 31,982 bytes, which is 32,000 (the maximum size of User Callback buffers) minus 18 (the size of the header).

Arguments

Argument	pams_put_msg Format	pams_get_msg Format
Target	UCB	Client
Source	Client	UCB
Class	MSG_CLAS_APPC	UCB_CLAS_APPC
Type	MSG_TYPE_LU62_RECV_DATA	MSG_TYPE_LU62_RECV_DATA

LU62_REQ_CONFIRM

The LU62_REQ_CONFIRM message issues a CONFIRM on the conversation specified in the LU62_CONV_ID field contained in the message, provided that SYNC_LEVEL=CONFIRM processing is not disabled.

This message is discarded if SYNC_LEVEL=CONFIRM processing has been disabled using the DMQLU62$DISABLE_CONFIRM logical name.

C Message Structure

```
struct lu62_req_confirm_struct {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
       } ;
```

Message Data Fields

Header Fields

Field	Data Type	Description
LU62_CONV_ID	longword	Identifies the conversation involved in the request.

Arguments

None.

LU62_REQ_TO_SEND

When sent to the CICS (IBM) partner, the LU62_REQ_TO_SEND message issues a REQUEST_TO_SEND on the conversation specified in the LU62_CONV_ID field in the message.

When received from the CICS (IBM) partner, the LU62_REQUEST_TO_SEND message indicates that the remote partner on the conversation has issued a REQUEST_TO_SEND.

C Message Structure

```
struct lu62_req_confirm_struct {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
       } ;
```

Message Data Fields

Header Fields

Field	**Data Type**	**Description**
LU62_CONV_ID	longword	Identifies the conversation involved in the request.

Arguments None

LU62_SEND_CONFIRM

The LU62_SEND_CONFIRM message sends a CONFIRM on the conversation specified in the LU62_CONV_ID field.

C Message Structure

```
struct lu62_send_confirm_struct {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
       } ;
```

Message Data Fields

Header Fields

Field	Data Type	Description
LU62_CONV_ID	longword	Identifies the conversation involved in the request.

Arguments

None

LU62_SEND_DATA

The `LU62_SEND_DATA` message sends the data block contained in the message on the conversation specified in the `LU62_CONV_ID` field.

C Message Structure

```
struct lu62_send_data_struct {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
       char lu62_data_message [31982];
       } ;
```

Message Data Fields

Header Fields		
Field	**Data Type**	**Description**
`LU62_CONV_ID`	longword	Identifies the conversation involved in the request.
`LU62_MSG_LEN`	word	Contains the length of the data block that was sent.
Message Fields		
Field	**Data Type**	**Description**
`LU62_DATA_MESSAGE`	text 1-31982	Contains the data block to send. The message size is limited to 31,982 bytes, which is 32,000 bytes (UCB buffer maximum size) minus 18 bytes (the header).

Arguments

None.

LU62_SEND_ERROR

The LU62_SEND_ERROR message sends a SEND_ERROR on the conversation specified in the message. This notifies the remote program that an error has occurred and places the conversation in a send state.

C Message Structure

```
struct lu62_send_error_struct {
       struct  {
              int32 lu62_requester;
              int32 lu62_conv_id;
              char lu62_tpn [8];
              int16 lu62_msg_len;
              } lu62_header_struct;
       int32 lu62_error_code;
       } ;
```

Message Data Fields

Header Fields

Field	Data Type	Description
LU62_CONV_ID	longword	Identifies the conversation involved in the request.

Message Fields

Field	Data Type	Description
LU62_ERROR_CODE	longword	Contains the primary error received from the User Callback on an LU62_ERROR message.

Arguments

Argument	pams_put_msg Format	pams_get_msg Format
Target	UCB	NA
Source	Client	NA
Class	MSG_CLAS_APPC	NA
Type	MSG_TYPE_LU62_SEND_ERROR	NA

A LU6.2 User Callback Interface Logical Names and Error Codes

This appendix describes the logical names and error codes used in the LU6.2 User Callback Interface.

User Callback Logical Names

Table A-1 describes the logical names that affect the behavior of the LU6.2 User Callback, upon which the LU6.2 Port Server is based.

Table A-1 User Callback Support Logical Names

Use This Logical Name . . .	To . . .
`DMQLU62$BUFFER_SIZE`	Set the maximum size of buffers in the private buffer pool used by BEA MessageQ applications or set the buffer size to the actual expected size of the load. You can set the buffer size equal to the largest user data message plus 18 bytes, the number of bytes required for the buffer header. The minimum value is 100. The maximum value is 32,000 bytes. If this logical name is not defined, the buffer size is set equal to the size of the largest buffers defined in `DMQ$INIT.TXT`.

`DMQLU62$BUFFER_COUNT`	Set the size of the BEA MessageQ LU6.2 Services private buffer pool. Set the number of buffers equal to the number of LUs defined in the `LU_TABLE` configuration file plus 4. The minimum value is 20. The maximum value is 500. If this logical name is not defined, the buffer count is set equal to the number of large buffers defined in `DMQ$INIT.TXT`. The buffer pool must be large enough to hold all messages received from a remote IBM partner in a single burst or chain. For example: If the remote IBM partner sends 100 messages in a response to a query from the BEA MessageQ client, `DMQLU62$BUFFER_COUNT` must be at least 100. If multiple active conversations receive traffic in large bursts, the value of `DMQLU62$BUFFER_COUNT` must be increased accordingly. The BEA MessageQ LU6.2 Services Port Server logs a `PAMSLU62_NOBUFFER` error if `DMQLU62$BUFFER_COUNT` is inadequate. (For a description of `PAMSLU62_NOBUFFER`, see Table A-2.)
`DMQLU62$SELECT_SYNC`	Define this logical name as any arbitrary value that enables selectable `SYNC_LEVEL` processing and allows you to set `SYNC_LEVEL` to 0 or 1 for each target in the `TARGET_TABLE` configuration file. Enabling `DMQLU62$SELECT_SYNC` overrides the disabling of `CONFIRM` by defining `DMQLU62$DISABLE_CONFIRM`. This value is not interpreted or otherwise used by the LU6.2 User Callback.
`DMQLU62$DISABLE_CONFIRM`	Explicitly disable `CONFIRM` processing. Disabling `CONFIRM` processing causes all conversations to operate at `SYNC_LEVEL=NONE`. The `LU62_REQ_CONFIRM` message is ignored by the User Callback if `CONFIRM` processing is disabled. Note that `DMQLU62$DISABLE_CONFIRM` is ignored if `DMQLU62$SELECT_SYNC` is defined. This value is not interpreted or otherwise used by the LU6.2 User Callback.
`DMQLU62$TRACE`	When defined as a valid OpenVMS file specification, provides a trace of LU6.2 User Callback activity. Trace output shows each routine entered and the status returned by each APPC routine.

Linking a User-Written Port Server

To link a user-written port server with the LU6.2 User Callback, include the `DMQLU62_LIB` and `MSG.LIB` libraries in the following order, with the specified linker options files:

```
link /exe:user_prog.exe user_prog.obj,-
      dmqlu62$dir:dmqlu62_message_pointer.obj,-
      dmqlu62$dir:dmqlu62_lib/lib/inc=(dmqlu62_user_callback),-
      dmq$lib:msg/lib,-
      dmq$lib:dmq$olb/opt,-
      dmqlu62$dir:snalu62/opt
```

Error Handling

The LU6.2 User Callback can return an error in two ways:

- As a return status value from a `pams_put_msg` or `pams_get_msg` call
- Through the `LU62_ERROR` message

Table A-2 describes the error codes specific to the User Callback and the method by which they are delivered to the user.

Table A-2 User Callback Error Codes

Error Code	Delivery	Meaning
`PAMSLU62_EXCEEDLUMAX`	Message	The number of active conversations is already at its maximum limit (256); no additional conversations can be allocated.
`PAMSLU62_NOBUFFER`	Status	The User Callback was unable to allocate a buffer from the private buffer pool. If this error occurs, increase the size of the buffer pool by defining the `DMQLU62$BUFFER_COUNT` logical name or, if it is already defined, by increasing the value.
`PAMSLU62_UNEXPECTED`	Message	An unexpected value for `WHAT_RECEIVED` was returned on an `SNALU62$RECEIVE_IMMEDIATE` call. This generally indicates that a problem with the network has resulted in loss or truncation of a data message.
`PAMSLU62_SESSFAILED`	Message	A previously activated session has been disconnected while no conversation was active. The `LU62_REQUESTER` field contains the value passed on the `LU62_ACTIVATE` message that activated the failed session.

PAMSLU62_BADMSGTYPE	Message	A message sent to the User Callback has an invalid message type.
PAMSLU62_NOSUCHCONV	Message	The value in the LU62_CONV_ID field was invalid.
PAMSLU62_TRUNCATED	Message	The previous LU62_RECV_DATA message on the conversation specified in the LU62_CONV_ID field was truncated. This error occurs because the size of the buffers in the private buffer pool is insufficient. Increase the size of the private buffers by defining the DMQLU62$BUFFER_SIZE logical name or, if it is already defined, by increasing the value.

APPENDIX

B Notes on IMS

The LU6.2 User Callback has been tested with the IMS LU6.1 Adapter for LU6.2 applications. The IMS LU6.1 Adapter is a VTAM program that provides bidirectional translation between the LU6.1 protocol used by the IMS Inter-System Communications (ISC) facility and the LU6.2 procotol used by APPC. A number of restrictions apply to the use of APPC verbs with the IMS LU6.1 Adapter. These restrictions are described in the IMS LU6.1 Adapter software documentation.

Users who want to communicate with IMS using the LU6.2 User Callback should take note of the following additional restrictions:

- Synchronization level

  The IMS LU6.1 Adapter does not provide direct support for `SYNC_LEVEL=CONFIRM`. When operating at `SYNC_LEVEL=CONFIRM`, confirmation from the IMS LU6.1 Adapter indicates that the transaction has been accepted by the IMS queue manager, *not* that the transaction has been processed by the target queue.

- Error handling

  The IMS LU6.1 Adapter does not support normal methods of signaling errors through APPC. Programmers developing applications that require this capability must design their applications accordingly.

- Transaction program names

  The IMS LU6.1 Adapter uses a constant TPN (`IMSASYNC`) when attaching remote LU6.2 applications. This means that the TPN cannot be used to distinguish inbound (from the IBM system) conversations from each other. Programmers developing applications that require this capability must design their applications accordingly.

- Session establishment

The IMS LU6.1 Adapter supports establishment of an SNA session on behalf of IMS, but only if the IMS `/OPNDST` command is issued to request the session. Applications can eliminate this requirement by having the LU6.2 User Callback request the session. To do this, set the `LU62_DEFINE_INIT_TYPE` field in the `LU62_DEFINE_LU` message for the LU that is to accept incoming conversation to 0 (zero). The session is then established by sending an `LU62_ACTIVATE` message for the LU. Following activation of the session, the IMS LU6.1 Adapter can use the session.

Note: Prior to activating the session, at least one valid local TPN must be established by sending an `LU62_DEFINE_TP` message to the User Callback. Failure to establish valid TPNs prior to session activation results in allocation failures in the IMS LU6.1 Adapter.

C Examples of BEA MessageQ LU6.2 Inbound and Outbound Applications

The following sections provide sample Inbound and Outbound applications that exchange data with an APPC application in an SNA network. These sample applications are created using the tables described in Chapter 2, “Developing Applications Using BEA MessageQ LU6.2 Services.”

Sample Inbound Application

```
-------------------------------------------------------------------------------
-
/*
** Copyright (c) BEA Systems, Inc., 1999
** All Rights Reserved.
**
** This software is furnished under a license and may be used and copied
** only  in  accordance  with  the  terms  of such  license and with the
** inclusion of the above copyright notice. This software or  any  other
** copies thereof may not be provided or otherwise made available to any
** other person. No title to and ownership of  the  software  is  hereby
** transferred.
```

```
**
** The information in this software is subject to change without  notice
** and  should  not be  construed  as  a commitment by BEA Systems, Inc.
**
**
**    FILE:          inbound.c
**
**    DESCRIPTION:   Illustrates the use of BEA MessageQ LU6.2 Services
**                   to implement an application that activates an
**                   Inbound application and exchanges data with an
**                   APPC application in an SNA network.
**
**    REQUIREMENTS:  The queue named "LU62_SERVER" must
**                   be defined in your init file and must translate
**                   to the group and queue of the DMQ LU6.2 Port Server
**
*/


/**  .........................  **/
/**  BEA MessageQ include files **/
/**  .........................  **/

#include <p_entry.h>
#include <p_return.h>
#include <p_symbol.h>
#include <p_typecl.h>
#include <pamslu62_server_msg.h>

/**  .......................  **/
/**  C library include files  **/
/**  .......................  **/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


/*
** Set a max user message size
*/

#define MAX_USER_MESSAGE_SIZE 4096

/*
** Define values for states: use an enumerated type here
**  to make sure each value is unique
*/
 typedef enum  {
      STATE_UNDEFINED,
     STATE_CONNECTING,
     STATE_WAIT_CONNECT,
     STATE_WAIT_RESPONSE,
     STATE_WAIT_COMPLETE,
     STATE_EXITING,
     STATE_LAST
     } aState;

/*
**  Define a UNION for all messages, with a buffer
*/
    typedef union _lu62_msg {
       /*
       **  PORT_SERVER messages
       */
       register_target       regist;
       connect_request      conreq;
       data_message      data;
       connection_terminated term;
       connect_accept      accept;
       connect_reject      reject;
       change_direction      change;
       /*
       ** buffer area
       */
       char p_buffer[MAX_USER_MESSAGE_SIZE-8];  /*
       ** the 8 byte adjustment allows for the maximum
       ** overhead in any Port Server message */

    } Lu62Msg;

/*
** Routine to attach a queue so the application can send and
** receive BEA MessageQ traffic
*/
int32
AttachQueue(q_address *q_attached)
{
 int32   status;
    int32 attach_mode;
    int32   q_type;

    attach_mode   = PSYM_ATTACH_TEMPORARY;
    q_type        = PSYM_ATTACH_PQ; /* causes the tempory queue to be */
                                    /* a temporary primary queue      */

    status        = pams_attach_q(
                          &attach_mode,
                          q_attached,
                          &q_type,        /*  make a temp primary queue */
                          (char *) 0,     /*  q_name not needed         */
                          (int32 *) 0,    /*  q_name_len not needed     */
                          (int32 *) 0,    /*  Use default name space    */
                          (int32 *) 0,    /*  No name space list len    */
                          (int32 *) 0,    /*  Timeout Value             */
                          (char *)  0,    /*  Reserved by BEA           */
                          (char *)  0 );  /*  Reserved by BEA           */
(online_chunk)
    if ( status == PAMS__SUCCESS )
            printf("Attached successfully to temporary queue %d.\n",
                       q_attached->au.queue);
    else
            printf("Error attaching temporary; status returned is: %ld\n",
                    status );

 return(status);
}


/*
** Routine to locate the DMQ LU6.2 Services Port Server
** based on its' name
*/
int32
LocateServer(q_address *server_q)
{

 int32 status;
    int32 queue_name_len;
    int32 wait_mode;
    int32 req_id;

    /*
    **  Attempt to locate the queue_name in the process and group name spaces
    */
    queue_name_len = strlen("LU62_SERVER");
    wait_mode      = PSYM_WF_RESP;
    req_id         = 1;

    status         = pams_locate_q(
                           "LU62_SERVER",
                           &queue_name_len,
                           server_q,
                           &wait_mode,
                           &req_id,
                           (int32 *) 0,  /* No response queue  */
                           (int32 *) 0,  /* Use default name space list of
                                            process and group  */
                          (int32 *) 0,  /* name space list len not needed */
                          (char *) 0 );



    switch (status )
    {
       case PAMS__SUCCESS :
          printf( "\nLocated queue named: \"%s\" at %d.%d\n", "LU62_SERVER",
                  server_q->au.group, server_q->au.queue );
       break;

       case PAMS__NOOBJECT :
          printf( "\nQueue: \"%s\" not found.\n", "LU62_SERVER" );
       break;

       default :
          printf( "\nUnexpected error returned from pams_locate_q: %ld\n",
                   status );
       break;
    }/*end case */

 return(status);
}

/*
** WaitMsg
*/

int32
WaitMsg (Lu62Msg *msg, short *bytes_rcvd, short *type_rcvd, q_address *from_addr,
short bufsize)
{
 int32 status;

   char        priority=0;
   long        timeout=300; /* wait 30 seconds */
   short       msg_class;

    /*  Get a message  */
 status = pams_get_msgw(
                     (char *)msg,
                     &priority,
                     from_addr,
                     &msg_class,
                     type_rcvd,
                     &bufsize,
                     bytes_rcvd,
                     &timeout,
                     (long *) 0,
                     (struct PSB *) 0,
                     (struct show_buffer *) 0,
                     (long *) 0,
                     (char *) 0,
                     (char *) 0,
                     (char *) 0 );


 switch ( status )
 {
  case PAMS__SUCCESS :
     printf( "\nReceived Message:Class:%d\tType:%d\n",msg_class,*type_rcvd );
  break;

  case PAMS__TIMEOUT :
     printf( "\nTimed out waiting for messages\n" );
  break;

  default :
     printf( "\nError getting message; status returned is %ld.\n",
              status );
  break;


 }/* end case */


 return(status);
}

/*
** Routine to send a message to the remote partner
*/
int32
SendData(Lu62Msg *msg, short msglen, short msgtyp, q_address server_q)
{
  int32 status;

 char        priority;
 char        delivery;
 char        uma;
 short       msg_class;
 long        timeout;
 struct PSB  put_psb;

   priority    = 0;                 /* Regular priority; use 0, NOT '0'     */
   msg_class   = MSG_CLAS_APPC;
   delivery    = PDEL_MODE_WF_MEM;  /* Return bad status if undeliverable   */
   timeout     = 100;              /* Wait 10 seconds before giving up     */
   uma         = PDEL_UMA_DISCL;    /* If can't deliver it, DISCard and Log */


   status = pams_put_msg(
              (char *)msg,
              &priority,
              &server_q,        /* passed in */
              &msg_class,
              &msgtyp,
              &delivery,
              &msglen,
              &timeout,
              &put_psb,
              &uma,
              (q_address *) 0,
              (char *) 0,
              (char *) 0,
              (char *) 0 );

      if (status == PAMS__SUCCESS )
         printf( "Put message type %d\n",msgtyp);
      else
         printf( "Error putting message; status returned is: %ld.\n",
                  status );

 return(status);
}

/*
** Routine to send an abnormal termination message to the
** Port Server
*/
int32
SendAbort(short connection, q_address server_q, int32 reason)
{
 int32 status;
 Lu62Msg term_msg;

 memset(&term_msg,0,sizeof(term_msg.term));
 term_msg.term.connection_index = connection;
 term_msg.term.terminate_type   = DISCONNECT_ERROR;
 term_msg.term.terminate_reason = reason;
 /*
 ** Send the message - set STATE_EXITING unconditionally
 */
 status = SendData(&term_msg,sizeof(term_msg.term),
                 MSG_TYPE_CONNECTION_TERMINATED,server_q);
 return(status);
```

```
}

/*
** Routine to send a connect request message to the
** Port Server
*/

int32
SendConnect(aState *state, char *tp_name, q_address server_q)
{
 int32 status;
 Lu62Msg msg;

 /*
 **  Set up a connect request and send it to the port server.  If the send
 **  is successful, change the state to STATE_WAIT_CONNECT.
 */
 memset(&msg,0,sizeof(msg.conreq));

 strncpy(msg.conreq.target_name,tp_name,sizeof(msg.conreq.target_name));

 status = SendData(&msg,sizeof(msg.conreq),MSG_TYPE_CONNECT_REQUEST, server_q);

 if (status == PAMS__SUCCESS)
  *state = STATE_WAIT_CONNECT;
 else
  *state = STATE_EXITING;


 return(status);
}

/*
** Routine to handle traffic received while we are in the WAIT_CONNECT State
*/
int32
WaitConnect(aState *state, short *connection, Lu62Msg *msg, short msg_type,
q_address server_q)
{
 int32 status;
 Lu62Msg data_msg;

 switch (msg_type)
 {


  case MSG_TYPE_CONNECT_ACCEPT:
   /*
   **  If the message is a connect response, save the connection index, format a
   **  data message, set the change_direction indicator ti CHANGE_DIRECTION, which
   **  will make us the receiver when the Port Server processes the message.
   */
   *connection = msg->accept.connection_index;
   memset(&data_msg.data,0,MAX_USER_MESSAGE_SIZE);
   data_msg.data.connection_index = *connection;
   data_msg.data.change_direction = CHANGE_DIRECTION;
   /*
   ** Put some data in the message body
   */
   strcpy(data_msg.data.data,"R 000666");
   /*
   ** Send the message - if the send works, set the state to STATE_WAIT_RESPOMSE
   */
   status =
SendData(&data_msg,MAX_USER_MESSAGE_SIZE,MSG_TYPE_DATA_MESSAGE,server_q);
   if (status == PAMS__SUCCESS)
    *state = STATE_WAIT_RESPONSE;
   break;
  case MSG_TYPE_CONNECT_REJECT:
   printf("Port Server rejected connect request.\n");
   *state = STATE_EXITING;
   break;
  default:
   printf("WaitConnect: received unexpected message of type %d\n",msg_type);
   status = PAMS__SUCCESS;
   *state = STATE_WAIT_CONNECT;
   break;
 }
 return(status);
}

/*
** Routine to handle traffic received while we are in the WAIT_RESPONSE State
*/
int32
WaitResponse(int32 *state, short connection, short msg_type, q_address server_q)
{
 int32 status;
 switch (msg_type)
 {
  case MSG_TYPE_DATA_MESSAGE:
   printf("WaitResponse: received response message\n");
   status = PAMS__SUCCESS;
   *state = STATE_WAIT_COMPLETE;
   break;


  case MSG_TYPE_CHANGE_DIRECTION:
   /*
   ** The partner program has violated the agreed-upon conversation rules:
   **  disconnect the conversation.  We send a "disconnect reason" of -1; this
  **  does not get passed back beyon d the Port Server but is useful in application
   **  debugging, since we can see what routine is generating the abort message
   **  by providing a unique reason code for each place we abort a conversation.
   */
   printf("WaitResponse: Received unexpected Change Direction message\n");
   status = SendAbort(connection, server_q, -1);
   status = PAMS__SUCCESS-1;
   *state = STATE_EXITING;
   break;

  case MSG_TYPE_CONNECTION_TERMINATED:
   printf("WaitResponse: Port Server has terminated connection\n");
   status = PAMS__SUCCESS-1;
   *state = STATE_EXITING;
   break;

  default:
   printf("WaitResponse: received unexpected message of type %d\n",msg_type);
   status = PAMS__SUCCESS;
   *state = STATE_WAIT_RESPONSE;
   break;
 }
 return(status);
}


int32
WaitComplete(int32 *state, short connection, short msg_type, q_address server_q)
{
 int32 status;
 Lu62Msg term_msg;

 switch (msg_type)
         {
    case MSG_TYPE_CHANGE_DIRECTION:
    printf("WaitComplete: received Change Direction message\n");
    memset(&term_msg,0,sizeof(term_msg.term));
    term_msg.term.connection_index = connection;
    term_msg.term.terminate_type   = DISCONNECT_NORMAL;
           term_msg.term.terminate_reason = 0;
           /*
    ** Send the message - set STATE_EXITING unconditionally
    */
    status = SendData(&term_msg,sizeof(term_msg.term),
    MSG_TYPE_CONNECTION_TERMINATED,server_q);
    *state = STATE_EXITING;
    status = PAMS__SUCCESS-1;   /* force the main loop to exit */
    break;

    case MSG_TYPE_DATA_MESSAGE:
    /*
    **  The partner program has violated the agreed-upon conversation rules:
    **  disconnect the conversation.  We send a "disconnect reason" of -2; this
    **  does not get passed back beyon d the Port Server but is useful in
application
    **  debugging, since we can see what routine is generating the abort message
    **  by providing a unique reason code for each place we abort a conversation.

    */
  printf("WaitComplete: received unexpected data message\n");
  status = SendAbort(connection, server_q, -2);
  status = PAMS__SUCCESS-1;
  *state = STATE_EXITING;
  break;

  case MSG_TYPE_CONNECTION_TERMINATED:
  printf("WaitComplete: Port Server has terminated connection\n");
   status = PAMS__SUCCESS-1;
   *state = STATE_EXITING;
   break;



  default:
   printf("WaitComplete: received unexpected message of type %d\n",msg_type);
   status = PAMS__SUCCESS;
   *state = STATE_WAIT_RESPONSE;
   break;
 }
 return(status);
}


void
main()
{
 int32 status;

 q_address q_attached,
    server_q,
    from_addr;

 aState state=STATE_UNDEFINED;

 Lu62Msg msg;
```

```
/*
** various variables.  "connection" will receive the
**  "connection index" returned to us by the Port Server, which
** we will use to identify which connection we want the port server
** to use when we send data.  On received messages, the port server
** will give us the connection index so we can tell what connection
** the data came from.  This allows a client program to have many
** connections running at the same time.
*/

short connection,
  bytes_rcvd,
  bufsize=sizeof(Lu62Msg),
  type_rcvd;

/*
** Attach a queue for ourselves; if that works, locate the server.
** Exit in the event either operation fails.
*/

status = AttachQueue(&q_attached);


if (status == PAMS__SUCCESS)
 status = LocateServer(&server_q);
 if (status != PAMS__SUCCESS)
  pams_exit();

if (status != PAMS__SUCCESS)
  return;

/*
** Initialize the application by setting the state to CONNECTING
**  and sending the connect request
*/

state = STATE_CONNECTING;
status = SendConnect(&state,"UPDATE",server_q);

while (status == PAMS__SUCCESS)
 {
  status = WaitMsg(&msg, &bytes_rcvd, &type_rcvd, &from_addr, bufsize);
  if (status != PAMS__SUCCESS)
   state = STATE_EXITING;

  switch (state) {

   case STATE_WAIT_CONNECT:
     status = WaitConnect(&state,  &connection, &msg, type_rcvd, server_q);
     break;

    case STATE_WAIT_RESPONSE:
     status = WaitResponse(&state, connection, type_rcvd, server_q);
     break;

    case STATE_WAIT_COMPLETE:
     status = WaitComplete(&state, connection, type_rcvd, server_q);
     break;

    case STATE_EXITING:
     status = PAMS__SUCCESS-1; /* terminate the WHILE */
     break;

    default:
     state = STATE_EXITING;
     break;
   }
  }

 pams_exit();
}
-------------------------------------------------------------------------------
```

Sample Outbound Application

```
-------------------------------------------------------------------------------
-
/*
** Copyright (c) BEA Systems, Inc., 1999
** All Rights Reserved.
**
** This software is furnished under a license and may be used and copied
** only  in  accordance  with  the  terms  of such  license and with the
** inclusion of the above copyright notice. This software or  any  other
** copies thereof may not be provided or otherwise made available to any
** other person. No title to and ownership of  the  software  is  hereby
** transferred.

**
** The information in this software is subject to change without  notice
** and  should  not be  construed  as  a commitment by BEA Systems, Inc.
**
**
**    FILE:          outbound.c
**
**    DESCRIPTION:   Illustrates the use of BEA MessageQ LU6.2 Services
**                   to implement an application that waits for data to
**                   arrive on an outbound (from IBM) conversation, and
**                   exchanges data with the initiating APPC application
**                   in an SNA network.
**
**    REQUIREMENTS:  The queue named "LU62_SERVER" must
**                   be defined in your init file and must translate
**                   to the group and queue of the DMQ LU6.2 Port Server
**
**
*/

/**  .........................  **/
/**  BEA MessageQ include files **/
/**  .........................  **/

#include <p_entry.h>
#include <p_return.h>
#include <p_symbol.h>
#include <p_typecl.h>
#include <pamslu62_server_msg.h>

/**  .......................  **/
/**  C library include files  **/
/**  .......................  **/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


/*
** Set a max user message size
*/

#define MAX_USER_MESSAGE_SIZE 4096

/*
** Define values for states: use an enumerated type here
**  to make sure each value is unique
*/
 typedef enum  {
      STATE_UNDEFINED,
     STATE_REGISTERING,
     STATE_WAIT_REGISTER,
     STATE_WAIT_DATA,
     STATE_WAIT_TO_SEND,
     STATE_WAIT_DISCONNECT,
     STATE_EXITING,
     STATE_LAST
     } aState;
/*
**  Define a UNION for all messages, with a buffer
*/

    typedef union _lu62_msg {
       /*
       **  PORT_SERVER messages
       */
       register_target       regist;
       connect_request      conreq;
       data_message          data;
       connection_terminated term;
       connect_accept      accept;
       connect_reject      reject;
       change_direction      change;
       /*
       ** buffer area
       */
       char p_buffer[MAX_USER_MESSAGE_SIZE-8];  /*
       ** The 8 byte adjustment allows for the maximum overhead in any Port
       ** Server message.
                                 */

    } Lu62Msg;


/*
** Routine to attach a queue so the application can send and
** receive BEA MessageQ traffic
*/
int32
AttachQueue(q_address *q_attached)
{
 int32   status;
    int32 attach_mode;
    int32   q_type;

    attach_mode   = PSYM_ATTACH_TEMPORARY;
    q_type        = PSYM_ATTACH_PQ; /* causes the tempory queue to be */
                                    /* a temporary primary queue      */

    status        = pams_attach_q(
                          &attach_mode,
                          q_attached,
                          &q_type,        /*  make a temp primary queue */
                          (char *) 0,     /*  q_name not needed         */
                          (int32 *) 0,    /*  q_name_len not needed     */
                          (int32 *) 0,    /*  Use default name space    */
                          (int32 *) 0,    /*  No name space list len    */
                          (int32 *) 0,    /*  Timeout Value   */
                          (char *)  0,    /*  Reserved by BEA   */
                          (char *)  0 );  /*  Reserved by BEA          */

    if ( status == PAMS__SUCCESS )
            printf("Attached successfully to temporary queue %d.\n",
                       q_attached->au.queue);
    else
            printf("Error attaching temporary; status returned is: %ld\n",
                    status );

 return(status);
}

/*
** Routine to locate the DMQ LU6.2 Services Port Server
** based on its' name
*/
int32
LocateServer(q_address *server_q)
{
 int32 status;
    int32 queue_name_len;
    int32 wait_mode;
    int32 req_id;

    /*
    **  Attempt to locate the queue_name in the process and group name spaces
    */
    queue_name_len = strlen("LU62_SERVER");
    wait_mode      = PSYM_WF_RESP;
    req_id         = 1;

    status         = pams_locate_q(
                           "LU62_SERVER",
                           &queue_name_len,
                           server_q,
                           &wait_mode,
                           &req_id,
                           (int32 *) 0,  /* No response queue  */
                           (int32 *) 0,  /* Use default name space list of
                                            process and group  */
```

```
                         (int32 *) 0,  /* name space list len not needed */
                         (char *) 0 );

    switch (status )
    {
       case PAMS__SUCCESS :
          printf( "\nLocated queue named: \"%s\" at %d.%d\n", "LU62_SERVER",
                  server_q->au.group, server_q->au.queue );
       break;

       case PAMS__NOOBJECT :
          printf( "\nQueue: \"%s\" not found.\n", "LU62_SERVER" );
       break;

       default :
          printf( "\nUnexpected error returned from pams_locate_q: %ld\n",
                   status );
       break;
    }/*end case */

 return(status);
}

/*
** WaitMsg
*/

int32
WaitMsg (Lu62Msg *msg, short *bytes_rcvd, short *type_rcvd, q_address *from_addr,
short bufsize)
{
 int32 status;

   char        priority=0;
   long        timeout=300; /* wait 30 seconds */
   short       msg_class;

    /*  Get a message  */
 status = pams_get_msgw(
                     (char *)msg,
                     &priority,
                     from_addr,
                     &msg_class,
                     type_rcvd,
                     &bufsize,
                     bytes_rcvd,
                     &timeout,
                     (long *) 0,
                     (struct PSB *) 0,
                     (struct show_buffer *) 0,
                     (long *) 0,
                     (char *) 0,
                     (char *) 0,
                     (char *) 0 );


 switch ( status )
 {
  case PAMS__SUCCESS :
     printf( "\nReceived Message:Class:%d\tType:%d\n",msg_class,*type_rcvd );
  break;

  case PAMS__TIMEOUT :
     printf( "\nTimed out waiting for messages\n" );
  break;

  default :
     printf( "\nError getting message; status returned is %ld.\n",
              status );
  break;

 }/* end case */


 return(status);
}

/*
** Routine to send a message to the remote partner
*/
int32
SendData(Lu62Msg *msg, short msglen, short msgtyp, q_address server_q)
{
  int32 status;

 char        priority;
 char        delivery;
 char        uma;
 short       msg_class;
 long        timeout;
 struct PSB  put_psb;

   priority    = 0;                /* Regular priority; use 0, NOT '0'     */
   msg_class   = MSG_CLAS_APPC;
   delivery    = PDEL_MODE_WF_MEM;  /* Return bad status if undeliverable   */
   timeout     = 100;              /* Wait 10 seconds before giving up     */
   uma         = PDEL_UMA_DISCL;    /* If can't deliver it, DISCard and Log */
   status = pams_put_msg(
              (char *)msg,
              &priority,
              &server_q,        /* passed in */
              &msg_class,
              &msgtyp,
              &delivery,
              &msglen,
              &timeout,
              &put_psb,
              &uma,
              (q_address *) 0,
              (char *) 0,
              (char *) 0,
              (char *) 0 );

      if (status == PAMS__SUCCESS )
         printf( "Put message type %d\n",msgtyp);
      else
         printf( "Error putting message; status returned is: %ld.\n",
                  status );

 return(status);
}

/*
** Routine to send an abnormal termination message to the
** Port Server
*/
int32
SendAbort(short connection, q_address server_q, int32 reason)
{
 int32 status;
 Lu62Msg term_msg;


memset(&term_msg,0,sizeof(term_msg.term));
term_msg.term.connection_index = connection;
term_msg.term.terminate_type   = DISCONNECT_ERROR;
term_msg.term.terminate_reason = reason;
/*
** Send the message - set STATE_EXITING unconditionally
*/
status = SendData(&term_msg,sizeof(term_msg.term),
        MSG_TYPE_CONNECTION_TERMINATED,server_q);
return(status);
}

/*
```

```
** Routine to send a register request message to the
** Port Server
*/

int32
SendRegister(aState *state, char *target, q_address server_q, q_address
my_address)
{
 int32 status;
 Lu62Msg msg;

 /*
 ** Set up a connect request and send it to the port server.  If the send
 **  is successful, change the state to STATE_WAIT_CONNECT.
 */
 memset(&msg,0,sizeof(msg.regist));


 strncpy(msg.conreq.target_name,target,sizeof(msg.regist.target_name));
 msg.regist.target_group   = my_address.au.group;
 msg.regist.target_process = my_address.au.queue;

 status = SendData(&msg,sizeof(msg.regist),MSG_TYPE_REGISTER_TARGET,
        server_q);

 if (status == PAMS__SUCCESS)
  *state = STATE_WAIT_REGISTER;
 else
  *state = STATE_EXITING;

 return(status);
}

/*
** Routine to handle traffic received while we are in the WAIT_REGISTER State
*/
int32
WaitRegister(aState *state, short *connection, Lu62Msg *msg, short msg_type,
q_address server_q)
{
 int32 status;

 switch (msg_type)
 {
  case MSG_TYPE_REGISTER_TARGET:
   /*
      **  Resigtration was accepted - now we wait...
   */
   status = PAMS__SUCCESS;
   *state = STATE_WAIT_DATA;
   break;
  case MSG_TYPE_CONNECTION_TERMINATED:
   printf("Port Server rejected  registration request.\n");
   status = PAMS__SUCCESS-1;
   *state = STATE_EXITING;
   break;
  default:
   printf("WaitRegister: received unexpected message of type %d\n",msg_type);
   status = PAMS__SUCCESS;
   *state = STATE_WAIT_REGISTER;
   break;

 }
 return(status);
}

/*
** Routine to handle traffic received while we are in the WAIT_DATA State
*/
int32
WaitData(int32 *state, short *connection, short msg_type, q_address server_q)
{
 int32 status;
 Lu62Msg msg;

 switch (msg_type)
 {
  case MSG_TYPE_DATA_MESSAGE:
   printf("WaitData: received data message\n");
   /*
   ** save the connection index - we will need to use this later
   */
   *connection = msg.data.connection_index;
   status = PAMS__SUCCESS;
   *state = STATE_WAIT_TO_SEND;
   break;

  case MSG_TYPE_CHANGE_DIRECTION:
   /*
   ** The partner program has violated the agreed-upon conversation rules:
   **  disconnect the conversation.  We send a "disconnect reason" of -1; this
   **  does not get passed back beyon d the Port Server but is useful in application
   **  debugging, since we can see what routine is generating the abort message
   **  by providing a unique reason code for each place we abort a conversation.
   */
   printf("WaitData: Received unexpected Change Direction message\n");
   status = SendAbort(*connection, server_q, -1);
   status = PAMS__SUCCESS-1;
   *state = STATE_EXITING;
   break;

  case MSG_TYPE_CONNECTION_TERMINATED:
   printf("WaitData: Port Server has terminated connection\n");
   status = PAMS__SUCCESS-1;
   *state = STATE_EXITING;
   break;

  default:
   printf("WaitData: received unexpected message of type %d\n",msg_type);
   status = PAMS__SUCCESS;
   *state = STATE_WAIT_DATA;
   break;
 }
 return(status);
}

int32
WaitSend(int32 *state, short connection, short msg_type, q_address server_q)
{
 int32 status;
 Lu62Msg msg;

 switch (msg_type)
 {


  case MSG_TYPE_CHANGE_DIRECTION:
   printf("WaitSend: received Change Direction message\n");
   connection = msg.accept.connection_index;
   memset(&msg.data,0,MAX_USER_MESSAGE_SIZE);
   msg.data.connection_index = connection;
   msg.data.change_direction = CHANGE_DIRECTION;
   /*
   ** Put some data in the message body
   */
   strcpy(msg.data.data,"HELLO");
   /*
   ** Send the message - if the send works, set the state to STATE_WAIT_DISCONNECT
   */
   status = SendData(&msg,MAX_USER_MESSAGE_SIZE,MSG_TYPE_DATA_MESSAGE,server_q);
   if (status == PAMS__SUCCESS)
    *state = STATE_WAIT_DISCONNECT;
   else {
    *state = STATE_EXITING;
    status = PAMS__SUCCESS-1;       /* force the main loop to exit */
   }
   break;
  case MSG_TYPE_DATA_MESSAGE:
   /*
   ** The partner program has violated the agreed-upon conversation rules:
   **  disconnect the conversation.  We send a "disconnect reason" of -2; this
  **  does not get passed back beyon d the Port Server but is useful in application
   **  debugging, since we can see what routine is generating the abort message
   **  by providing a unique reason code for each place we abort a conversation.
   */
   printf("WaitSend: received unexpected data message\n");
   status = SendAbort(connection, server_q, -2);
   status = PAMS__SUCCESS-1;
   *state = STATE_EXITING;
   break;

  case MSG_TYPE_CONNECTION_TERMINATED:
   printf("WaitComplete: Port Server has terminated connection\n");
   status = PAMS__SUCCESS-1;
   *state = STATE_EXITING;
   break;

  default:
   printf("WaitSend: received unexpected message of type %d\n",msg_type);
   status = PAMS__SUCCESS;
   *state = STATE_WAIT_TO_SEND;
   break;
 }
 return(status);
}


int32
WaitDisconnect(int32 *state, short connection, short msg_type, q_address
server_q)
{
 int32 status;

 switch (msg_type)
 {
  case MSG_TYPE_CONNECTION_TERMINATED:
   printf("WaitDisconnect: received Connection Terminated message\n");
   *state = STATE_EXITING;
   status = PAMS__SUCCESS-1;   /* force the main loop to exit */
   break;

  case MSG_TYPE_DATA_MESSAGE:
  /*
  ** The partner program has violated the agreed-upon conversation rules:
  **  disconnect the conversation.  We send a "disconnect reason" of -2; this
  **  does not get passed back beyon d the Port Server but is useful in application
  **  debugging, since we can see what routine is generating the abort message
  **  by providing a unique reason code for each place we abort a conversation.
  */
  printf("WaitDisconnect: received unexpected data message\n");
  status = SendAbort(connection, server_q, -2);
  status = PAMS__SUCCESS-1;
  *state = STATE_EXITING;
  break;

  case MSG_TYPE_CHANGE_DIRECTION:
  /*
  ** The partner program has violated the agreed-upon conversation rules:
  **  disconnect the conversation.  We send a "disconnect reason" of -3; this
  **  does not get passed back beyon d the Port Server but is useful in application
  **  debugging, since we can see what routine is generating the abort message
  **  by providing a unique reason code for each place we abort a conversation.
  */
  printf("WaitDisconnect: received unexpected change direction message\n");
  status = SendAbort(connection, server_q, -3);
  status = PAMS__SUCCESS-1;
  *state = STATE_EXITING;
  break;

  default:
   printf("WaitSend: received unexpected message of type %d\n",msg_type);
   status = PAMS__SUCCESS;
   *state = STATE_WAIT_DISCONNECT;
   break;
 }
 return(status);
}

void
main()
{
 int32 status;

 q_address q_attached,
    server_q,
    from_addr;

 aState state=STATE_UNDEFINED;

 Lu62Msg msg;

 /*
 ** various variables.  "connection" will receive the
 **  "connection index" returned to us by the Port Server, which
 ** we will use to identify which connection we want the port server
 ** to use when we send data.  On received messages, the port server
 ** will give us the connection index so we can tell what connection
 ** the data came from.  This allows a client program to have many
 ** connections running at the same time.
 */

 short connection,
   bytes_rcvd,
   bufsize=sizeof(Lu62Msg),
   type_rcvd;

 /*
 ** Attach a queue for ourselves; if that works, locate the server.
 ** Exit in the event either operation fails.
 */

 status = AttachQueue(&q_attached);

 if (status == PAMS__SUCCESS)
  status = LocateServer(&server_q);
  if (status != PAMS__SUCCESS)
   pams_exit();

 if (status != PAMS__SUCCESS)
   return;

 /*
 ** Initialize the application by setting the state to CONNECTING
 **  and sending the connect request
 */

 state = STATE_REGISTERING;
 status = SendRegister(&state,"NEWORDER",server_q, q_attached);

 while (status == PAMS__SUCCESS)
  {
  status = WaitMsg(&msg, &bytes_rcvd, &type_rcvd, &from_addr, bufsize);
  if (!((status == PAMS__SUCCESS) || (status == PAMS__TIMEOUT)))
  state = STATE_EXITING;

  switch (state) {

   case STATE_WAIT_REGISTER:
    /*
    ** Timeouts are valid in WAIT_DATA, invalid elsewhere.
    */
    if (status == PAMS__SUCCESS)
     status = WaitRegister(&state,  &connection, &msg, type_rcvd, server_q);
    break;

    case STATE_WAIT_DATA:
    /*
    ** If we timed out just go back and wait again
    */
    if (status == PAMS__TIMEOUT)
    status = PAMS__SUCCESS;
    else
     status = WaitData(&state, &connection, type_rcvd, server_q);
    break;

    case STATE_WAIT_TO_SEND:
    /*
    ** Timeouts are valid in WAIT_DATA, invalid elsewhere.
    */
    if (status == PAMS__SUCCESS)
     status = WaitSend(&state, connection, type_rcvd, server_q);
    break;

    case STATE_WAIT_DISCONNECT:
    /*
    ** Timeouts are valid in WAIT_DATA, invalid elsewhere.
    */
    if (status == PAMS__SUCCESS)
     status = WaitDisconnect(&state, connection, type_rcvd, server_q);
    break;

    case STATE_EXITING:
        status = PAMS__SUCCESS-1;
        /* terminate the WHILE */
     break;

    default:
     state = STATE_EXITING;
     break;
   }
  }

 pams_exit();
}

-------------------------------------------------------------------------------
```

D Examples of CICS Inbound and Outbound Applications

The following sections provide samples of CICS Inbound and Outbound applications.

Sample CICS Inbound Application

```
--------------------------------------------------------------------------------

         TITLE ’VAXIN - BACKEND TRANSACTION PROGRAM’                    00010001
*********************************************************************** 00020000
*                                                                     * 00030000
* THIS PROGRAM CAN BE ACTIVATED UNDER THE TRANSACTION ’VXIN’.         * 00040056
*                                                                     * 00080000
*********************************************************************** 00090000
*                                                                       00100000
R15      EQU   15                                                       00110000
R14      EQU   14                                                       00120000
R13      EQU   13                                                       00130000
R12      EQU   12                                                       00140000
R11      EQU   11                                                       00150000
R10      EQU   10                                                       00160000
R9       EQU   9                                                        00170000
R8       EQU   8                                                        00180000
R7       EQU   7                                                        00190000
R6       EQU   6                                                        00200000
R5       EQU   5                                                        00210000
R4       EQU   4                                                        00220000
```

```
R3       EQU   3                                                        00230000
R2       EQU   2                                                        00240000
R1       EQU   1                                                        00250000
R0       EQU   0                                                        00260000
*                                                                       00280000
* FIXED REGISTERS                                                       00290000
*                                                                       00300000
EIBREG   EQU   R9                                                       00310000
*                                                                       00320000
*                                                                       00330000
         PRINT NOGEN                                                    00331096
*                                                                       00333010
VAXIN    DFHEIENT  EIBREG=EIBREG,DATAREG=(13,4)                         00340000
*                                                                       00350000
** MOVE CONSTANTS TO WORKING STORAGE                                    00350000
*                                                                       00350000

         MVC   TRANID,CTRANID
         MVC   SYNLVL,CSYNLVL
         MVC   SYSID,CSYSID
         MVC   DECOUT,CDECOUT
         MVC   TPN,CTPN
         MVC   COMMA1,CCOMMA
         MVC   TERMEQ,CTERMID
         MVC   COMMA2,CCOMMA
         MVC   DATAEQ,CDATA
*
******** EXEC  CICS HANDLE CONDITION ERROR(EXFREE)                      00351098
*                                                                       00352096
         MVC   TERMID,EIBTRMID        SAVE THE PRINCIPLE FACILITY       00362075
*                                     NAME. (TERMINAL ID)               00370096
         MVC   CONVID,EIBTRMID        BACKEND XACTION THIS IS ALSO      00370196
*                                     THE CONVERSATION ID               00370296
** EXTRACT THE CONVERSATION-RELATED INFORMATION FROM THE ATTACH FMH     00370399
*                                                                       00370496
         EXEC  CICS EXTRACT PROCESS                                    +00370579
               PROCNAME  (PROCNAM)                                     +00370679
               PROCLENGTH(PROCLEN)                                     +00370779
               CONVID    (CONVID)                                      +00370896
               SYNCLEVEL (SYNLVL)                                       00370979
*                                                                       00371099
         MVC   TEROUT,TERMID          TERMINAL ID TO HEADER             00371199
         MVC   TPNOUT,PROCNAM         LOCAL TPN TO HEADER               00371299
*                                                                       00371396
** RECEIVE THE MESSAGE FROM THE COOPERATING TPN                         00371496
*                                                                       00371596
POSTREAD DS    0H                                                       00372023
         MVC   INLEN,=H’4096’         SET MAXIMUM RECEIVE LENGTH        00380053
         EXEC  CICS RECEIVE                                            +00390000
               CONVID (CONVID)                                         +00391096
               LENGTH (INLEN)                                          +00400000
               INTO   (DECIN)                                           00410000
```

```
*                                                                       00440000
         LH    R5,INLEN               GET LENGTH OF HEADER MESSAGE      00450096
         LA    R5,MSGLEN(R5)          AND THE RECEIVED MESSAGE          00460000
         STH   R5,OUTLEN              SET AS SEND LENGTH                00470000
*                                                                       00480000
*****************************************************************       00481004
*    TEST CONDITIONS SET IN THE EXEC INTERFACE BLOCK (EIB)              00482096
*****************************************************************       00483004
*                                                                       00483196
EIBTEST  DS    0H                                                       00484004
         MVC   XDFEIFLG,EIBSYNC           SAVE EIB                      00485096
*                                                                       00486157
TESTCONF DS    0H                                                       00486357
         CLI   XCONF,X’FF’                PARTNER WANT A CONFIRM?       00486457
         BNE   TESTSYNC                   NO                            00486599
*                                         YES, ISSUE CONFIRMED]]]]]]    00486657
*                                         SYNC LEVEL (2) PROCESSING     00486799
         EXEC  CICS ISSUE CONFIRMATION                                  00486864
*                                                                       00486957
TESTSYNC DS    0H                    SYNCPOINT IS NOT IMPLEMENTED YET   00487096
         CLI   XSYNC,X’FF’           ON THE VAX SIDE.................   00488096
         BNE   TESTFREE                                                 00489004
         EXEC  CICS SYNCPOINT                                           00489305
*                                                                       00489404
TESTFREE DS    0H                     CEB RECEIVED = ASYNC MESSAGE      00489596
         CLI   XFREE,X’FF’                                              00489604
         BNE   TESTRECV               SYNC MESSAGE GO TEST NEXT SWITCH  00489796
**********************************************************************  00490099
*  ASYNC CONVERSATION MESSAGE RECEIVED (CEB SET)                        00490196
**********************************************************************  00490299
*                                                                       00490399
** NOW SWITCH FROM A BACKEND TRANSACTION TO A FRONT END TRANSACTION     00490496
*                                                                       00490596
*  1. FREE CURRENT CONVERSATION                                         00490699
*  2. ALLOCATE AND CONNECT THE NEW CONVERSATION (OUTBOUND CONVERSATION) 00490799
*  3. SEND THE DATA BACK WITH CEB (ASYNC CONVERSATION)                  00490899
*                                                                       00490958
         EXEC  CICS FREE                                                00491064
         EXEC  CICS ALLOCATE                                           +00491158
                    SYSID (SYSID)                                       00491297
*                       << SYSID IS FROM CICS DEFINITIONS >>            00491397
         MVC   RESOURCE,EIBRSRCE        SAVE THE FRONT END TRANSACTION  00491496
*                                       CONVERSATION ID                 00491596
         EXEC  CICS CONNECT                                            +00491658
               PROCESS                                                 +00491796
               CONVID    (RESOURCE)                                    +00491896
               PROCNAME  (TRANID)                                      +00491996
               PROCLENGTH(TRANLEN)                                     +00492096
               SYNCLEVEL (SYNLVL)                                       00492196

*                                                                       00492296
         EXEC CICS SEND LAST CONFIRM                                   +00492364
```

```
              CONVID(RESOURCE)                                         +00492496
              FROM  (DECOUT)                                           +00492596
              LENGTH(OUTLEN)                                            00492696

*                                                                       00492764
         B     IMMET                   EXIT THE PROGRAM                 00492896
*                                                                       00492958
TESTRECV DS    0H                      RECEIVE STATE AND MULTIPLE       00493099
         CLI   XRECV,X'FF'             LOGICAL RECORDS..........        00493199
         BNE   ENDTEST                 PROCESS THE ONLY RECORD          00493224
*                                                                       00493399
** FIRST SEND THE CURRENT LOGICAL RECORD                                00493499
*                                                                       00493599
         EXEC CICS SEND                                                +00493699
               CONVID(CONVID)                                          +00493899
               FROM  (DECOUT)                                          +00493999
               LENGTH(OUTLEN)                                           00494099
*                                                                       00494199
** NOW READ THE NEXT LOGICAL RECORD                                     00494299
*                                                                       00494399
         B     POSTREAD                                                 00495599
*                                                                       00495623
**  PROCESS THE LOGICAL RECORD FOR BACKEND TRANSACTION                  00495796
*                                                                       00495804
ENDTEST  DS    0H                                                       00495904
         CLC   DECIN(7),=C'$*$TERM'         TERMINATE THE CONVERSATION  00496055
         BE    SENDCEB                      WHEN $*$TERM IS RECEIVED..  00496199
*                                                                       00496221
** SEND THE REPLY                                                       00497000
*                                                                       00500000
* SEND WITH CDI AND REQUEST CONFIRM                                     00500130
*                                                                       00500225
SENDCDI  DS    0H                                                       00501030
         EXEC CICS SEND INVITE CONFIRM                                 +00510039
               CONVID(CONVID)                                          +00511096
               FROM  (DECOUT)                                          +00520000
               LENGTH(OUTLEN)                                           00530025
         B     POSTREAD                   GO WAIT FOR THE NEXT MESSAGE  00540299
*                                                                       00540325
* SEND WITH CEB AND REQUEST CONFIRM (ONLY OR LAST LOGICAL RECORD)       00540439
*                                                                       00540525
SENDCEB  DS    0H                                                       00540630
         EXEC CICS SEND LAST CONFIRM                                   +00540854
               CONVID(CONVID)                                          +00540996
               FROM  (DECOUT)                                          +00541025
               LENGTH(OUTLEN)                                           00541139
*                                                                       00541230
** FREE THE CONVERSATION                                                00541399
** AND                                                                  00541499
** IMMEDIATE RETURN TO CICS                                             00541599
*                                                                       00541699
IMMET    DS   0H                                                        00542043
```

```
         EXEC CICS FREE                                                 00561064
         EXEC CICS RETURN                                               00570041
*                                                                       00580000
*********************************************************************   00581099
*                                                                       00582099
** CONSTANTS - VARIABLES - DATA AREAS                                   00590099
*                                                                       00600000
CTRANID  DC    CL4'IMSA'                                                00601099
TRANLEN  DC    AL2(*-CTRANID)                                           00602079

CSYNLVL  DC    H'1'                                                     00603099
CSYSID   DC    CL4'ST04'                                                00610099
CDECOUT  DC    XL8'00000003010000FF'                                    00621699
CTPN     DC    C'** TPN = '                                             00650299
CCOMMA   DC    C','                                                     00650493
CTERMID  DC    C' TERMID = '                                            00650696
CDATA    DC    C' DATA = '                                              00650993
         LTORG *                                                        00660000

*                                                                       00600000
DFHEISTG DSECT ,
********************************************************************    00651421
*    EIB  EXEC INTERFACE BLOCK STORAGE AREA                             00651521
********************************************************************    00651621
TRANID   DS    CL4                    OUTBOUND TPN                      00601099
SYNLVL   DS    H                      SYNC LEVEL                        00603099
SYSID    DS    CL4                    SYSID FOR LU ON OUTBOUND ALLOCATE 00610099
TERMID   DS    CL4                    EIBTRMID SAVE AREA                00611075
CONVID   DS    CL4                    BACKEND CONVERSATION ID           00612096
RESOURCE DS    CL8                    FRONT END CONVERSATION ID         00620096
INLEN    DS    H                      RECEIVED INPUT MESSAGE LENGTH     00620196
OUTLEN   DS    H                      OUTPUT MESSAGE LENGTH             00620296
PROCLEN  DS    H                      PROCESS NAME LENGTH               00620396
PROCNAM  DS    CL8                    LOCAL TPN (PROCESS NAME)          00621096
PROCFILL DS    CL24                   TEMP FILLER FOR EXTRACT OVERFLOW  00621199
*                                     CICS RETURNS 32 BYTES FOR PROCNAM 00621299
*                                                                       00621399
**  OUTPUT MESSAGE HEADER AND DATA BUFFER                               00621499
*                                                                       00621596
DECOUT   DS    XL8                                                      00621699
TPN      DS    CL9                                                      00650299
TPNOUT   DS    CL4                                                      00650391
COMMA1   DS    CL1                                                      00650493
TERMEQ   DS    CL10                                                     00650696
TEROUT   DS    CL4                                                      00650791
COMMA2   DS    CL1                                                      00650893
DATAEQ   DS    CL8                                                      00650993
MSGLEN   EQU   *-DECOUT                                                 00651148
DECIN    DS    CL4096                 RECEIVED INPUT BUFFER             00651296
*                                     AND OUTPUT DATA BUFFER            00651399
XDFEIFLG DS    0CL10                                                    00651819
XSYNC    DS    C                                                        00652015
```

```
XFREE    DS    C                                                     00653015
XRECV    DS    C                                                     00654015
XSEND    DS    C                                                     00655015
XATT     DS    C                                                     00656015
XEOC     DS    C                                                     00657015
XFMH     DS    C                                                     00658015
XCOMPL   DS    C                                                     00658119
XSIG     DS    C                                                     00658219
XCONF    DS    C                                                     00658319
*                                                                    00651796
         END                                                         00670000
--------------------------------------------------------------------------------
```

Sample CICS Outbound Application

```
--------------------------------------------------------------------------------

         TITLE 'FROMIBM - INIT A TRANSACTION ON THE VAX'                00010001
*********************************************************************** 00020000
*                                                                     * 00030000
*                                                                     * 00031033
* THIS PROGRAM IS DESIGNED TO COMMUNICATE WITH ANOTHER LU6.2 LOGICAL  * 00040000
* UNIT WHICH MAY BE A VMS SYSTEM. IT PERFORMS THE FOLLOWING FUNCTIONS * 00050000
* IN A LOOP WHICH IS REPEATED TEN TIMES :-                            * 00060000
*                                                                     * 00070000
*      1) ALLOCATE A SESSION TO REMOTE SYSTEM 'DC1R'. PL381021        * 00080000
*                                                                     * 00090000
*      2) CONNECT TO PROCESS 'IMSAYNC' AT SYNCHPOINT LEVEL ZERO       * 00100000
*                                                                     * 00110000
*      3) ISSUE A CONVERSE REQUEST TO SEND AND RECEIVE DATA.          * 00120000
*                                                                     * 00130000
*      4) FREE THE SESSION.                                           * 00140000
*                                                                     * 00150000
* ANY NUMBER OF TRANSACTIONS CAN USE THIS PROGRAM AT ANY ONE TIME     * 00160000
*                                                                     * 00170000
*********************************************************************** 00210000

*                                                                       00220000
R15      EQU   15                       REGISTERS                       00230000
R14      EQU   14                       REGISTERS                       00240000
R13      EQU   13                       REGISTERS                       00250000
R12      EQU   12                       REGISTERS                       00260000
R11      EQU   11                       REGISTERS                       00270000
R10      EQU   10                       REGISTERS                       00280000
```

```
R9       EQU   9                       REGISTERS                        00290000
R8       EQU   8                       REGISTERS                        00300000
R7       EQU   7                       REGISTERS                        00310000
R6       EQU   6                       REGISTERS                        00320000
R5       EQU   5                       REGISTERS                        00330000
R4       EQU   4                       REGISTERS                        00340000
R3       EQU   3                       REGISTERS                        00350000

R2       EQU   2                       REGISTERS                        00360000
R1       EQU   1                       REGISTERS                        00370000
R0       EQU   0                       REGISTERS                        00380000
         PRINT NOGEN                                                    00390000

*                                                                       00400000
* FIXED REGISTERS                                                       00410000
*                                                                       00420000
EIBREG   EQU   R9                                                       00430000
*                                                                       00440000
VAXOUT   DFHEIENT  EIBREG=(EIBREG)                                      00450000
*                                                                       00460000
****    REQUEST SYSID FROM THE TERMINAL                                 00662017
*                                                                       00670000
SYSLOOP  DS    0H                                                       00670512
*                                                                       00670413
         MVC   TBUFLEN,SYSRQLEN                                         00670213
*                                                                       00670712
****    SEND A MESSAGE TO THE TERMINAL                                  00670817
*                                                                       00670917
         EXEC  CICS SEND                                               +00671012
                    FROM  (SYSIDREQ)                                   +00671113
                    LENGTH(TBUFLEN)                                    +00671230

                    ERASE                                               00671412
*                                                                       00671512
******* GET SYSID FROM THE TERMINAL                                     00671613
*                                                                       00671702
         LA    R8,L’TERMBUF            SET RECEIVE BUFFER LENGTH
         STH   R8,TBUFLEN
*                                                                       00671702
         EXEC  CICS RECEIVE                                            +00671813
                    INTO  (TERMBUF)                                    +00671913
                    LENGTH(TBUFLEN)                                     00672030
*                                                                       00673013
******* SAVE CONNECTION SYSID                                           00671613
*                                                                       00673013
         LH   R8,TBUFLEN
         CH   R8,=H’4’
         BL   SYSLOOP
         MVC  SYSID,TERMBUF
```

```
*                                                                       00560000
** ALLOCATE A SESSION                                                   00570000
*                                                                       00580000
         EXEC  CICS ALLOCATE                                           +00590000
                    SYSID    (SYSID)                                    00600020
*                                                                       00601013
         MVC   MESSAGE(L’MHEADER),MHEADER INITIALIZE MESSAGE HEADER     00610013
         MVC   RESOURCE,EIBRSRCE       SAVE RESOURCE (CONVID)           00610013
         MVC   MSGTRMID,EIBTRMID       SAVE THE TERMINAL ID             00611013
*                                                                       00620000

** CONNECT TO VAX TRANSACTION PROGRAM                                   00630000
*                                                                       00640000
         EXEC  CICS CONNECT PROCESS                                    +00650020
                    CONVID    (RESOURCE)                               +00651020
                    PROCNAME  (TRANID)                                 +00652020
                    PROCLENGTH(TRNLEN)                                 +00653020
                    SYNCLEVEL (SYNLVL)                                  00654020
*                                                                       00661017
****    SEND A GREETING TO THE TERMINAL                                 00662017
*                                                                       00670000
         MVC   TERMBUF(L’GREETING),GREETING                             00670113
         MVC   TBUFLEN,GRTNGLEN                                         00670213
*                                                                       00670413
MSGLOOP  DS    0H                                                       00670512
*                                                                       00670712
****    SEND A MESSAGE TO THE TERMINAL                                  00670817
*                                                                       00670917
         EXEC  CICS SEND                                               +00671012
                    FROM  (TERMBUF)                                    +00671113
                    LENGTH(TBUFLEN)                                    +00671230
                    ERASE                                               00671412
*                                                                       00671512
******* GET A MESSAGE FROM THE TERMINAL                                 00671613
*                                                                       00671702
         LA    R8,L’TERMBUF            SET RECEIVE BUFFER LENGTH
         STH   R8,TBUFLEN
*                                                                       00671702
         EXEC  CICS RECEIVE                                            +00671813
                    INTO  (TERMBUF)                                    +00671913
                    LENGTH(TBUFLEN)                                    +00672030
                    ASIS                                                00672213
*                                                                       00673013
         CLC   TERMBUF(7),=C’$*$TERM’                                   00676118
         BE    PLUTERM                                                  00676221
*                                                                       00677013
** SEND THE MESSAGE TO THE LU62 CONVERSATION PARTNER                    00680013
*                                                                       00690000
         LH    R8,TBUFLEN
         LA    R8,L'MHEADER(0,R8)     INCREASE MSG LENGTH FOR HEADER
         STH   R8,TBUFLEN
*                                                                       00690000
         EXEC  CICS CONVERSE                                           +00700000
               CONVID    (RESOURCE)                                    +00710000
               FROM       (MESSAGE)                                    +00720000
               FROMLENGTH (TBUFLEN)                                    +00730030
               SET        (R6)                                         +00740008
               TOLENGTH   (INLEN)                                       00750030
*                                                                       00760000

         MVC   XDFEIFLG,EIBSYNC        SAVE EIB FLAGS                   00770021
TESTCONF CLI   XCONF,X'FF'                                              00780021
         BNE   TESTFREE                                                 00790021
*                                                                       00800021
         EXEC  CICS ISSUE CONFIRMATION                                  00800121
TESTFREE CLI   XFREE,X'FF'                                              00800221
         BE    SLUTERM                                                  00800321
TESTRECV CLI   XRECV,X'FF'                                              00800421
*                                                                       00801009
***   DISPLAY THE MESSAGE FROM THE REMOTE TPN ON THE 3270 TERMINAL      00802009
*                                                                       00803009
         LA    R4,TERMBUF          POINT TO 3270 TERMINAL BUFFER
         LA    R5,L'TERMBUF        SET TO LENGTH OF 3270 BUFFER
         LH    R7,INLEN            SET TO LENGTH OF REMOTE TPN MSG
         CR    R5,R7               IF 3270 BUFFER IS SMALLER THAN MSG
         BL    *+6                    MOVE BUFFER NUMBER OF BYTES
         LR    R5,R7               ELSE, MOVE MSG NUMBER OF BYTES
         STH   R5,TBUFLEN          SAVE MESSAGE LENGTH FOR SEND
         MVCL  R4,R6              MOVE REMOTE TPN MSG TO 3270 BUFFER   00803116
*                                                                       00810000
** SEND THE NEXT MESSAGE PLEASE                                         00820009
*                                                                       00830000
         B     MSGLOOP             LOOP UNTIL $*$TERM                   00840017
*                                                                       00850000
PLUTERM  DS    0H                                                       00850121
         MVC   TERMBUF(L'PLUTMSG),PLUTMSG                               00850221
         B     SEND3270                                                 00850321
SLUTERM  DS    0H                                                       00850421
         MVC   TERMBUF(L'SLUTMSG),SLUTMSG                               00850521
*                                                                       00850621
SEND3270 DS    0H                                                       00850721
         EXEC  CICS SEND                                               +00850821
                    FROM  (TERMBUF)                                    +00850921
                    LENGTH(TMSGLEN)                                    +00851027
                    ERASE                                               00851221
*                                                                       00851321

SENDCEB  DS    0H                                                       00851417
```

```
         EXEC  CICS SEND LAST                                           +00852019
               CONVID (RESOURCE)                                        +00853019
               FROM   (TERMBUF)                                         +00854019
               LENGTH (TBUFLEN)                                          00855030
*                                                                        00856017
** AND RETURN                                                            00860000
*                                                                        00870000
RETURN   DS    0H                                                        00880000
*                                                                        00882017
         EXEC  CICS FREE                                                +00883017
               SESSION    (RESOURCE)                                     00884017
         EXEC  CICS RETURN                                               00890000

*                                                                        00900000
TRANID   DC  CL4'NOTR'                                                   00930032
TRNLEN   DC  AL2(*-TRANID)                                               00940000

SYNLVL   DC  H'0'                                                        00950026
SYSIDREQ DC  C'ENTER CONNECTION SYSID : '                                00951012
SYSRQLEN DC  AL2(*-SYSIDREQ)                                             00952012
GREETING DC  C'ENTER MESSAGE : '                                         00951012
GRTNGLEN DC  AL2(*-GREETING)                                             00952012
PLUTMSG  DC  C'*** TERMINATED BY HOST TPN   ***'                         00953027
SLUTMSG  DC  C'*** TERMINATED BY REMOTE TPN ***'                         00954021
TMSGLEN  DC  AL2(*-SLUTMSG)                                              00955027
MHEADER  DC  C'***      *** '                                            00960000
*                                                                        01152121
         LTORG                                                           01154023
*                                                                        01152121
********************************************************* ************ 01152221
*  EIB EXEC INTERFACE BLOCK STORAGE AREA                                 01152321
********************************************************* ************ 01152421
*                                                                        01152521
DFHEISTG DSECT  ,
XDFEIFLG DS  0CL10                                                       01152621
XSYNC    DS  C                                                           01152722
XFREE    DS  C                                                           01152822
XRECV    DS  C                                                           01152922
XSEND    DS  C                                                           01153022
XATT     DS  C                                                           01153122
XEOC     DS  C                                                           01153222
XFMH     DS  C                                                           01153322
XCOMPL   DS  C                                                           01153422
XSIG     DS  C                                                           01153522
XCONF    DS  C                                                           01153622
*                                                                        01153721
TBUFLEN  DS  H                                                           01152013
INLEN    DS  H                                                           01142008
SYSID    DS  CL4                                                         00910031
```

```
RESOURCE DS  CL8                                                        00920005
MESSAGE  DS  CL4                                                        00960000
MSGTRMID DS  CL4                                                        00970000
         DS  CL4                                                        00980027
TERMBUF  DS  CL1920                                                     01151013
********************************************************************** 01153924
         END                                                            01160023
------------------------------------------------------------------------------
```

Index