SunLink™ P2P LU6.2 9.1 Programmer's Manual

The Network Is the Computer™

Sun Microsystems Computer Company
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Part No.: 802-2680-12
Revision A, August 1997

Contents

Figures

Tables

Preface

This book is a reference for developers of SunLink P2P LU6.2 9.1 programs on Sun™ Workstations™. It describes the SunLink implementation of the LU6.2 transaction verbs. These verbs allow program-to-program communications across an IBM SNA network using Logical Unit type 6.2 (LU6.2). SunLink P2P LU6.2 9.1 uses the services of the SunLink SNA PU2.1 9.1 server product to provide peer-to-peer SNA communications support for Sun Workstations. The SunLink SNA PU2.1 9.1 product set includes:

- SNA interface, including SDLC, X.25, IBM Token Ring, Ethernet, and physical device drivers implemented in Unix System V streams
- A server that controls the SNA interface and provides SNA network access to its client applications
- Client programs, including `sun3270`, `sun3770`, and `sunSNM`
- The SunLink P2P LU6.2 9.1 Application Programming Interfaces (APIs) for users who want to create their own LU6.2 applications, i.e., the SunLink P2P LU6.2 9.1 API described in this document, and the SunLink P2P CPI-C 9.1 API
- Dependent LU APIs for users who want to create their own client applications, including the SunLink LU0 API

Who Should Use This Book

This manual documents the SunLink P2P LU6.2 9.1 API. Use this book as a guide and reference for using the API. As a SunLink P2P LU6.2 9.1 programmer, you are expected to be familiar with the C programming language and the Unix operating system. You should also be familiar with the SNA Advanced-Program-to-Program-Communication (APPC) model for distributed transaction processing.

How to Use This Book

This manual is organized as follows:

- **Chapter 1, "Introduction,"** introduces the functions and features of SunLink P2P LU6.2 9.1. It describes how the client/server paradigm is used to allow the distribution of your programs on the Local Area Network (LAN), and illustrates how programs written using the API communicate with peer programs in an SNA environment to provide an APPC application.
- **Chapter 2, "SunLink LU6.2 Concepts,"** introduces LU6.2 terminology and describes the concepts you need to write APPC applications.
- **Chapter 3, "Getting Started with SunLink LU6.2,"** is designed to introduce you to SunLink P2P LU6.2 9.1 application programming. The chapter guides you through the steps necessary to build and run a sample APPC application. The sample programs that comprise the APPC application are located in the appendixes, together with the sample SNA configuration. This is an important chapter for understanding and implementing the SunLink P2P LU6.2 9.1 application program.
- **Chapter 4, "Configuration,"** describes how to configure local LU6.2s and define TPs, partner LUs, and modes. Corresponding VTAM, CICS and OS/400 configurations are discussed.
- **Chapter 5, "Using the LU6.2 API,"** describes the nature of the LU6.2 API and its use. This chapter shows you how to organize your program and, with examples, illustrates how the API is used to perform most of the "standard" LU6.2 operations.
- **Chapter 6, "man Page Conventions,"** describes the format and conventions used in the man pages in chapters 7 to 12.

- **Chapter 7, "Connection Verbs,"** describes the connection verbs that are used to establish and maintain connection to one or more SunLink SNA PU2.1 9.1 servers.
- **Chapter 8, "Basic Conversation Verbs,"** describes the SunLink SNA PU2.1 9.1 basic conversation verbs.
- **Chapter 9, "Mapped Conversation Verbs,"** describes the SunLink SNA PU2.1 9.1 mapped conversation verbs.
- **Chapter 10, "Type-Independent Verbs,"** describes the SunLink SNA PU2.1 9.1 type-independent verbs for basic and mapped conversations.
- **Chapter 11, "Control Operator Verbs,"** describes the SunLink SNA PU2.1 9.1 control operator (COPR) verbs that are used to control and monitor the local LU.
- **Chapter 12, "SunLink LU6.2 Utilities,"** describes the utilities supplied with the SunLink LU6.2 API.
- **Appendix A, "SunLink LU6.2 Return Codes,"** explains the return codes that are passed to the program when an execution verb is completed.
- **Appendix B, "Conversation State Table,"** provides a conversation state table.
- **Appendix C, "LU 6.2 Include Files,"** contains the SunLink P2P LU6.2 9.1 include files.
- **Appendix D, "Sample LU6.2 Programs,"** provides code examples.
- **Appendix E, "SunLink LU6.2 Configuration Examples,"** provides code samples of SunLink configurations.
- **Appendix F, "LU6.2 Sync-Point,"** describes Sync-Point flows and recovery.
- **Appendix G, "SunLink LU6.2 9.0 to 9.1 Instructions,"** describes the migration path for using your SunLink 8.0 P2P LU6.2 applications. Also described are the object files with libraries and discrepancies between the SunLink 8.0 and SunLink 9.1 products.

Related Documentation

Sun Documentation

- *SunLink SNA PU2.1 9.1 Server Configuration and Administration Manual*
- *SunLink SNA 3270 9.1 End Node Planning and Installation Manual*
- *SunLink P2P CPI-C 9.1 Programmer's Guide*

IBM Documentation

- *IBM Systems Network Architecture Concepts and Products, GC30-3072*
- *IBM Systems Network Architecture Technical Overview, GC30-3073*
- *IBM SNA Transaction Programmer's Reference Manual, GC30-3084*
- *IBM SNA LU6.2 Reference: Peer Protocols, SC31-6808*
- *IBM SNA Formats, GA27-3136*

Ordering Sun Documents

SunDocs^{SM} is a distribution program for Sun Microsystems technical documentation. Easy, convenient ordering and quick delivery is available from SunExpress. You can find a full listing of available documentation on the World Wide Web: `http://www.sun.com/sunexpress/`

Table P-1 SunExpress Contact Information

Country	**Telephone**	**Fax**
United States	1-800-873-7869	1-800-944-0661
United Kingdom	0800-89-88-88	0800-89-88-87
Canada	1-800-873-7869	1-800-944-0661
France	0800-90-61-57	0800-90-61-58
Belgium	02-720-09-09	02-725-88-50
Luxembourg	32-2-720-09-09	32-2-725-88-50
Germany	01-30-81-61-91	01-30-81-61-92
The Netherlands	06-022-34-45	06-022-34-46
Sweden	020-79-57-26	020-79-57-27

Table P-1 SunExpress Contact Information *(Continued)*

Switzerland	0800-55-19-26	0800-55-19-27
Holland	06-022-34-45	06-022-34-46
Japan	0120-33-9096	0120-33-9097

Typographic Conventions

The following table describes the typographic changes used in this book.

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output.	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`

`AaBbCc123`	What you type, when contrasted with on-screen computer output.	`%` **`su`**
		`Password:`

AaBbCc123	Command-line variable: replace with a real name or value.	To delete a file, type `rm` *filename*.
	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be root to do this.

Sun Welcomes Your Comments

Please use the *Reader Comment Card* that accompanies this document. We are interested in improving our documentation and welcome your comments and suggestions.

If a card is not available, you can email or fax your comments to us. Please include the part number of your document in the subject line of your email or fax message.

- Email: `smcc-docs@sun.com`
- Fax: SMCC Document Feedback
  1-415-786-6443

Introduction 1

This book is a guide and reference for SunLink P2P LU6.2 9.1 developers. The SunLink P2P LU6.2 9.1 application programming interface (API) allows program-to-program communications using IBM SNA logical unit 6.2 (LU6.2). It provides a consistent and full-featured implementation of the LU6.2 verb set defined in the *IBM SNA Transaction Programmer's Reference Manual*.

You can use the SunLink P2P LU6.2 9.1 API program to:

- Connect to multiple SunLink SNA PU2.1 9.1 servers
- Associate with multiple local LUs
- Initiate or accept multiple conversations
- Handle multiple conversations concurrently
- Issue Control Operator verbs.

1.1 SunLink SNA PU2.1 9.1 Server

The API provides programs with access to the SunLink SNA PU2.1 9.1 server. The server can support multiple SunLink P2P LU6.2 9.1 client programs, and client programs can connect to multiple servers. As a TCP/IP network server, the SunLink SNA PU2.1 9.1 server provides LU6.2 and PU2.1 node services to SunLink P2P LU6.2 9.1 client programs and SunLink IBM connectivity end-point products such as `sun3270` and `sun3770` running anywhere in the

TCP/IP internetwork. The SunLink SNA PU2.1 9.1 server supports IBM Token Ring, Ethernet, SDLC, and X.25 network connections into the SNA backbone network.

Figure 1-1 shows the environment in which your SunLink P2P LU6.2 9.1 programs run. Instances of SunLink LU6.2 programs are shown as TPx. Note the CPI-C program CP1, which is written using the SunLink LU6.2 CPI-C API. This API implements the "standard" programming interface for LU6.2 specified by *IBM System Application Architecture (SAA) Common Programming Interface*. SunLink LU6.2 CPI-C is layered on top of the SunLink LU6.2 API. See the *SunLink P2P CPI-C 9.1 Programmer's Manual* for more information.

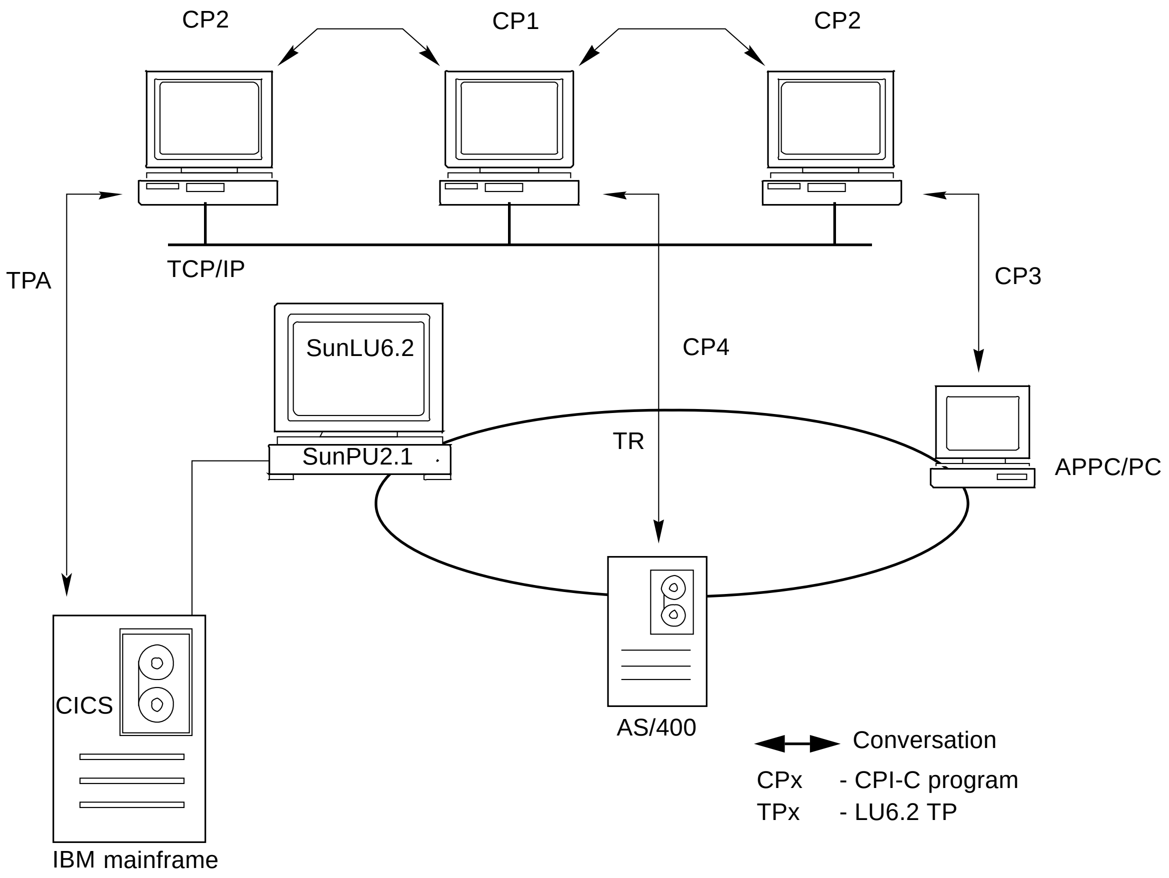


Figure 1-1 SunLink P2P LU6.2 Application Support

1.2 SunLink LU6.2 API Features

Table 1-1 below list the LU6.2 verbs supported by the SunLink P2P LU6.2 9.1 API. See the [TPRM] for a functional description of the LU6.2 verbs. The tables also identify Sun extensions to the LU6.2 verb set. These extensions are noted by (*), and their function is summarized.

Note – If you are the SunLink PTP LU6.2 8.0 user, be sure to read the file, SunLink Notes that is included in the distribution media. It describes how to convert the 8.0 TP and utility proprietary verbs into 9.1 proprietary verbs.

1.2.1 Conversation Verbs

Table 1-1 LU6.2 Verbs supported by SunLink P2P LU6.2 9.1 API

SunLink LU6.2	LU6.2
Basic Conversation Verbs	
`lu62_allocate`	`ALLOCATE`
`lu62_confirm`	`CONFIRM`
`lu62_confirmed`	`CONFIRMED`
`lu62_deallocate`	`DEALLOCATE`
`lu62_flush`	`FLUSH`
`lu62_get_attributes`	`GET_ATTRIBUTES`
`lu62_post_on_receipt`	`POST_ON_RECEIPT`
`lu62_prep_to_receive`	`PREPARE_TO_RECEIVE`
`lu62_receive_and_wait`	`RECEIVE_AND_WAIT`
`lu62_receive_immediate`	`RECEIVE_IMMEDIATE`
`lu62_request_to_send`	`REQUEST_TO_SEND`
`lu62_send_data`	`SEND_DATA`
`lu62_send_error`	`SEND_ERROR`
`lu62_test`	`TEST`

Table 1-1 LU6.2 Verbs supported by SunLink P2P LU6.2 9.1 API *(Continued)*

SunLink LU6.2	LU6.2
Mapped Conversation Verbs	
lu62_mc_allocate	MC_ALLOCATE
lu62_mc_confirm	MC_CONFIRM
lu62_mc_confirmed	MC_CONFIRMED
lu62_mc_deallocate	MC_DEALLOCATE
lu62_mc_flush	MC_FLUSH
lu62_mc_get_attributes	MC_GET_ATTRIBUTES
lu62_mc_post_on_receipt	MC_POST_ON_RECEIPT
lu62_mc_prep_to_receive	MC_PREPARE_TO_RECEIVE
lu62_mc_receive_and_wait	MC_RECEIVE_AND_WAIT
lu62_mc_receive_immediate	MC_RECEIVE_IMMEDIATE
lu62_mc_request_to_send	MC_REQUEST_TO_SEND
lu62_mc_send_data	MC_SEND_DATA
lu62_mc_send_error	MC_SEND_ERROR
lu62_mc_test	MC_TEST
Type-Independent Verbs	
*lu62_abort	Aborts conversation processing
*lu62_accept	Listens for and accept an incoming conversation
*lu62_register_tp	Registers a local TP for incoming conversations
lu62_get_tp_properties	GET_TP_PROPERTIES
lu62_get_type	GET_TYPE
*lu62_listen	Listens for an incoming conversation
*lu62_send_ps_data	Sends Presentation Services data
lu62_wait	WAIT

1.2.2 Control Operator Verbs

The SunLink P2P LU6.2 9.1 API implements the following control operator verbs, as shown in Table 1-2.

Table 1-2 Control Operator Verbs Implemented by API

SunLink LU6.2	**LU6.2**
CNOS Verbs	
`lu62_change_session_limit`	`CHANGE_SESSION_LIMIT`
`lu62_initialize_session_limit`	`INITIALIZE_SESSION_LIMIT`
`lu62_process_session_limit`	`PROCESS_SESSION_LIMIT`
`lu62_reset_session_limit`	`RESET_SESSION_LIMIT`
Session Control Verbs	
`lu62_activate_session`	`ACTIVATE_SESSION`
`lu62_deactivate_conv_group`	`DEACTIVATE_CONVERSATION_GROUP`
`lu62_deactivate_session`	`DEACTIVATE_SESSION`
LU Definition Verbs	
`lu62_display_local_lu`	`DISPLAY_LOCAL_LU`
`lu62_display_mode`	`DISPLAY_MODE`
`lu62_display_remote_lu`	`DISPLAY_REMOTE_LU`
`lu62_display_tp`	`DISPLAY_TP`

1.2.3 Connection Verbs

Connection verbs are SunLink P2P LU6.2 9.1 extensions that are used to establish and control the connections to SunLink SNA PU2.1 9.1 server(s). Your program establishes a connection to the SunLink SNA PU2.1 9.1 server for each local LU with which it is associated.

Conversations are allocated on sessions between the local LU and one of its defined partner LUs. All such conversations are multiplexed over the single connection to the server. Conversations may be `BLOCKING` or `NON_BLOCKING`. When a verb is issued on a non-blocking conversation, you must issue a `lu62_wait_server` call when you are ready to process the return.

Table 1-3 Connection Verbs Supported by SunLink P2P LU6.2 9.1

SunLink LU6.2	**Function**
*`lu62_open`	Opens a connection with the SunLink PU2.1 SNA server
*`lu62_close`	Closes a connection with the SunLink PU2.1 SNA server
*`lu62_set_processing_mo e`	Sets to `BLOCKING` or `NON_BLOCKING` mode
*`lu62_wait_server`	Waits for any posted conversation
*`lu62_get_readfds`	Returns the select `fdsets` for posted conversations

1.2.4 Character Conversion

Your Unix system uses the ASCII character set while the LU6.2 protocol requires that certain characteristics are transmitted as EBCDIC characters. The SunLink P2P LU6.2 9.1 API allows you to work with native ASCII characters for all such characteristics, and converts them automatically to and from EBCDIC, as required. These characteristics are:

- Mode name
- Partner `LU` name
- Remote `TP` name
- Log data
- Conversation security `user_ID`
- Conversation security password
- Conversation security profile

User data must, however, be sent in the character set expected by the remote TP, ASCII or EBCDIC. SunLink P2P LU6.2 9.1 API provides conversion routines for EBCDIC character set 00640.

1.2.5 Unix Security

SunLink P2P LU6.2 9.1 can be configured to use Unix security mechanisms to enforce conversation-level security. Thus, the user identifier, password (and optional profile), supplied on a conversation start-up request, is required to correspond, respectively, to a user name, password (and group), defined to Unix. See the *SunLink SNA PU2.1 9.1 Configuration and Administration Manual* for more information.

1.2.6 Tracing

SunLink P2P LU6.2 9.1 incorporates extensive trace capabilities to assist in developing and debugging your programs. Trace points are built-in to the SunLink P2P LU6.2 9.1 API to provide information on program calls, program errors, exchange of buffers with the SunLink SNA PU2.1 9.1 server, and internal API errors. Traces may be selectively enabled using external trace flags, and output is written to a unique file. The trace facility is exposed so that you can include trace points in your own programs.

1.3 Unsupported Features

The following subsections describe the P2P LU6.2 unsupported features.

1.3.1 Map Names

SunLink P2P LU6.2 9.1 does not currently support Map Names in its mapped conversation support.

1.3.2 Sync-point Services

SunLink P2P LU6.2 9.1 provides limited support for LU6.2 sync-point services. TPs can establish sync-level sync-point conversations and exchange sync-point flows using the `lu62_send_ps_data` verb. An external sync-point manager can send sync-point recovery GDS variables as FMH data on mapped conversations. See Appendix F for a more complete description of the sync-point support provided.

Sync-point verbs are not recognized, sync-point states are not maintained and, if the conversation sync-level is sync-point, deallocate and `prepare_to_receive` types of sync-level are not supported.

1.3.2.1 Control Operator LU6.2 Definition Verbs

SunLink P2P LU6.2 9.1 does not currently support dynamic configuration. The SunLink P2P LU6.2 9.1 API, therefore, does not support the following [TPRM] LU6.2 verbs:

- `DEFINE_LOCAL_LU`
- `DEFINE_MODE`
- `DEFINE_TP`
- `DEFINE_REMOTE_LU`
- `DELETE`

1.4 SunLink LU6.2 Components

To develop a SunLink P2P LU6.2 9.1 application program, you should be familiar with the SunLink P2P LU6.2 9.1 components. In particular, the programmer should know the LU6.2 parameters defined to the SunLink SNA PU2.1 9.1 server. Refer to the *SunLink SNA PU2.1 9.1 Server Configuration and Administration Manual* for more information. Figure 1-2 identifies the SunLink P2P LU6.2 9.1 components, including the SunLink P2P LU6.2 9.1 API.

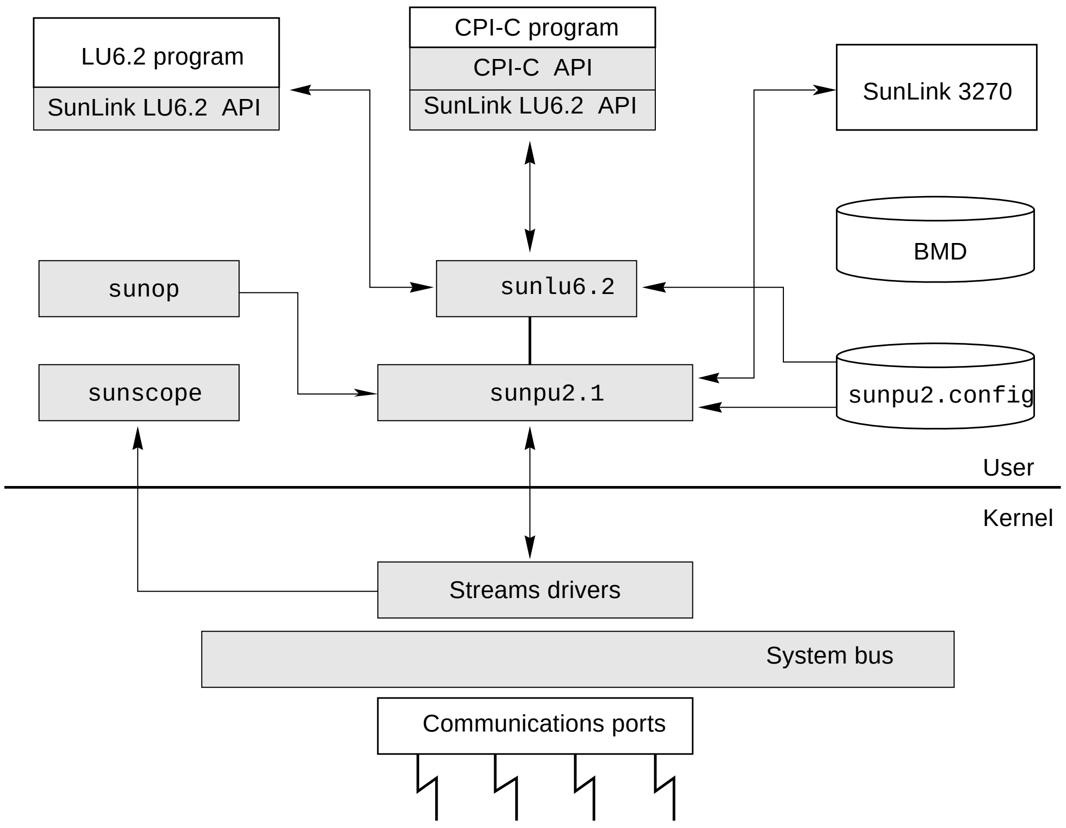


Figure 1-2 SunLink P2P LU6.2 9.1 Components

The following subsections describe the LU6.2 components described in Figure 1-2.

`sunlu6.2`

Your programs connect to the `sunlu6.2` daemon process for LU and presentation services. The `sunlu6.2` interfaces with the `sunpu2.1` daemon to access the network and to cooperate in control point functions.

`sunpu2.1`

The `sunpu2.1` daemon process implements the higher-level SNA protocols and services. This process runs in the background in the application space. It is responsible for starting the `sunlu6.2` daemon and interfaces to the protocol drivers and with its client programs. In addition, the `sunpu2.1` processes local operator requests from `sunop`.

`sunpu2.config`

The `sunpu2.1` and `sunlu6.2` daemon processes read a configuration file to determine the configured SNA resources. The default name of this configuration file is `sunpu2.config`.

BMD

The SunLink Basic Message Database lists all error and informational messages displayed by SunLink IBM Connectivity software.

`sunop`

The `sunop` process implements the SunLink IBM Connectivity local operator interface. This process allows you to monitor and control SNA resources.

`sunscope`

The `sunscope` process is a logical data scope, which allows you to monitor the data sent and received on the data links.

Streams Drivers

The serial communication boards and network interface controllers are managed by STREAMS device drivers.

SunLink LU6.2 Concepts 2

This chapter introduces SunLink LU6.2 terminology and describes the concepts you need for writing APPC applications and assumes that you have an understanding of IBM System Network Architecture (SNA). For more information on SNA, refer to IBM's *Systems Network Architecture Concepts and Products*. For the formal definition of the SunLink LU6.2 verb set, see the *IBM SNA Transaction Programmer's Reference Manual*.

2.1 APPC and Logical Unit Type 6.2 (LU6.2)

The SNA Advanced program-to-program communication (APPC) architecture defines the Logical Unit type 6.2 (LU6.2). Its purpose is to support communication between application programs running on any APPC-capable node in an SNA network. This enables distributed processing, in which two or more programs running on different systems cooperate to perform a particular function. Figure 2-1 summarizes the SunLink LU6.2 model.

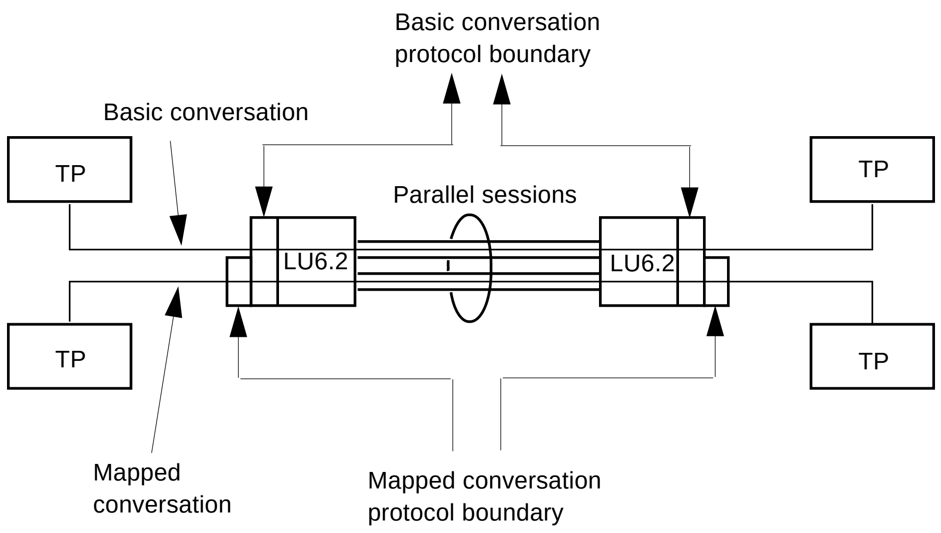

Basic conversation protocol boundary
Basic conversation
TP
TP
Parallel sessions
LU6.2
LU6.2
TP
TP
Mapped conversation
Mapped conversation protocol boundary

Figure 2-1 SunLink LU6.2 Model

All the elements of the SunLink LU6.2 model shown in Figure 2-1 are described below.

2.1.1 Transaction Program (TP)

A TP accesses the SNA network through its local SunLink LU6.2. The SunLink LU6.2 makes resources, including sessions, available to the TP, and operates the session protocols. A key feature of the SunLink LU6.2 is that it can invoke a local TP on receiving an allocation (or *Attach*) request from a remote TP.

Although SunLink LU6.2 has most often been used to implement distributed transaction processing systems, it can be used to support any type of program-to-program communication. In this document, the term *transaction program* is used to describe any program that uses the SunLink LU6.2 protocol to communicate.

TPs, SunLink LU6.2 and sessions are described from the perspective of the local node. Thus the *local* TP connects to its *local* SunLink LU6.2, which in turn is in session with its *partner* (or *remote*) SunLink LU6.2. The *remote* TP is located at the partner LU.

2.1.2 Conversations and Verbs

TPs always communicate in pairs. A logical connection called a *conversation* is established between the two programs. Conversations temporarily use *sessions* that are set up between the TPs' respective SunLink LU6.2s. Programs communicate on conversations using the verbs defined on the *conversation protocol boundary* between the TP and its local SunLink LU6.2; mapped and basic conversation types are supported.

2.1.3 Mapped and Basic Conversations

In a basic conversation, TPs exchange data in a logical record format that includes a two-byte prefix. In a mapped conversation, TPs exchange simple data records (with no length prefix). TPs can also specify mapping information so that the local SunLink LU6.2 can convert data to and from the format understood by the local TP. This feature, however, is usually not implemented. Mapped conversations are easier to program since the TP is not responsible for formatting data into logical records.

2.1.4 Conversation Initiation and State Transitions

A conversation is initiated by one TP that issues an allocate verb. The local SunLink LU6.2 assigns a session (provided one is available or can be activated) to the conversation and issues an allocation request on the session. The partner SunLink LU6.2 receives the request and (provided the request is valid), invokes the remote TP. The conversation is then established with the conversation initiator in *Send* state, and the initiated TP in *Receive* state. As the conversation progresses, the conversation, as perceived by the individual TPs, changes state as the TPs issue verbs. For example, the TP in *Send* state can transition to *Receive* state and cause the remote TP to enter *Send* state by issuing a `Prepare_To_Receive verb`.

The verbs that a program may issue on a conversation depend on the state of the conversation. This is enforced by the conversation protocol boundary. Appendix B, “Conversation State Table,” defines the verbs that are valid in each conversation state, and the state changes (or transitions) that can occur when a verb is issued.

2.1.5 Sessions

In SNA, sessions are the logical connections that are maintained between network addressable units (NAUs). SunLink LU6.2 sessions are allocated to one conversation at a time, but since their establishment and removal involves significant processing, a session is usually left running when a conversation terminates, so it can be available for the next conversation.

2.1.6 Modes

The characteristics of a session between a pair of SunLink LU6.2s are specified with a mode. The mode defines various communications and usage parameters, including:

- *preferred RU size* specifies the preferred size of the request/response units (RUs) that are exchanged by the two SunLink LU6.2s
- *parallel session support* specifies whether parallel sessions are supported between the two SunLink LU6.2s
- *maximum session limit* defines the maximum number of active sessions of this particular type that can exist between the two SunLink LU6.2s. If this number is greater than 1, *parallel sessions* must be supported
- *polarity* of the sessions

Modes must be defined identically to each SunLink LU6.2. Session limits and session polarities may be changed using a set of control operator verbs called change number of sessions (CNOS) verbs.

2.1.7 Parallel Sessions

Parallel session support indicates that the SunLink LU6.2 pair can support more than one active session of the same mode at a time. Parallel sessions allow optimal use of the SunLink LU6.2 pair, enabling multiple transactions of the same type to proceed concurrently. An installation will normally designate some of the defined sessions as *auto-activated* sessions. These sessions are automatically activated whenever the local SunLink LU6.2 is started or reset. Thus a free *pool* of sessions is established, avoiding the session activation overhead whenever a conversation requests a session.

2.1.8 Session Polarity

The session polarity determines which SunLink LU6.2 “wins” if both LUs simultaneously attempt to activate a particular session. When the session is a *contention-winner* (or *first-speaker*) the local LU always wins when there is contention. When the session is a *contention-loser* (or *bidder*), the partner LU always wins. In this case the contention-loser SunLink LU6.2 bids the contention-winner to use the session. The mode defines the minimum number of *conwinner* and *conloser* sessions available.

2.1.9 Control Operator Verbs

The SunLink LU6.2 supports another protocol boundary not shown in Figure 2-1. Control operator verbs are specified that allow a control operator program to define, monitor, and control SunLink LU6.2 resources. The control operator verbs fall into three main categories:

- Change Number of Sessions (CNOS) verbs
- Session control verbs
- SunLink LU6.2 definition verbs

CNOS verbs enable the control operator to initialize, change, or reset a mode session limit and session polarities. When the mode supports parallel sessions, changes to these CNOS parameters are negotiated by the involved SunLink LU6.2s. The local SunLink LU6.2 initiates a conversation with its partner and transmits a CNOS request. The partner SunLink LU6.2 returns a CNOS response, at which point both SunLink LU6.2s make the agreed changes.

Session control verbs enable the control operator to activate and deactivate sessions. SunLink LU6.2 definition verbs are used to define and display the network resources controlled by the local SunLink LU6.2.

Getting Started with SunLink LU6.2 3

Getting Started is designed to guide you through the steps necessary to install and run an APPC application in which two SunLink P2P LU6.2 9.1 client programs, running on different Unix workstations, send and receive data. The sample programs used are distributed with SunLink P2P LU6.2 9.1 and are included in Appendix D.

The `tp_sr` initiates the conversation. The `tp_sr` allocates a basic conversation and sends data with or without a request for confirmation to `tp_rs.`The `tp_sr` then enters *Receive* state to receive data from `tp_rs`.

The `tp_rs` accepts incoming conversations, and issues an `lu62_receive_and_wait` command to receive information. If data is received, it is displayed. If a confirmation request is received, it is confirmed. If a send indication is received, the program enters *Send* state and sends data to `tp_sr`.

These programs are compiled and linked with the SunLink P2P LU6.2 9.1 API library, `liblu62.a`. Before the programs can be run, however, the SunLink P2P LU6.2 9.1 software must be installed, configured, and activated. Sun delivers SunLink P2P LU6.2 9.1 with installation scripts and sample configurations to simplify the process.

Getting started with SunLink P2P LU6.2 9.1 is performed in three stages.

- In the first stage, the sample application will be run over an *intra-node* session, i.e., `tp_sr` and `tp_rs` and is executed on the same workstation. This allows you to remain isolated from network considerations and to concentrate solely on the LU6.2 configuration.

- In the second stage, `tp_sr` and `tp_rs` will again execute on the same workstation, but will communicate over the Token Ring. This allows you to verify the Token Ring connection.
- In the final stage, the programs are run on different workstations in the SNA network. The LU6.2 configuration is split between two peer SunLink SNA PU2.1 9.1 nodes. This introduces you to the network configuration issues.

Getting Started in the *SunLink SNA PU2.1 9.1 Server Configuration and User's Manual* provides step-by-step instructions to configure and start up the SunLink PU2.1 SNA server.

The sample configuration files are distributed with SunLink P2P LU6.2 9.1 and are documented in Appendix E. The configuration files are located in the SunLink installation directory.

3.1 Installing SunLink LU6.2

The installation process extracts the SunLink SNA PU2.1 9.1 server and SunLink P2P LU6.2 9.1 software from the distribution media and installs it on your system. Please consult the *SunLink SNA3270 9.1 End Node Planning and Installation Manual* for detailed instructions. A summary of the procedure follows:

1. Install communications hardware and software, as necessary.
2. Install the SunLink SNA PU2.1 9.1 server, SunLink P2P LU6.2 9.1, and FlexLM product files from the distribution media.
3. Obtain and install the required FlexLM licenses.

SunLink SNA PU2.1 9.1 server, and SunLink P2P LU6.2 9.1 are now installed on your Sun Workstation and ready for use. All path names in the remainder of this chapter are specified relative to the Sun installation directory, which should be `/opt`. The *SunLink SNA 3270 9.1 End Node Planning and Installation Guide*, which is system-specific, identifies the installation directory for your particular system type.

3.2 Intra-Node Configuration

Figure 3-1 depicts the sample intra-node configuration. The corresponding configuration file, `sunlu62.local`, is distributed with SunLink LU6.2 and is included in Appendix E. Figure 3-1 shows the pertinent configuration parameters.

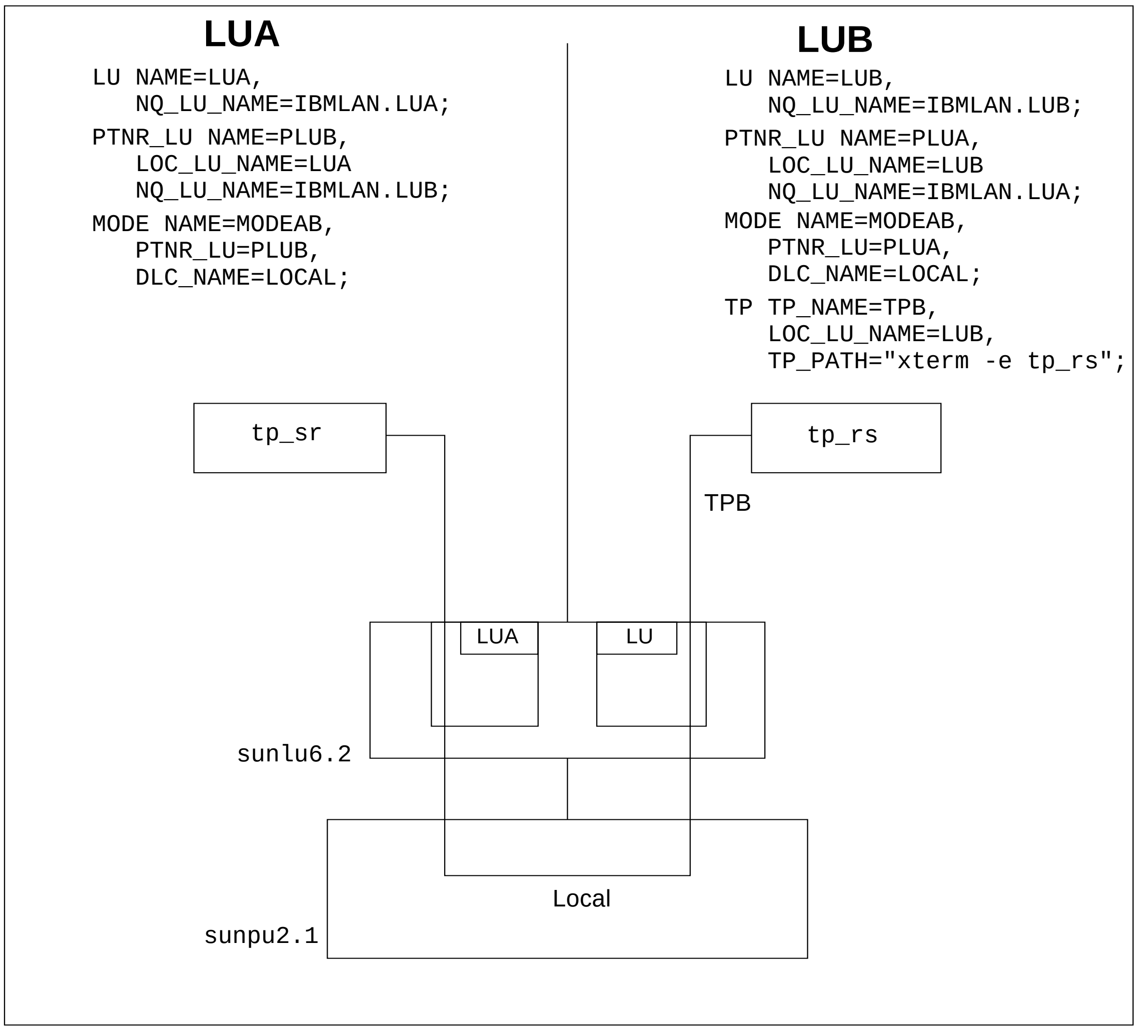


Figure 3-1 Intra-Node Configuration

tp_sr will connect to LUA and allocate a conversation with TPB located at the partner LU, PLUB, using mode MODEAB. At LUB, a TP_PATH parameter is configured for TPB. Thus tp_rs will be invoked using this shell command when an allocate request is received for TPB.

3.3 Starting the SunLink PU2.1 SNA Server

You are now ready to start the SunLink SNA PU2.1 9.1 server. The *SunLink SNA PU2.1 9.1 Server Configuration and Administration Manual* lists the options for invoking the sunpu2.1 process. This manual has a troubleshooting chapter to aid in resolving SNA connectivity problems.

The sunpu2.1 reads the local configuration file to learn about its SNA resources during initialization.

♦ **To invoke the SunLink PU2.1 SNA server with the sample configuration sunlu62.local, enter the following Unix command as superuser:**

```
# cd /opt/SUNWpu21
# sunpu2.1 -f ../SUNWlu62/config/sunlu62.local
```

The sunpu2.1 daemon will terminate immediately if an error is detected during initialization. If initialization is successful, sunpu2.1 responds as follows:

```
# sunpu2.1 -f ../SUNWlu62/config/sunlu62.local
PU200001 : Initializing SunLink PU2.1 SNA Server
BCFG0104 : *** WARNING: Duplicate NQ Name 'IBMLAN.LUA' Specified
PU200002 : Initialization complete
Copyright (c)1997 Sun Microsystems, Inc.
LU620003 : Parsing Configuration,
../SunLU62/config/sunlu62.local
BCFG0104 : *** WARNING: Duplicate NQ Name 'IBMLAN.LUA' Specified
LU620005 : Initialization Started
LU620006 : Initialization Complete
```

The “Duplicate NQ Name” WARNINGs can be ignored because this is a loopback configuration.

The `sunlu6.2` is started automatically by `sunpu2.1`. Note the use of the -d (debug) flag. This specifies that `sunpu2.1` is to remain in the foreground, and enables you to invoke `tp_rs` in a shell tool window with standard IO available. The `-d` flag is normally omitted, however. This allows `sunpu2.1` to become a daemon that runs as its own process group leader is disassociated from the controlling terminal, and is detached from standard IO.

3.4 Stopping the SunLink PU2.1 SNA Server

♦ **Use the `sunop` application to terminate the SunLink PU2.1 SNA server:**

```
% sunop
SunLink Controller
-> kill
(0) kill
->
OP200018 : SunPU2 SNA Server connection broken
%
```

If you change the configuration, you must stop and restart the server for the changes to occur. The SunLink PU2.1 SNA server only accesses its configuration file during initialization.

3.5 Running the Application

♦ **Compile and link `tp_sr` and `tp_rs` using the `Makefile` provided.**
The SunLink P2P LU6.2 9.1 API verbs are located in the library (random archive,” `liblu62.a`.

```
% cd /opt/SUNWlu62/examples
% make
```

```
Then, with sunpu2.1 running, invoke tp_sr as follows.
% tp_sr
```

tp_sr connects to LUA and allocates a conversation with TPB located at the partner LU, PLUB, using mode MODEAB. tp_rs is invoked using the configured TP_PATH command when the allocated request is received for TPB. If tp_rs is successfully started, the programs will start sending and receiving data.

Both tp_sr and tp_rs accept a number of command line options. The tp_sr can, for example, initiate a conversation with a synchronization level of confirm. Consult the program headers for more information on the available options. In the example here, the program usage is kept as simple as possible.

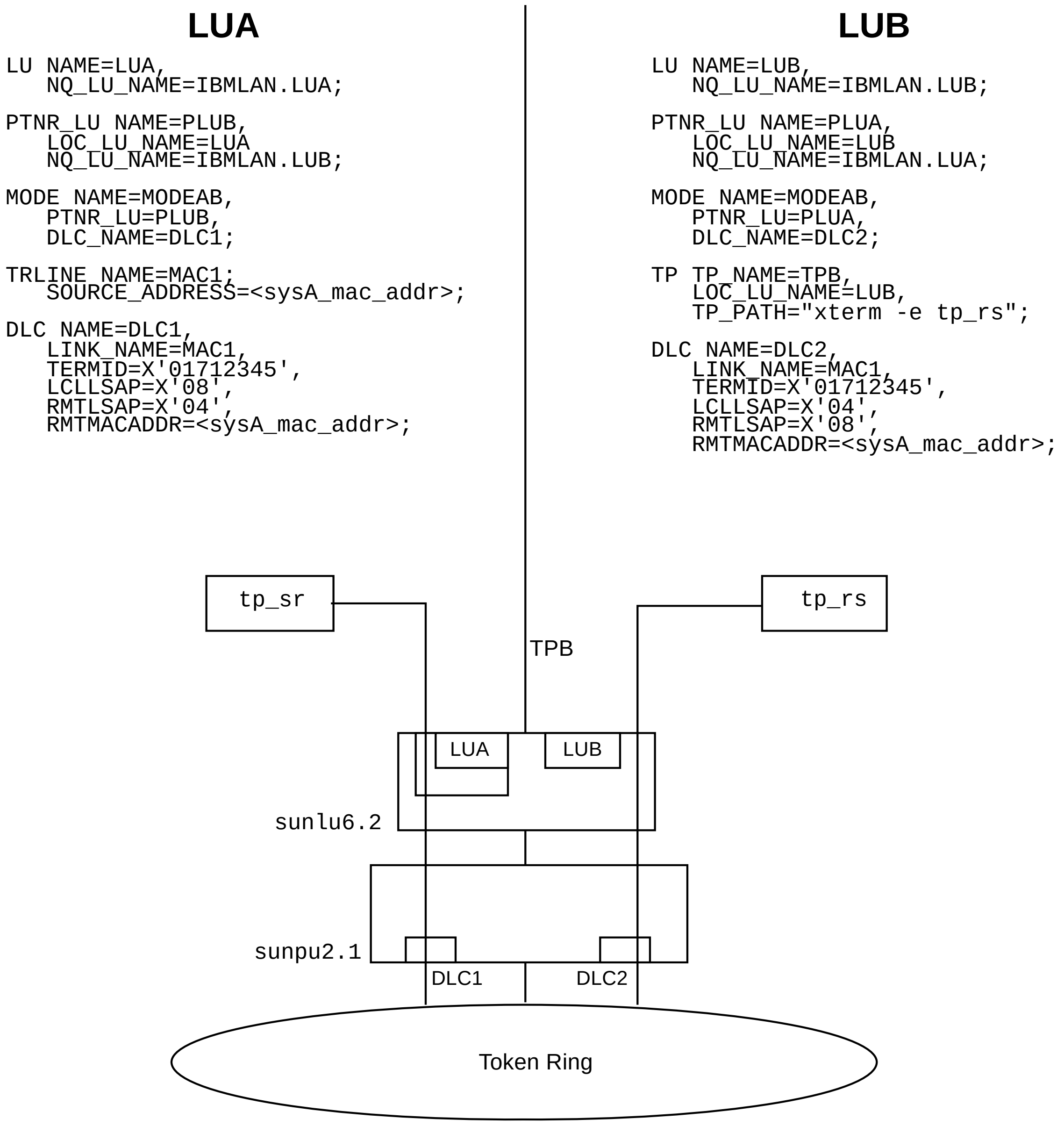
To stop the application, type Control-C to terminate tp_sr or tp_rs.

3.6 Token Ring Peer-to-Peer Configuration

Figure 3-3 depicts the example peer-to-peer Token Ring configuration. The corresponding configuration files, sunlu62.a.tr and sunlu62.b.tr, are distributed with the SunLink P2P LU6.2 9.1 product and are included in Appendix E. Figure 3-3 shows the pertinent configuration parameters.

The sunlu62.loopback.tr configuration is split between the two workstations, System A and System B. System A defines DLC1 as its data link connection with System B. Note the use of the DLC parameters RMTMACADDR, LCLLSAP, and RMTLSAP. The RMTMACADDR parameters correspond to the TRLINE SOURCE_ADDRESS of the other system.

Re-run the application as described in Section 3.5, “Running the Application.”



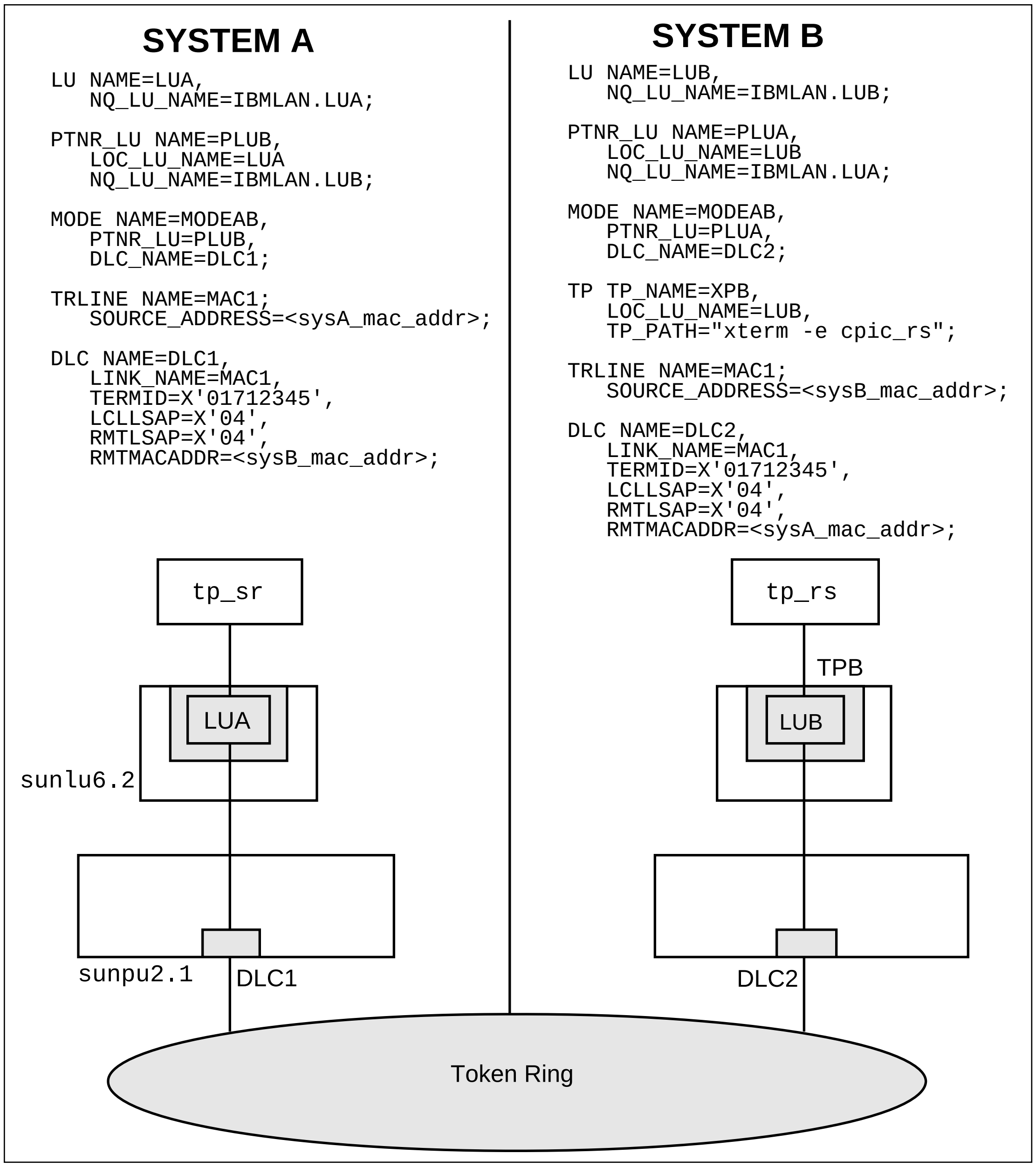
LUA

```
LU NAME=LUA,
   NQ_LU_NAME=IBMLAN.LUA;

PTNR_LU NAME=PLUB,
   LOC_LU_NAME=LUA
   NQ_LU_NAME=IBMLAN.LUB;

MODE NAME=MODEAB,
   PTNR_LU=PLUB,
   DLC_NAME=DLC1;

TRLINE NAME=MAC1;
   SOURCE_ADDRESS=<sysA_mac_addr>;

DLC NAME=DLC1,
   LINK_NAME=MAC1,
   TERMID=X'01712345',
   LCLLSAP=X'08',
   RMTLSAP=X'04',
   RMTMACADDR=<sysA_mac_addr>;
```

LUB

```
LU NAME=LUB,
   NQ_LU_NAME=IBMLAN.LUB;

PTNR_LU NAME=PLUA,
   LOC_LU_NAME=LUB
   NQ_LU_NAME=IBMLAN.LUA;

MODE NAME=MODEAB,
   PTNR_LU=PLUA,
   DLC_NAME=DLC2;

TP TP_NAME=TPB,
   LOC_LU_NAME=LUB,
   TP_PATH="xterm -e tp_rs";

DLC NAME=DLC2,
   LINK_NAME=MAC1,
   TERMID=X'01712345',
   LCLLSAP=X'04',
   RMTLSAP=X'08',
   RMTMACADDR=<sysA_mac_addr>;
```

Figure 3-2 Token Ring Loopback Configuration



```
LU NAME=LUA,
   NQ_LU_NAME=IBMLAN.LUA;

PTNR_LU NAME=PLUB,
   LOC_LU_NAME=LUA
   NQ_LU_NAME=IBMLAN.LUB;

MODE NAME=MODEAB,
   PTNR_LU=PLUB,
   DLC_NAME=DLC1;

TRLINE NAME=MAC1;
   SOURCE_ADDRESS=<sysA_mac_addr>;

DLC NAME=DLC1,
   LINK_NAME=MAC1,
   TERMID=X'01712345',
   LCLLSAP=X'04',
   RMTLSAP=X'04',
   RMTMACADDR=<sysB_mac_addr>;
```

```
LU NAME=LUB,
   NQ_LU_NAME=IBMLAN.LUB;

PTNR_LU NAME=PLUA,
   LOC_LU_NAME=LUB
   NQ_LU_NAME=IBMLAN.LUA;

MODE NAME=MODEAB,
   PTNR_LU=PLUA,
   DLC_NAME=DLC2;

TP TP_NAME=XPB,
   LOC_LU_NAME=LUB,
   TP_PATH="xterm -e cpic_rs";

TRLINE NAME=MAC1;
   SOURCE_ADDRESS=<sysB_mac_addr>;

DLC NAME=DLC2,
   LINK_NAME=MAC1,
   TERMID=X'01712345',
   LCLLSAP=X'04',
   RMTLSAP=X'04',
   RMTMACADDR=<sysA_mac_addr>;
```

Figure 3-3 Token Ring Peer-to-Peer Configuration

3.7 SunLink PU2.1 Monitoring and Control

Use the `sunop` application to display the status of the SunLink SNA PU2.1 9.1 server data links. The `sunop` application prompts you for management requests (`->` is the `sunop` prompt).

To view the PU2.1 status:

1. **Enter `dis` (display status), as follows:**

```
% sunop
(0) dis
->
OP200038 : (0) Control Point SUNCP
OP200039 : (0)   Independent LU LUA - 1 active sessions
OP200025 : (0) Link MAC1 - (2) Active
OP20001e : (0)   Physical Unit/DLC DLC1 - (5) Pending
Active/Contacted
```

2. **Exit from `sunop` by typing `quit` at the `sunop` prompt.**

3.8 Using `sunscope`

The SunLink SNA PU2.1 9.1 server is distributed with `sunscope`. `sunscope` is a logical data scope, which allows you to monitor the data sent and received via a physical device driver, in this case, the Token Ring network interface device, `/dev/nit` (see `TRLINE DEVICE` parameter). Once `sunpu2.1` is running, use the `sunscope` command as follows:

```
# sunscope -t -e -d /dev/zbxa
```

The `-t` option is required for Token Ring devices. The `-e` option causes printable data to be displayed in EBCDIC. Refer to the *SunLink SNA PU2.1 9.1 Server Configuration and Administration Manual* for more information on the `sunscope` command and its use.

Configuration 4

This chapter concentrates on the former directives and describes how to configure your system for LU6.2 with respect to host and peer nodes within the SNA network. A complete description of the configuration directives, parameters, and arguments is found in the *SunLink SNA PU2.1 9.1 Server Configuration and Administration Manual*. The purpose of this chapter is to highlight certain configuration issues and to describe the SunLink P2P LU6.2 9.1 security model. When applicable, configuration parameters for the following systems are discussed:

- VTAM
- CICS
- AS/400

The SunLink SNA PU2.1 9.1 server configuration specifies the operating characteristics of the local SNA resources with respect to LU6.2. Of particular importance to the LU6.2 application programmer are the logical resources described in Table 4-1.

Table 4-1 Logical Sources

Resource	Description
`LU`	Logical unit
`PTNR_LU`	Partner LU
`MODE`	Mode
`TP`	Transaction Program
`SECURITY`	LU security access information
`SEC_ACCESS`	TP Security Resource Access Information

Note – If the TP configuration does not specify an LU, it is available to all LUs.

The resources shown below are also important, but mainly for physical connectivity rather than LU6.2 programming.

Table 4-2 Physical Connectivity Logical Sources

Resource	Description
`CP`	Control point
`SDLCLINE`	SDLC Serial Link
`QLLCLINE`	X.25 PDSN Link
`TRLINE`	Token Ring LAN Link
`LLC`	Logical Link Control for LAN Links

4.1 Resource Definition

The following subsections provide definitions for various resources.

4.1.1 Logical Unit (LU)

The LU configuration directive defines an LU6.2 entry point from the local node into the SNA network. When connecting to an SNA host (i.e., VTAM or CICS) either dependent and independent LUs may be specified. A dependent LU requires SSCP assistance to establish a session and must act as a secondary, single-session LU. Independent LUs require no SSCP intervention and can be a primary LU that supports parallel sessions. Independent LUs, normally used for connecting with peer PU2.1 systems such as an AS/400 or a PC running APPC/PC, can also be connected directly into an IBM host. From the perspective of the LU6.2 programmer, only the name of the LU is relevant, since the other aspects are controlled by the configuration.

4.1.2 VTAM

The SunLink LU configuration is matched against the VTAM LU when it is connected with an IBM host.

♦ **For a *dependent* LU configure as follows:**

```
LOCADDR=n where n is a non-zero value representing the local
address.
```

A dependent LU must be associated with a specific DLC (formerly PU2) directive in the SunLink configuration file. Network or fully-qualified names, which consist of an optional network identifier period, separated from a node identifier, can be used and are defined with the `NQ_LU_NAME` parameter as follows:

```
NQ_LU_NAME=[NETID.]NODEID
```

Note – When connecting to a host as a dependent LU, the uninterpreted LU name, `UI_LU_NAME`, should be set to the host configured value to ensure proper SSCP translation.

Independent LUs connecting to a host are not associated with a specific local address and must be specified as follows:

```
LOCADDR=0
```

There is no corresponding PU as with the case of a dependent LU. Instead, independent LUs are, by default, associated with the local control point (CP). Both the CP and LU have network-qualified names, of which the network id prefix must be identical. This network id prefix must also match what is defined for the network. Remember that the host may also connect to SunLink PU2.1 as a peer and use independent LUs. In this case, the PU definition in VTAM must include the following:

```
PUTYPE=2
XID=YES
```

Consult the *SunLink SNA PU2.1 9.1 Configuration and Administration Manual* for more information on PU definition.

4.1.3 CICS

When connecting with CICS for local LU definition, aspects of the connection definition must be examined. Specifically, for connection definition, look for the following:

Code Example 4-1

```
CONNECTION IDENTIFIERS
         NETNAME
         INDSYS

REMOTE ATTRIBUTES
         REMOTESYSTEM
         REMOTENAME

CONNECTION PROPERTIES
         ACCESSMETHOD:   VTAM
         PROTOCOL:       APPC
```

```
CONNECTION IDENTIFIERS
         DATASTREAM:     USER
         RECORDFORMAT:   U
```

To define the Terminal Control Table, DFHTCT, with respect to the Local LU definition, use the following parameters:

Code Example 4-2

```
TRMTYPE=LUTYPE62
SYSIDNT=idname
NETNAME=netname
```

4.1.4 AS/400

When connecting to an AS/400 for local LU definition, the Line Description, Controller Description, and Device Descriptions must be modified. Verify the following parameters:

Code Example 4-3

```
CRTCLTAPPC
    RMTCPNAME=remote-control-point-name
    RMTNETID=remote-network-identifier
    AUT=authorization-list-name
```

4.2 Partner Logical Unit (PTNR_LU)

The PTNR_LU configuration directive defines type 6.2 LUs on remote systems that are accessible to a locally-defined LU6.2. Each remote LU6.2 has a name that is locally known. Using locally known LU names allows network reconfiguration to be transparent to the LU6.2 programmer. The partner LU can reside on either a host or a peer node. There are some minor differences in defining partner LUs on each of these alternate node types. These differences are highlighted later in this chapter.

4.2.1 VTAM

Partner LUs on the host follow the model that was presented for LUs in the previous section. Partner LUs for local dependent LUs must be primary, single-session LUs, in contrast to the independent LUs, which may be either primary or secondary LUs that support parallel sessions. VTAM defines resources that are viewed as remote LUs by SunLink P2P LU6.2 9.1 with the APPL resource definition directive. Parameters of particular interest are shown below.

Code Example 4-4

```
APPC
PARSESS   =YES         for independent LUs
           =NO         for dependent LUs
```

Other parameters in the APPL resource definition directive are used for LU6.2, and are discussed in the following subsections.

4.2.2 CICS

When connecting with CICS for partner LU definition, aspects of the connection definition must be examined. For connection definition see the example below.

Code Example 4-5

```
CONNECTION PROPERTIES
SINGLESESS
SECURITY
BINDPASSWORD
```

The following parameters are relevant for defining the terminal control table, DFHTCT, with respect to the partner LU definition:

Code Example 4-6

```
FEATURE   =SINGLE
          =PARALLEL
BINDPWD=xxxxxxxxxxxxxxxx
```

`BINDPWD` is the password used to encrypt and decrypt random data on both the BIND and its associated response. This field is matched against the `LU_LU_PASS` parameter in the `PTNR_LU` configuration directive.

4.2.3 AS/400

When connecting to an AS/400 for partner LU definition, the line description, controller description, and device descriptions must be modified. Verify that the parameters shown in the code example below are present.

Code Example 4-7

```
CRTDEVAPPC
    LCLLOCNAME=local-location-name
    SNGSSN=number-of-conversations
    LOCPWD=location-password
    SECURELOC=YES|NO
    AUT=authorization-list-name
```

4.2.4 Mode (MODE)

The Mode configuration directive defines the characteristics of sessions between logical units. The mode, along with the partner LU and the transaction program name, is specified on the various types of ALLOCATE requests. The mode name must be common to both the local and partner LUs.

Note – In addition to the values described in the *SunLink SNA PU2.1 9.1 Configuration and Administration Manual*, you may specify a `UNIQUE_SESSION_NAME` parameter with LU6.2 that is used to specify the node in the allocate and listen verbs.

4.2.5 VTAM

Mode entries are defined using the MODEENT macroinstruction. Parameters of particular interest are shown below.

Code Example 4-8

```
FMPROF=X'13'
TSPROF=X'07'
SSNDPAC=n
SRCVPAC=n
RUSIZES=n
```

4.2.6 CICS

When connecting with CICS for mode definition, aspects of the session definition must be examined. Verify the following for the session definition as shown below.

Code Example 4-9

```
OBJECT CHARACTERISTICS

SESSION_IDENTIFIERS
    MODENAME
    SENDSIZE
    RECEIVESIZE
```

For defining the Terminal Control Table, DFHTCT, with respect to the mode definition, the parameters shown below are relevant.

Code Example 4-10

```
MODENAME=mode-name
FEATURE=features
RUSIZE=n
```

4.2.7 AS/400

When connecting to an AS/400 for mode definition, the line description, controller description, and device descriptions must be modified. Verify that the parameters shown below are present.

Code Example 4-11

```
CRTDEVAPPC
    MODE=mode-name

MODE DESCRIPTION
    MODD=mode-name
    COS=class-of-service-name
    MAXSSN=maximum-sessions
    MAXCNV=maximum-conversations
    LCLCTLSSN=locally-controlled-sessions
    PREESTSSN=pre-established-sessions
    INPACING=inbound-pacing-value
    OUTPACING=outbound-pacing-value
    MAXLENRU=maximum-length-of-request-unit
```

4.3 Transaction Program (TP)

The TP configuration directive defines transaction programs to which incoming allocate requests may be directed. Access to TPs may be restricted depending upon both LU access security and TP resource access security. Security issues are explained in both the following subsections and in Section 4.4, "Security." Note that TPs on remote systems are not configured to the local system. The selection of the remote TPs is made by the LU6.2 program so you should know how TPs are defined on the remote system.

4.3.1 CICS

CICS transaction programs are defined by the processing program table, `DFHPPT`, and the program control table, `DFHPCT`, statements.
Code Example 4-12 defines a CICS program, `PRG1`, which supports two transactions, TR100 and TR101. These `TRANSID` names correspond to the `remote_tp_name` parameter to `lu62_allocate` (8.1) or `lu62_mc_allocate` (9.1) verbs.

Code Example 4-12

```
****************************************************************
,” PPT: define PRG1
****************************************************************
           DFHPPT TYPE=ENTRY,
           PROGRAM=PRG1,
           PGMLANG=ASSEMBLER,

****************************************************************
,” PCT: define PRG1 transactions
****************************************************************
           DFHPCT TYPE=ENTRY,
           TRANSID=TR100,
           PROGRAM=PRG1,
           DTB=(NO),

           DFHPCT TYPE=ENTRY,
           TRANSID=TR101,
           PROGRAM=PRG1,
           DTB=(NO),
```

4.3.2 LU Access Security Information (SECURITY)

The `SECURITY` configuration directive defines Access Security information for the local LU. If Access Security is included, only authorized users are allowed access to the services of this LU. Further restrictions on access to the TPs can be done through TP resource access and partner LU security acceptance (`SEC_ACCEPT`). Inclusion of the `SECURITY` directive for a particular LU creates a Security Access list for that LU. The section on security at the end of this chapter will discuss LU security access in greater detail.

4.3.3 TP Resource Access Security Information (`SEC_ACCESS`*)*

The `SEC_ACCESS` configuration directive defines resource access security information for the local LU. Resource Access security further restricts access to the local TP, even if `SECURITY` checks are enabled. Inclusion of the `SEC_ACCESS` directive for a particular TP creates a resource access list for that LU. The section on security at the end of this chapter discusses TP resource access in greater detail.

4.4 Security

The SunLink LU6.2 security paradigm is a multi-tiered access model that provides LU6.2-based and Unix-based security. Sun has integrated Unix security into the model to more closely follow the restriction put in place by the Unix system administrator.

4.4.1 Session-Level Security

Prior to any transaction program security access, session(s) must first be established between the communicating logical units. Session verification between two LUs occurs during BIND processing by the use of LU-LU passwords. Specifically, if session-level security will be used for BIND verification, then both sides must have the operation enabled. LU-LU verification is done with the specification of the `LU_LU_PASS` parameter on the `PTNR_LU` configuration directive, as shown in Code Example 4-13.

Code Example 4-13

```
PTNR_LU    Name=SECURELU,
           ...
           LU_LU_PASS=X'23AF9006DD71',
           ...;
```

Verification of the BIND and response to the BIND is internal to SunLink LU6.2 and is not visible to the LU6.2 programmer. Consult the LU6.2 architectural specifications for details on session-level security.

4.4.2 Conversation-Level Security

Conversation level security occurs after a session is established between two LUs. Whether the two LUs have used LU-LU verification while a session was established is not important for conversation-level security. Conversation-level security occurs when an allocation request is received by the partner LU. An allocation request is generated when a program issues an `(MC_)ALLOCATE` verb or a `CPI-C Allocate` (`CMALLC`) call. The request is transmitted as an SNA Request Unit (RU) containing a Function Management Header Type 5 (FMH-5), also known as an attach. Allocation requests can also occur as a result of an explicit or implicit Change Number of Session (CNOS) request. The attach structure can contain the access security information as described in Table 4-1.

Table 4-3 Access Security Information

Parameter	**Description**
`USER_ID`	User identification name
`PASSWORD`	User password
`PROFILE`	User profile characteristics
`ALREADY_VERIFIED`	Flag for password identification conducted by local LU

`ALREADY_VERIFIED` signifies that the identity of the user was previously verified by the local LU, the assumption being that the partner LU relies on the local LU. No password is sent in this case. The operation is specified by the `PTNR_LU` configuration directive as described in Code Example 4-14.

Code Example 4-14

```
PTNR_LU     Name=ALRVERLU,
          ...
          SEC_ACCEPT=ALREADY_VERIFIED,
          ...;
```

This operation is supported through BIND processing and its associated response. The use of the other security access parameters, `USER_ID`, `PASSWORD`, and `PROFILE`, with respect to conversation-level security and Unix-level security is shown in Figure 4-1.

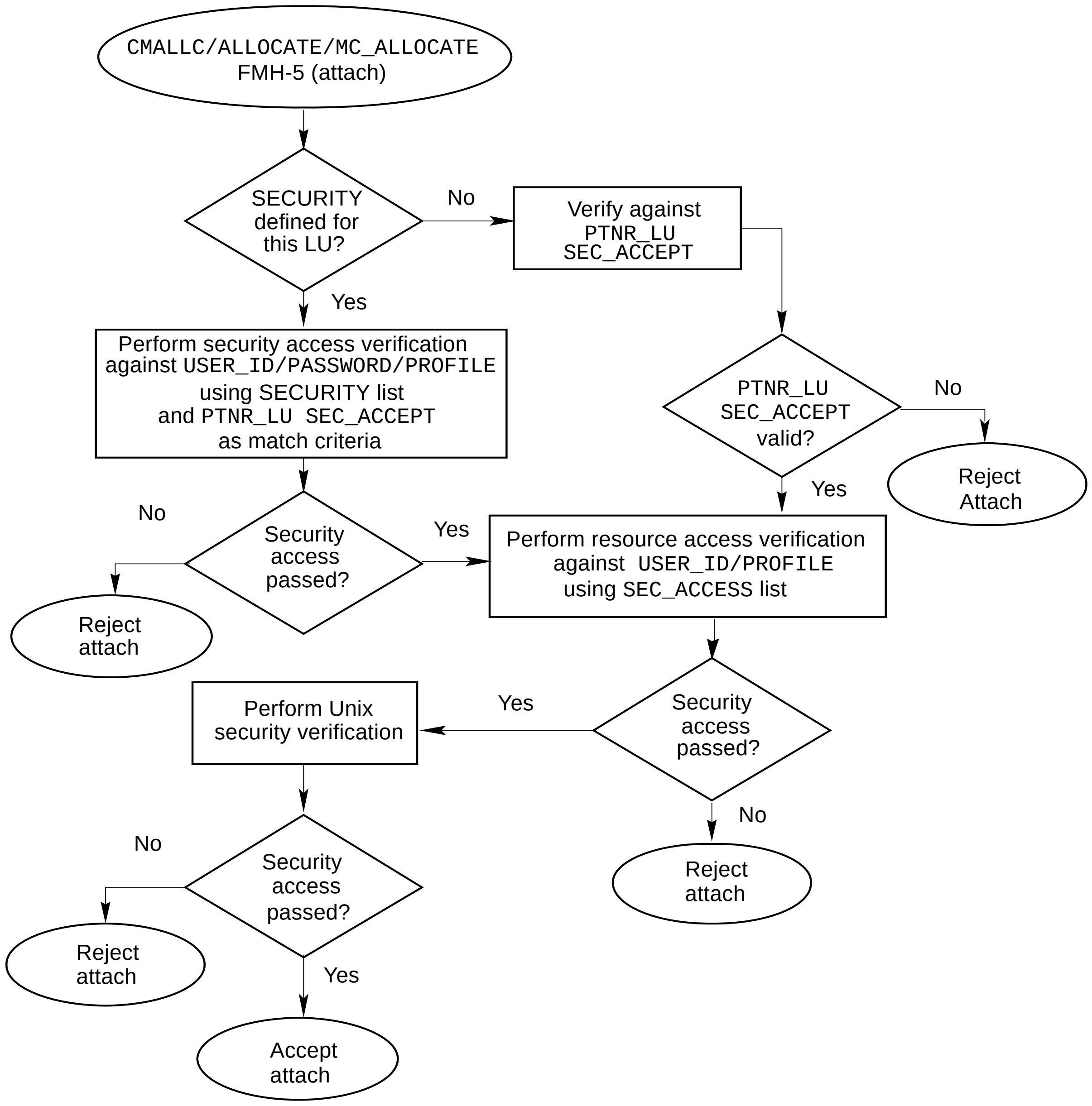

CMALLC/ALLOCATE/MC_ALLOCATE
FMH-5 (attach)
SECURITY defined for this LU?
No
Verify against PTNR_LU SEC_ACCEPT
Yes
Perform security access verification against USER_ID/PASSWORD/PROFILE using SECURITY list and PTNR_LU SEC_ACCEPT as match criteria
PTNR_LU SEC_ACCEPT valid?
No
Reject Attach
Yes
No
Security access passed?
Yes
Reject attach
Perform resource access verification against USER_ID/PROFILE using SEC_ACCESS list
Perform Unix security verification
Yes
Security access passed?
No
Reject attach
No
Security access passed?
Reject attach
Yes
Accept attach

Figure 4-1 SunLink LU6.2 Security Processing

If the attach contains security information, and the LU has a security restriction, then parameters in the attach are matched against those in the LU security list. There is one security list per LU and entries are created by the SECURITY configuration directive. An example of such a directive is described in Code Example 4-15.

Code Example 4-15

```
SECURITY      LOC_LU_Name=SECURELU,
                  USER_ID=USER1,
                  PASSWORD=PASSXXX,
                  PROFILE=ACCTRCV;...
```

In addition to zero or more PROFILEs, multiple directives may be defined per LU. PASSWORDs on subsequent directives for the same user and LU are optional and if additional PASSWORDs are supplied, its definition will replace the last one defined.

The LU security list is searched using the SEC_ACCEPT parameter of the PTNR_LU configuration directive. If NONE is specified, then the request will be rejected if any security parameters are present. If CONVERSATION is specified, then any request containing security, but not an already-verified indicator will be searched for a match. A match will be successful, if an entry in the security list matches all the security fields in the attach, otherwise the request will be rejected. If the match is successful, additional verification may be done; this is described later. Finally, if ALREADY_VERIFIED is configured, then the attach will be checked for not only conversation, but also for already-verified state, in place of a password.

If no security parameters are defined for this LU, none are located in the attach, and the partner LU expects no security. The attach is then passed to resource verification; otherwise, it is rejected.

Resource access verification is conducted after security access; it is a restriction on access to the transaction program. Resource access verification is conducted against the TP resource access list. There is one resource access list per TP and entries are created by the SEC_ACCESS configuration directive. An example of such a directive is given in Code Example 4-16.

Code Example 4-16

```
SEC_ACCESS    LOC_LU_Name=SECURELU,
              TP_NAME=SECURETP,
              USER_ID=USER1,
              PROFILE=ACCTPAY,
              PROFILE=ACCTRCV;
```

In addition to zero or more PROFILEs, multiple directives may be defined per TP. PASSWORDs are not specified for resource access. Resource access checking is conducted if any of the following arguments are specified for the SEC_REQUIRED parameter in the TP configuration directive: USER_ID, PROFILE, or USER_ID_PROFILE. If a value of CONVERSATION is specified for this field, then security access is verified, but not the resource access. If a value of NONE is specified for this field, then no verification is required. If the security parameters are received, however, CONVERSATION checks need to be conducted. If any of these checks fail, then the request is rejected; otherwise an additional set of tests is conducted.

4.4.3 Unix-Level Security

The final set of security checks is a Sun enhancement to the LU6.2 architecture for security management on Unix systems, known as Unix-level security. Unix-level security has two modes. The first, the default mode, occurs when UNIX_SEC is set to a value of NO. Specifically, this controls how processes are invoked from remote systems onto the local system. SunLink LU6.2 does not allow processes to be invoked as root unless it is explicitly instructed to do so as described later.

The effective user id for SunLink LU6.2 under Unix is sunlu62. This value must be added to /etc/passwd at installation time. Unlike other system files, which are automatically modified when the Sun product is installed, /etc/passwd (or any associated NIS password database) is not modified for security reasons. If you try to invoke a TP through an attach and /etc/passwd has not been updated, the attach will be rejected and the session is brought down with a security violation sense code.

The default user id for TP process invocation can be changed from sunlu62 to a user-specified value by modifying the CP configuration directive. This is also true of the group id, which normally gets its default value from /etc/passwd. The CP directive would be modified as shown in Code Example 4-17.

Code Example 4-17

```
CP      NAME=SUNCP,
        NQ_CP_NAME=IBMLAN.SUNCP,
        USER=sunadm,
        GROUP=sungrp,
        UNIX_SEC=NO;
```

If `sunadm` and `sungrp` are properly defined in `/etc/passwd` and `/etc/group`, then programs invoked on this machine will have the corresponding user and group id. If there are any misconfigurations, the attach will be rejected and the session will be terminated. Error messages will also appear on the operator console to indicate that such an action is occurring.

The second mode of Unix security involves using the user id, password, and profile from the attach to map onto Unix system parameters. Such processing occurs when the `UNIX_SEC` parameter of the `CP` configuration directive is set to `YES`. The attach parameters are matched against `/etc/passwd` and `/etc/group` as the final check. If there is verification, then the TP process is invoked with:

- Effective user id = `Attach`*(USER_ID)*
- Effective group id = `Attach`*(PROFILE)*

An example `CP` directive that supports such an operation is shown in Code Example 4-18.

Code Example 4-18

```
CP      NAME=SUNCP,
        NQ_CP_NAME=IBMLAN.SUNCP,
        USER=sundeflt,
        UNIX_SEC=YES;
```

The `USER` specified is the default when there is no security present on the attach. If there are any violations, the attach will be rejected and the session will be brought down. Error messages will also appear on the operator console to indicate that such an action is occurring. Note that in this mode of security processing, a `PASSWORD` value is not configured in the Security Access List since the password on the attach is verified against `/etc/passwd.`

Using the LU6.2 API 5

This chapter describes the SunLink P2P LU6.2 9.1 API. It shows how to organize your program by providing examples and illustrates how the API is used to perform standard LU6.2 operations. It also shows how the advanced features of the SunLink P2P LU6.2 9.1 API can be used to implement more powerful programs.

5.1 Call Conventions

User programs make simple function calls to issue SunLink P2P LU6.2 9.1 API verbs. To use the API you must include the include file, `sunlu62.h`, in your program files and link with the SunLink P2P LU6.2 9.1 library, `liblu621.a`. `sunlu62.h` is listed in Appendix C, "LU 6.2 Include Files."

The SunLink P2P LU6.2 9.1 API provides a consistent function call interface. The following example of `lu62_prep_to_receive` is typical.

Code Example 5-1

```
#include “sunlu62.h”

    extract from sunlu62.h...

    typedef enum {
        PR_SYNC_LEVEL = 0,
        PR_FLUSH,
        PR_CONFIRM
    } lu62_prep_to_receive_type_e;

    typedef struct
        bit32 conv_id;                              ,“ s ,“
        lu62_prep_to_receive_type_e type;           ,“ s ,“
        bit32 return_code;                          ,“ r ,“
    lu62_prep_to_receive_t;

    end of extract

...



lu62_prep_to_receive_t *rqp;
bit32 conv_id;
int rc;


,” Issue PREPARE_TO_RECEIVE ,“
rqp = (lu62_prep_to_receive_t ,“
        calloc(1,sizeof(lu62_prep_to_receive_t));
rqp->conv_id = conv_id;
rqp->type = PR_FLUSH;
rc = lu62_prep_to_receive(rqp);
if (rc ,“ LU62_ERROR)
    errmsg(“lu62_prep_to_receive error = %x\n”, rqp-
>return_code);
    exit(1);
```

Note the following regarding Code Example 5-1:

- All verbs (except certain connection verbs) have an associated request structure such as `lu62_prep_to_receive_t`. Some request structures are more complicated than others. Request structure fields are either supplied inputs (`/* s */`), returned values (`/* r */`), or both (`/* sr */`).
- Mapped and basic verbs share the same request structures. Fields that are only applicable to one conversation type are ignored by the other.
- The request structure is initialized to zero before it is used (in this case by `calloc`). First initialize the request structures to zero and then set them with the required parameter values. This ensures that pointer values are initialized to NULL and enumerated type values are defaulted. Enumerated types are defined so that the default value is zero.
- Enumerated types such as `lu62_prep_to_receive_type_e` are used extensively to describe field values.
- All verbs return an integer value, `LU62_OK` or `LU62_ERROR`. If `LU62_ERROR` is returned, the external variable, *lu62_errno* contains the reason for the error.
- All verb request structures include a `return_code` field that is set to the result of the operation. When the return value is `LU62_ERROR`, the `return_code` contains the same value as *lu62_errno*. `return_code` values are defined in `sunlu62.h`.
- `conv_id` is the handle used to identify the conversation. `conv_id` is assigned by the API when a conversation is allocated by `lu62_(mc_)allocate` or accepted by `lu62_accept`.

5.2 Handling Connections to the SunLink PU2.1 SNA Server

Your program operates as a separate process from the SunLink SNA PU2.1 9.1 server. Before you can issue SunLink P2P LU6.2 9.1 verbs you must establish a connection to a SunLink SNA PU2.1 9.1 server using the `lu62_open` connection verb. When your program has completed communications with its peers, it should disconnect from the server using the `lu62_close` verb. Be sure to use `lu62_open` and `lu62_close` to “bracket” your program as shown in Code Example 5-2.

Code Example 5-2

```
main ,“
{
    lu62_open(&open_req);

        API calls
        ...

    lu62_close(&close_req);

        ...
}
```

5.2.1 Connecting to the SunLink PU2.1 SNA Server

To open a connection, the user program sets up an open request data structure specifying the name of the local LU to which it wants to be associated. The open request can also be used to specify the SunLink PU2.1 SNA server host, that is, the workstation in the local area network on which the server is running. This open request is passed to the API in the `lu62_open` call as illustrated in the Code Example 5-3.

Code Example 5-3

```
lu62_open_req_t open_req;

,” initialize open request to default values ,“
bzero(&open_req, sizeof(struct lu62_open_req));

strncpy(open_req.host, LU62_SERVER, MAXHOSTNAMELEN);
strncpy(open_req.lu_name, LOCAL_LU, LU62_LU_NAME_LEN);

if (lu62_open(&open_req) ,“ LU62_ERROR) {
    printf(“lu62_open error, 0x%x\n”, lu62_errno);
    exit(1);
port_id = open_req.port_id; ,“ save returned port_id ,“
```

5.2.2 Disconnecting from the SunLink PU2.1 SNA Server

A user program relinquishes a connection to the SunLink SNA PU2.1 9.1 server by calling `lu62_close`. Active conversations should first be deallocated, otherwise, the server will deallocate them on behalf of the program. The `lu62_close` call is issued as follows, where `port_id` is the value returned from the corresponding `lu62_open` request.

Code Example 5-4

```
lu62_close_req_t close_req;

close_req.port_id = port_id;

if (lu62_close(&close_req) ,“ LU62_ERROR) {
    printf(“lu62_close error, 0x%x\n”, lu62_errno);
    exit(1);
}
```

5.3 Allocating Conversations

LU6.2 programs initiate conversation with remote TPs using the `lu62_(mc_)allocate` verb. The verb specifies the `port_id` of the previously opened LU connection. If successful, it returns a conversation identifier, `conv_id`, which is used as the handle for all subsequent conversation verbs. Note that multiple conversations may be supported on the same LU connection.

The allocate request structure, `lu62_allocate_t`, specifies the location of the remote TP as follows:

- `lu_name`, specifies the locally known name of the partner LU at which the remote TP resides. This parameter corresponds to the `PTNR_LU NAME` parameter in the configuration.
- `mode_name`, identifies the characteristics of the required session. This value corresponds to the `MODE NAME` parameter in the configuration.
- `remote_tp_name`. This value corresponds to the `TP TP_NAME` parameter (or its equivalent) in the remote LU6.2 configuration.

As noted, these input parameters correspond to values in the configuration file. See Chapter 4, “Configuration,” for more information.

In the Code Example 5-5, the program initiates a CICS reservation update transaction, RESUPD. The conversation is mapped, `sync_level` is confirmed, and security access parameters are provided. Default values (of 0) are used for all other parameters.

Code Example 5-5

```
struct lu62_allocate_t allocate_req;

,” initialize allocate request with default values ,“
bzero(&allocate_req, sizeof(struct lu62_allocate_req));

,” specify the location of the remote TP ,“
strncpy(allocate_req.remote_tp_name, “RESUPD”,
LU62_TP_NAME_LEN);
strncpy(allocate_req.lu_name, “CICS2174”, LU62_LU_NAME_LEN);
strncpy(allocate_req.mode_name, “CRESUPD”, LU62_MODE_NAME_LEN);
```

Code Example 5-5 (Continued)

```
,” security information is supplied by the program ,“
allocate_req.security_type = SECURITY_PROGRAM;
strncpy(allocate_req.user_id, “USER”, LU62_MAX_USER_ID_LEN);
strncpy(allocate_req.passwd, “PASSWORD”, LU62_MAX_PASSWD_LEN);

,” conversation sync level ,“
allocate_req.sync_level = SYNC_LEVEL_CONFIRM;

if (lu62_mc_allocate(&allocate_req) ,“ LU62_ERROR) {
    printf(“lu62_allocate error, 0x%x\n”, lu62_errno);
    exit(1);

}
,” transaction continues ,“
```

5.4 Accepting Conversations

In the LU6.2 model, the logical unit initiates a local transaction program when an incoming allocation request is received. The transaction program can subsequently allocate further conversations, but it cannot receive any more allocation requests. SunLink P2P LU6.2 9.1 programs, however, can accept multiple conversations.

When an incoming conversation arrives, there are three ways to dispatch the transaction to a SunLink P2P LU6.2 9.1 program:

- Your program can be started (fork and exec) by the SunLink SNA PU2.1 9.1 server using the `TP TP_PATH` parameter. The server sets the UID and GID of the client process to that of the configured `CP USER` (default `sunlu62`). This approach is best for programs that handle a single transaction, run to completion, and then exit.
- Your program can be started ahead of time while you wait for an incoming conversation. Typically this approach is taken when a program accepts and processes multiple conversations concurrently.

Note – Allow the server to invoke the program when the first conversation is received. Your program should not fork a child to process the transaction (or wait for the next conversation). The SunLink P2P LU6.2 9.1 API maintains open file descriptors and context structures for each connection, and maintains context structures for each accepted conversation. If you want to build a Unix server of this nature, use the `lu62_listen` verb.

- You can build a custom transaction dispatcher using the `lu62_listen` verb (see Section 5.5, "Transaction Dispatch Using lu62_listen.")
- The behavior of your program is exactly the same in each case; it should perform its initialization, register a TP name with its local LU using `lu62_register_tp`, and then issue `lu62_accept`. The registered TP name is the same name used by the remote TP when it issues its ALLOCATE verb. It must be configured in the SunLink SNA PU2.1 9.1 server configuration. `lu62_accept` returns the conversation's identifier, `conv_id`, which is used as the handle for all subsequent conversation verbs.

In the Code Example 5-6 that follows, the program is set up to accept an incoming conversation for TP "FRED."

Code Example 5-6

```
lu62_accept_t accept_req;
lu62_register_tp_t register_req;

,” Register with LU as TP FRED ,“
register_req.port_id = open_req.port_id;
strncpy(register_req.tp_name, “FRED”, LU62_TP_NAME_LEN);
if (lu62_register_tp(&register_req) ,“ LU62_ERROR) {
    printf(“lu62_register_tp error, 0x%x\n”, lu62_errno);
    exit(1);

}
,” Wait for incoming conversation ,“
accept_req.port_id = open_req.port_id;
if (lu62_accept(&accept_req) ,“ LU62_ERROR) {
    printf(“lu62_accept error, 0x%x\n”, lu62_errno);
    exit(1);
```

Code Example 5-6

```
lu62_accept_t accept_req;

}
process_transaction(accept_req.conv_id);
```

5.4.1 Accepting Multiple Conversations

The program can accept multiple conversations for its TPs. Assuming the same set-up as above, this program extract accepts incoming conversations and processes the transaction. When the transaction is complete, the program waits for another conversation. In this example, the transactions are processed sequentially. In the next section, non-blocking operations are used to enable a program to process multiple transactions concurrently.

Code Example 5-7

```
,” Listen for incoming conversation ,“
accept_req.port_id = open_req.port_id;
while (lu62_accept(&accept_req) ,“ LU62_OK) {
    ,“ conversation accepted ,“
    process_transaction(accept_req.conv_id);
}
```

5.4.2 Accepting Conversations for Multiple TPs

Programs may register with the SunLink SNA PU2.1 9.1 server as more than one TP. In the example below, the program is set up to accept incoming conversations to two TPs. When a conversation is accepted, `lu62_get_tp_properties` is issued to determine which TP the conversation is for.

See Code Example 5-8.

Code Example 5-8

```
lu62_accept_t accept_req;
lu62_register_tp_t register_req;
lu62_get_tp_properties_t properties;

,” Register with LU as TP FRED ,“
register_req.port_id = open_req.port_id;
strncpy(register_req.tp_name, “FRED”, LU62_TP_NAME_LEN);
if (lu62_register_tp(&register_req) ,“ LU62_ERROR) {
    printf(“lu62_register_tp error, %0x%x\n”, lu62_errno);
    exit(1);

}
,” Register with LU as TP BARNEY ,“
register_req.port_id = open_req.port_id;
strncpy(register_req.tp_name, “BARNEY”, LU62_TP_NAME_LEN);
if (lu62_register_tp(&register_req) ,“ LU62_ERROR) {
    printf(“lu62_register_tp error, %0x%x\n”, lu62_errno);
    exit(1);

}
,” Listen for incoming conversation ,“
accept_req.port_id = open_req.port_id;
while (lu62_accept(&accept_req) ,“ LU62_OK) {
    ,“ conversation accepted - which TP ? ,“
    properties.conv_id = accept_req.conv_id;
    (void)lu62_get_tp_properties(&properties);

    ,“ dispatch ,“
    if (strcasecmp(properties.tp_name, “FRED”) ,“ 0) {
        process_transaction(fred, accept_req.conv_id,
&properties);

    }
    else {
        process_transaction(barney, accept_req.conv_id,
&properties);
    }
    }
```

5.5 *Transaction Dispatch Using* `lu62_listen`

`lu62_listen` is used much like `lu62_accept`, except that the SunLink SNA PU2.1 9.1 server does not allocate the conversation to the listening program. Instead, the listener is informed that a conversation (FMH5 Attach) has arrived and can then dispatch the transaction as required. It could fork and execute a child, or enqueue a message to the program transaction queue. The listen response contains all the conversation attributes and TP properties, including the *own_tp_instance* value that the SunLink SNA PU2.1 9.1 server assigns to the TP instance. The eventual acceptor issues `lu62_register_tp` and `lu62_accept` as usual, but passes the *own_tp_instance* value to `lu62_accept` to receive that TP instance.

If a child process is forked and executed to accept the conversation, the forked child should close the file descriptor for its connection to the local LU before executing as shown in Code Example 5-9.

Code Example 5-9

```
if (fork() ,“ 0) {
    ,“ child continues ,“
    close(listen_req.port_desc);
    ...
    exec();
}
```

In the Code Example 5-10, a listener program registers a wild-card TP name to dispatch all transactions in a program suite. A wild card TP name is configured in the SunLink SNA PU2.1 9.1 server configuration as described in the next section.

Code Example 5-10

```
TP  TP_Name=TR10?
    LOC_LU_NAME=LUA
    CONV_TYPE=MAPPED
    SYNC_LVL=CONFIRM
    ATTACH_TIMEOUT=5;
```

```
lu62_listen_t listen_req;
lu62_register_tp_t register_req;

,” Register with LU for all TPs “TR10?” ,“
register_req.port_id = open_req.port_id;
strncpy(register_req.tp_name, “TR10?”, LU62_TP_NAME_LEN);
if (lu62_register_tp(&register_req) ,“ LU62_ERROR) {
    printf(“lu62_register_tp error, %0x%x\n”, lu62_errno);
    exit(1);

}
,” Listen and dispatch incoming conversations ,“
listen_req.port_id = open_req.port_id;
while (lu62_listen(&accept_req) != LU62_ERROR) {
    if (strcmp(listen_req.tp_name, “TR100”) ,“ 0) {
        dispatch_tr100(listen_req.own_tp_instance);
    }
    else if (strcmp(listen_req.tp_name, “TR101”) ,“ 0) {
        dispatch_tr101(listen_req.own_tp_instance);
    }
    else if (strcmp(listen_req.tp_name, “TR102”) ,“ 0) {
        dispatch_tr102(listen_req.own_tp_instance);

    else {
        printf(“attach arrived for unknown TP”);
        ,“ let the server time it out ,“
    }
}

printf(“lu62_listen error, %0x%x\n”, lu62_errno);
```

5.6 Handling Multiple Concurrent Conversations

A SunLink P2P LU6.2 9.1 program can participate in multiple concurrent conversations. To do this effectively, the program cannot afford to block while it waits for an operation on one conversation to complete—it must be set up to operate in non-blocking mode. In non-blocking mode, verbs that require the API to issue a request to the SunLink SNA PU2.1 9.1 server return to your program with a `return_code` of `LU62_OPERATION_INCOMPLETE`. You cannot issue any further verbs on that conversation until the operation is completed, that is, the API has received an eventual response from the server. You may, however, issue verbs on other conversations. The `lu62_wait_server` call is issued when you need to synchronize completion of the operation.

Each conversation may be independently set into `PM_BLOCKING` or `PM_NON_BLOCKING` modes. The initial mode is established by `lu62_(mc_)allocate` and `lu62_accept` verbs. The `lu62_set_processing_mode` verb can be issued subsequently to change the mode.

Code Example 5-11 accepts incoming conversations and processes them concurrently. Note that the processing mode is set by `lu62_accept`.

Code Example 5-11

```
lu62_accept_t accept_req;
lu62_register_tp_t register_req;
bit32 conv_id, new_conv_id;

,” Register with LU as TP FRED ,“
register_req.port_id = open_req.port_id;
strncpy(register_req.tp_name, “FRED”, LU62_TP_NAME_LEN);
if (lu62_register_tp(&register_req) ,“ LU62_ERROR) {
    printf(“lu62_register_tp error, %0x%x\n”, lu62_errno);
    exit(1);

,” Listen for incoming conversation ,“
accept_req.port_id = open_req.port_id;
accept_req.processing_mode = PM_NON_BLOCKING;
if (lu62_accept(&accept_req) ,“ LU62_ERROR)
    printf("lu62_accept error, 0x%x, lu62_errno);
    exit(1);

}
/*
 * accept_req.return_code must be LU62_OPERATION_INCOMPLETE
 * and the conv_id for pending conversation is assigned.
 */
new_conv_id = accept_req.conv_id;

while (1) {
    /* wait indefinitely for operation to complete */
    if (lu62_wait_server(NULL, &conv_id) == LU62_ERROR) {
        printf("lu62_wait_server error, %0x%x\n", lu62_errno);
        exit(1);

    }
    /* received new conversation ? */
    if (conv_id == new_conv_id) {
        initiate_transaction(conv_id);

        /* and listen for the next one */
        if (lu62_accept(&accept_req) == LU62_ERROR) {
            printf("lu62_accept error, %0x%x\n", lu62_errno);
            exit(1);
        }
```

Code Example 5-11 (Continued)

```
lu62_accept_t accept_req;
        new_conv_id = accept_req.conv_id;
 }
    else{
        continue_transaction(conv_id);
    }
}
```

5.7 Mapped Conversations

Use mapped conversations whenever possible since they are easier to program. When a conversation is mapped, your program sends and receives one record at a time, where each record contains data only. Sending and receiving data, therefore, is relatively straightforward. In contrast, basic conversations require you to format your data into logical records, as described in the rest of this section.

Mapped conversations are so called because data mapping is one of the possible options in the [TPRM]. Data mapping allows communicating TPs to work with data formats that they understand. Each LU converts (or maps) received data into the local format using mapping routines and are identified by a *map_name*, specified by the local TP. The use of *map_names*, however, is not currently supported by SunLink P2P LU6.2 9.1.

5.7.1 Sending Data Records

Mapped conversation programs send data records to the remote TP using `lu62_mc_send_data`. The LU6.2 copies the data record into a Request Unit (RU) for transmission. The RU is not transmitted until either:

- The RU is completely full.
- The sending program issues an `lu62_mc_flush` verb.
- The sending program issues a verb that requires a response (`lu62_mc_confirm`, `lu62_mc_prep_to_receive`, `lu62_mc_receive_and_wait`, or `lu62_mc_deallocate`).

In theCode Example 5-12, the `lu62_mc_flush` verb is used to force a record to be sent to the remote TP.

Code Example 5-12

```
lu62_send_data_t send_data_req;
lu62_flush_t flush_req;
char data_rec[MAX_RECORD_LENGTH];
int len;

data_rec is set with data record to send, len is record length

send_data.conv_id = conv_id;
send_data_req.data = data_rec;
send_data_req.length = len;
if (lu62_mc_send_data(&send_data_req) == LU62_ERROR) {
    printf("lu62_mc_send_data error, %0x%x\n", lu62_errno);
    exit(1);

}
flush.conv_id = conv_id;
if (lu62_mc_flush(&flush_req) == LU62_ERROR) {
    printf("lu62_mc_flush error, %0x%x\n", lu62_errno);
    exit(1);
}
```

5.7.2 Receiving Data

On the receiving side, your program issues the `lu62_mc_receive_and_wait` verb or the `lu62_mc_receive_immediate` verb to receive information on the mapped conversation. This information may be data or status information. In this case the conversation `sync_level` characteristic is `SYNC_LEVEL_NONE`, so no confirmation request can be received.

Code Example 5-13 continues to illustrate the remote TP that is receiving one record at a time using `lu62_mc_receive_and_wait.` This verb will wait indefinitely until the local LU receives information for the program.

Code Example 5-13

```
lu62_receive_t receive_req;
bit8 buf[MAX_RECORD_LENGTH];
int receive_state = TRUE;

bzero(&receive_req, sizeof(lu62_receive_t));
receive_req.conv_id = conv_id;
receive_req.data = buf;
while (receive_state) {
    receive_req.length = MAX_RECORD_LENGTH;
    if (lu62_mc_receive_and_wait(&receive_req) == LU62_ERROR) {
        printf("lu62_mc_receive_and_wait error, %0x%x\n",
lu62_errno);
        exit(1);
    }
    switch (receive_req.what_received) {
    case WR_DATA_COMPLETE:
        process_record(buf, len);
        break;
    case WR_DATA_INCOMPLETE:
        /* issue another receive to determine why */
        break;
    case WR_SEND:
        receive_state = FALSE;
        break;
    }
}
```

5.8 Basic Conversations

In contrast to mapped conversations, basic conversations use a simple logical record format for transferring data. Logical records contain a 2- byte “LL” prefix that contains the length of the data record (including the LL prefix). Figure 5-1 depicts the logical record format.

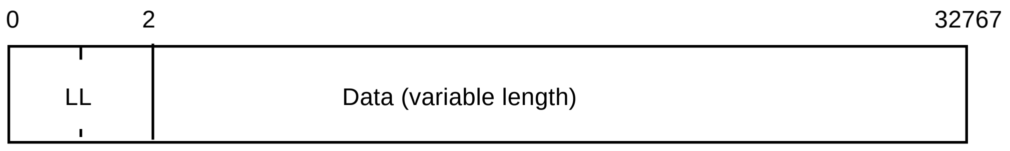


Figure 5-1 Logical Record Format

5.8.1 *Sending Logical Records*

Basic conversation programs send buffers to the LU6.2 using the `lu62_send_data` verb. Send buffers are not necessarily split into whole logical records; the sending program determines it. In Code Example 5-14, however, data is sent one logical record at a time, and confirmation requested for each record is sent. Note that the conversation sync_level must be `SYNC_LEVEL_CONFIRM`.

Code Example 5-14

```
lu62_confirm_t confirm_req;
lu62_send_data_t send_data_req;
struct {
    bit16 ll_hdr;
    data[MAX_DATA_LEN];
}
ll_rec;
int len;

build data into ll_rec.data maintaining len variable

/* Send Data */
len += 2; /* include ll_hdr */
ll_rec.ll_hdr = (bit16)len;
send_data_req.conv_id = conv_id;
send_data_req.data = &ll_rec;
send_data.length = len;
if (lu62_send_data(&send_data_req) == LU62_ERROR) {
    printf("lu62_send_data error, %0x%x\n", lu62_errno);
    exit(1);
}
confirm_req.conv_id = conv_id;
if (lu62_confirm(&confirm_req) == LU62_ERROR) {
    printf("lu62_confirm error, %0x%x\n", lu62_errno);
    exit(1);
}
```

5.8.2 Receiving Data

On the receiving side, the program issues the `lu62_receive_and_wait` verb or the `lu62_receive_immediate` verb to receive information on the basic conversation. This information may be data or status information. You may choose to receive one record at a time (fill type = `FILL_LL`), or you may choose to receive complete buffers (fill type = `FILL_BUFFER`). In the latter case, your program is responsible for extracting logical records from the buffer.

The example below illustrates the remote TP that receives one record at a time, and responds to confirmation requests with `lu62_confirmed.`

Code Example 5-15

```
lu62_receive_t receive_req;
lu62_confirmed_t confirmed_req;
bit8 buf[MAX_RECORD_LENGTH];
int receive_state = TRUE;
int rc = LU62_OK;

confirmed_req.conv_id = conv_id;
receive_req.conv_id = conv_id;
receive_req.data = buf;
receive_req.length = MAX_RECORD_LENGTH;
receive_req.fill = FILL_LL;
while ((rc == LU62_OK) && receive_state) {
    rc = lu62_receive_and_wait(&receive_req);
    if (rc == LU62_ERROR) {
        printf("lu62_receive_and_wait error, %0x%x\n",
lu62_errno);
        break;
    }
    switch (receive_req.what_received) {
    case WR_DATA_COMPLETE:
        rc = process_record(buf, len);
        break;
    case WR_LL_TRUNCATED:
    case WR_DATA_INCOMPLETE:
        /* issue another receive to determine why */
        break;
    case WR_SEND:
        receive_state = FALSE;
        break;
    case WR_CONFIRM:
        rc = lu62_confirmed(&confirmed_req);
        break;
    case WR_CONFIRM_SEND:
    case CM_CONFIRM_DEALLOCATE:
        receive_state = FALSE;
        rc = lu62_confirmed(&confirmed_req);
        break;
    }
}
```

5.9 Select Calls to Multiplex LU6.2 Events with Events from Other Devices

User programs are frequently required to handle multiple, inter-mixed events, for example, terminal input and LU6.2 events. The examples below illustrate the Unix `select` call to multiplex between standard input and LU6.2 connections.

5.9.1 Multiple Non-Blocking Conversations

In the Code Example 5-16, two conversations are multiplexed over a single LU6.2 connection.

Code Example 5-16

```
int tin;
fd_set readfds;

/* standard input file descriptor */
tin = fileno(stdin);

lu62_open(&open_req);

/* Issue Allocate requests in LU62_NON_PROCESSING_MODE */
allocate_req1.processing_mode = LU62_NON_BLOCKING;
lu62_allocate(&allocate_req1);
allocate_req2.processing_mode = LU62_NON_BLOCKING;
lu62_allocate(&allocate_req2);

/* Main Select Loop */
while (1) {
    /* set up fds for select call */
    FD_ZERO(&readfds);
    lu62_get_readfds(&readfds);         /* LU6.2 connections*/
    FD_SET(tin, &readfds);              /* standard input   */

    /* wait for read event to be posted */
   if ((n = select(FD_SETSIZE, &readfds, NULL, NULL, NULL)) < 0) {
        /* select call can be interrupted - ignore interrupts */
        if (errno != EINTR) {
            perror("select");
            exit(1);
```

Code Example 5-16 (Continued)

```
        }
    }
        if (FD_ISSET(tin, &read_fds) {
            read_and_process_terminal_input();
        }

        if (FD_ISSET(open_req.port_desc, &read_fds) {
            rc = lu62_wait_server(NULL, &conv_id);
            if (rc == LU62_ERROR)
              printf("lu62_wait_server error, %0x%x\n", lu62_errno);
                exit(1);
            }
            continue_conversation(conv_id);
        }
    }
```

When an LU6.2 connection event occurs, it signals the arrival of a verb response on one of the two conversations (or some other socket read event, such as a broken pipe). `lu62_wait_server` is called to receive off the socket and set-up any verb return values for the caller. The returned `conv_id` indicates the conversation on which the operation is completed.

5.9.2 Mixing Non-Blocking and Blocking Conversations

If you are using the `select` call to receive notification of LU6.2 events, special consideration is required if you want to mix non-blocking and blocking conversations on the same LU6.2 connection.

The calling sequence in Code Example 5-17 is used to illustrate the potential problem. In this example, a non-blocking `lu62_receive_and_wait` is issued followed by a blocking `lu62_send_data.` A `select` call is issued after the `lu62_send_data` completes to wait for the LU6.2 read event, corresponding to the arrival of the `lu62_receive_and_wait` response. When this event occurs, `lu62_send_data` is issued to receive the response.

Code Example 5-17

```
,” issue non-blocking receive on conversation 1 ,"
lu62_receive_and_wait(&receive_req);

/* issue blocking send on conversation 2 */
lu62_send_data(&send_req);

lu62_get_readfds(&readfds);             /* LU6.2 connections*/

/* wait for read event to be posted */
n = select(FD_SETSIZE, &readfds, NULL, NULL, NULL);
if (FD_ISSET(open_req.port_desc, &read_fds) {
    rc = lu62_wait_server(NULL, &conv_id);
}
```

If both pieces of conversations are multiplexed over the same LU6.2 connection while the LU6.2 API waits for the `lu62_send_data` response, the `lu62_receive_and_wait` verb response may arrive. In this case the API processes the `lu62_receive_and_wait` response and enqueues it so it is available when the program next issues `lu62_wait_server`. Then the `lu62_send_data` response is received and the `lu62_send_data` verb returns. When the `select` call is made, no read event is pending since the `lu62_receive_and_wait` response has already been received, and `lu62_wait_server` is not called.

If the conversations are multiplexed over different LU6.2 connections, the `lu62_receive_and_wait` response remains in the pipe and its presence is signaled by a `select` read event. The simplest solution, therefore, is *not* to mix non-blocking and blocking conversations on the same LU6.2 connection. Open a separate LU6.2 connection for each mode.

5.9.3 Polling for a Verb Response

Another approach to `select` handling is to poll the LU6.2 connections for pending verb responses after handling other read events. See Code Example 5-18 for details.

Code Example 5-18

```
,” return immediately from lu62_wait_server calls ,"
struct timeval timeout = , 0

while (1) {
    /* set up fds for select call */
    lu62_get_readfds(&readfds);         /* LU6.2 connections*/

    FD_SET(tin, &readfds);              /* standard input   */

    /* wait for read event to be posted */
   if ((n = select(FD_SETSIZE, &readfds, NULL, NULL, NULL)) < 0) {
        /* select call can be interrupted - ignore interrupts */
        if (errno != EINTR) {
            perror("select");
            exit(1);
        }
    }
    /* handle non-LU6.2 read events */
    if (FD_ISSET(tin, &read_fds) {
        read_and_process_terminal_input();
        /* verb may be issued */
    }

    /* poll LU6.2 connections */
    rc = lu62_wait_server(&timeout, &conv_id);
    if (rc == LU62_ERROR) {
        switch (lu62_errno) {
        case LU62_NO_VERB_IN_PROGRESS:
            break;
        case LU62_WAIT_TIMEOUT)
            /* operation remains incomplete */
            break;
        default:
            printf("lu62_wait_server error, %0x%x, lu62_errno);
            exit(1);
        }
    }
    else {
        continue_conversation(conv_id);
    }
}
```

5.10 Control Operator Programming

Control operator programs open connections to an LU in the SunLink SNA PU2.1 9.1 server using the `lu62_open` verb, much like transaction programs. The only difference is that the open request must specify a `tp_name`. To perform control operator functions, programs require special privileges. These privileges are assigned in the configuration using the `TP PRIVILEGE` parameter. The `tp_name` in the open request is used to associate the connecting program with a configured TP and, therefore, determine its privileges.

Once an LU connection is achieved, control operator verbs are issued using the `port_id` of the open connection. In the Code Example 5-19, a control operator connection is established to issue an `lu62_activate_session` verb.

Code Example 5-19

```
lu62_open_req_t open_req;
lu62_activate_session_t act_sess_req;
lu62_close_req_t close_req;

char *copr = "COPR"; /* configured Control Operator */

/* build open request */
bzero(&open_req, sizeof(struct lu62_open_req));
strncpy(open_req.host, LU62_SERVER, MAXHOSTNAMELEN);
strncpy(open_req.lu_name, LOCAL_LU, LU62_LU_NAME_LEN);
strncpy(open_req.tp_name, copr, LU62_TP_NAME_LEN);

if (lu62_open(&open_req) == LU62_ERROR) {
    printf("lu62_open error, 0x%x\n", lu62_errno);
    exit(1);
}

/*
 * activate session
 * - assume partner_lu and mode name are established
 */
act_sess_req.port_id = open_req.port_id;
strncpy(act_sess.lu_name, partner_lu, LU62_LU_NAME_LEN);
strncpy(act_sess.mode_name, mode_name, LU62_MODE_NAME_LEN);
if (lu62_activate_session(&act_sess_req) == LU62_ERROR) {
    printf("lu62_activate_session error, 0x%x\n", lu62_errno);
    exit(1);
}

close_req.port_id = port_id;
if (lu62_close(&close_req) == LU62_ERROR) {
    printf("lu62_close error, 0x%x\n", lu62_errno);
    exit(1);
}
```

5.11 Using the Select Call to Receive CNOS Notifications

Normally, all messages sent by the SunLink SNA PU2.1 9.1 server to a client program are done in response to a prior request. CNOS notifications are an exception. CNOS notifications are sent to requesting control operator programs whenever CNOS parameters are updated by the LU. These notifications are unsolicited and can arrive at any time.

To receive CNOS notifications, a special connection is made to the LU using the `lu62_request_notification` verb. This verb opens a separate socket connection to the SunLink SNA PU2.1 9.1 server. Your program then issues `lu62_receive_notification` to read a notification from the socket. `lu62_receive_notification` will, however, block until a notification is available to read. To prevent blocking, your program can poll the connection for pending notifications, using the `lu62_poll_notification` verb, or it can use the Unix `select` call to receive CNOS notifications asynchronously.

In Code Example 5-20, the `select` call is used to multiplex program control between COPR verb returns, CNOS events, and terminal input.

Code Example 5-20 (1 of 3)

```
lu62_open_req_t open_req;
lu62_request_notification_t req_notify;
lu62_notification_header_t notify_hdr;
lu62_cnos_notification_t cnos_notification;
int cnos_notifications_requested = TRUE;
int tin;
fd_set readfds;
char *copr = "COPR"; /* configured Control Operator */

/* standard input file descriptor */
tin = fileno(stdin);

/* connect to local_lu to issue COPR verbs */
bzero(&open_req, sizeof(lu62_open_req_t));
strncpy(open_req.host, LU62_SERVER, MAXHOSTNAMELEN);
strncpy(open_req.lu_name, LOCAL_LU, LU62_LU_NAME_LEN);
strncpy(open_req.tp_name, copr, LU62_TP_NAME_LEN);
if (lu62_open(&open_req) == LU62_ERROR) {
    printf("lu62_open error, 0x%x\n", lu62_errno);
    exit(1);
```

Code Example 5-20 (2 of 3)

```
}

/* request CNOS notifications from local_lu */
bzero(&req_notify, sizeof(lu62_request_notification_t));
strncpy(req_notify.lu_name, LOCAL_LU, LU62_LU_NAME_LEN);
rc = lu62_request_notification(&req_notify);
if (rc == LU62_ERROR) {
    printf("lu62_request_notification error = 0x%x\n",
           req_notify.return_code);
    exit(1);
}

/* initialize for CNOS notification processing */
notify_hdr.port_id  = req_notify.port_id;

while (rc != LU62_ERROR) {
    /* set up fds for select call */
    FD_ZERO(&readfds);
    lu62_get_readfds(&readfds);         /* LU62 channels    */
    if (cnos_notifications_requested)
        FD_SET(req_notify.port_desc, &readfds);
    FD_SET(tin, &readfds);              /* standard input   */

    /* wait for read event to be posted */
   if ((n = select(FD_SETSIZE, &readfds, NULL, NULL, NULL)) < 0) {
        /* select call can be interrupted - ignore interrupts */
        if (errno != EINTR) {
            perror("select");
            exit(1);
        }
    }

    /* dispatch read events */
    if (FD_ISSET(req_notify.port_desc, &readfds) {
        rc = lu62_receive_notification(&notify_hdr,
&cnos_notification);
        if (rc == LU62_OK) {
            switch (notify_hdr.op_code) {
            case LU62_REQUEST_NOTIFICATION_REPLY:
                break;
            case LU62_CNOS_NOTIFICATION:
                process_cnos_notification(&cnos_notification);
                break;
```

Code Example 5-20 (3 of 3)

```
             case LU62_STOP_NOTIFICATION_REPLY:
                 cnos_notifications_requested = FALSE;
                 break;
             }
     }

     else if (FD_ISSET(tin, &readfds) {
         read_and_process_terminal_input();
         /* copr verb may be issued */
     }
     else {
         /* receive response to COPR verb */
         rc = lu62_wait_server(NULL, &id);
         if (rc == LU62_OK) {
             process_copr_return(id);
     }
  }
```

man Page Conventions 6

The remaining chapters of this manual contains man pages for the SunLink P2P LU6.2 9.1 verbs. The function of each verb is described, and the following information is provided, if it applies:

Synopsis

ANSI C language function prototypes are used to show the name of the API call and parameters. For example:

```
int lu62_flush (lu62_flush_t *rqp);
```

where `lu62_flush` is the name of the call, which takes a request structure of type lu62_flush_t.

- Request Structure—Describes the request data structure, see Data Types and Request Structure, below.
- State Changes—Specifies the changes in the conversation state that can result from this call.

Usage Notes

Provides any additional information that applies to the call.

See Also

References material in this manual that is related to the call.

6.1 Data Types

Many of the parameters and data structure members are defined as `bit8`, `bit16`, or `bit32`. These data types are used instead of the standard integer types, when it is necessary to ensure that a parameter type is independent of the actual integer size of the particular machine. Table 6-1 defines these three data types.

Table 6-1 Data Types

Data Type	**Definition**
bit8	An unsigned character in the range of 0x00 to 0xff
bit16	An unsigned integer in the range of 0x0000 to 0xffff
bit32	An unsigned integer in the range of 0x00000000 to 0xffffffff

6.2 Request Structures

Each member (referred to in the following documentation as both parameter and field) of the request data structure is described as being supplied, returned, supplied / returned, or ignored.

- Supplied parameters are set by the user program.
- Returned parameters are set by the successful operation of the verb.
- Supplied / returned parameters are set by the user program when the verb is issued, but their values may change after the successful operation of the verb.
- Ignored parameters are not used or set by the verb.

Initialization of every member of the structure is handled by the application. You should first initialize the request structures to zero and then set them with the required parameter values. This ensures that pointer values are initialized to NULL and enumerated type values are defaulted. Enumerated types are defined so that the default value, which is underlined, is zero.

Supplied parameters are further specified as required, conditional, or optional:

- Required parameters must be set by the application program.
- Conditional parameters may have a required value, depending on the setting of another parameter.
- Optional parameters need not be set.

Connection Verbs 7

Connection verbs are used to establish and maintain connection to one or more SunLink SNA PU2.1 9.1 servers (Table 7-1). These verbs are all SunLink P2P 9.1 extensions to the LU6.2 verb set defined in the *IBM SNA Transaction Programmer's Reference Manual*.

Table 7-1 SunLink LU6.2 Connection Verbs

Verb	Function
`*lu62_open`	Opens a connection with the SunLink PU2.1 SNA server
`*lu62_close`	Closes a connection with the SunLink PU2.1 SNA server
`*lu62_set_processing_mode`	Sets to BLOCKING or NON_BLOCKING mode
`*lu62_wait_server`	Waits for a response from the server
`*lu62_get_readfds`	Returns the select fdsets for LU connections

The use of the connection verbs is summarized below, followed by detailed man pages.

7.1 Program Connections to the SunLink PU2.1 SNA Server

`lu62_open` must be issued before communication with the SunLink SNA PU2.1 9.1 server can take place. This verb establishes a socket connection between your program and a specific LU supported by the SunLink SNA PU2.1 9.1 server. Your program can connect to many LUs in the SunLink SNA PU2.1 9.1 server. It can also connect to multiple SunLink SNA PU2.1 9.1 servers. When your program has completed communication, it should disconnect from the server using `lu62_close`.

7.2 Multiplexed Communication Channels

Socket connections to the server are opened using `lu62_open`. When the connection is first opened, a control channel is set up. All LU-level verbs (`lu62_close`, control operator verbs, etc.) are transported on this control channel. Multiple conversations may be multiplexed over each connection. Each time a conversation is allocated or accepted, a new channel is established.

7.3 Processing Mode

Each communication channel may be independently set into `PM_BLOCKING` or `PM_NON_BLOCKING` modes. The initial mode for the control channel is established by `lu62_open`; the initial mode for conversation channels is established by `lu62_allocate` and `lu62_accept` verbs. The `lu62_set_processing_mode` verb is used to change the mode of the LU control channel or any conversation channel.

In `PM_BLOCKING` mode, all verbs wait for their response from the SunLink SNA PU2.1 9.1 server before returning to the caller. In `PM_NON_BLOCKING` mode, the verb returns to the caller once the request has been sent to the server.

The caller must issue an `lu62_wait_server` verb when it is ready to receive the response(s). This verb returns with the id of the first channel to receive a response to its outstanding verb. An `lu62_wait_server` call must eventually be made for all channels with verb responses outstanding.

7.4 User Select Control

You may also want to use the Unix `select` call to multiplex program control between SunLink P2P LU6.2 9.1 connections, and other open devices such as TTY. To do this, you must have access to the SunLink P2P LU6.2 9.1 socket file descriptors to construct the `select readfds`. Socket file descriptors are returned by `lu62_open`. In addition, `lu62_get_readfds` returns `select readfds` for all sockets that carry channels with verb responses outstanding. Add the file descriptors for other devices to the `readfds`, using `FD_SET` and issuing `select`. If `select` indicates a pending read on a SunLink P2P LU6.2 9.1 socket, call `lu62_wait_server` to perform the read.

7.5 *lu62_close

A program closes a connection to the SunLink SNA PU2.1 9.1 server using the `lu62_close` call. Before making this call, a program should ensure that all conversations are deallocated. The SunLink SNA PU2.1 9.1 server will deallocate (type = ABEND_SVC) any conversations that are active when `lu62_close` is issued.

Synopsis

```
int lu62_close(lu62_close_req_t *rqp);
```

Request Structure

```
typedef struct {
   bit32            port_id;                           /* s */
   bit32            return_code;                       /* r */
}lu62_close_req_t;
```

The `lu62_close_req_t` request structure members are:

`port_id`

(supplied); specifies the `port_id` of the LU connection to be closed. The *port_id* is returned by `lu62_open`.

`return_code`

(returned); specifies the result of verb execution to be one of:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  - `LU62_PORT_ID_UNKNOWN`
- `LU62_PROGRAM_STATE_CHECK`
  - `LU62_VERB_IN_PROGRESS`

State Changes

When `return_code` is `LU62_OK`, the LU connection is closed. All active conversations on the LU are terminated.

Usage Notes

When multiple connections to SunLink SNA PU2.1 9.1 server(s) are maintained, the program should retain the `port_ids` returned by `lu62_open` calls, so that it can use `lu62_close` to take down the connections gracefully.

See Also

`lu62_open`—describes how to handle connections to the SunLink SNA PU2.1 9.1 server.

7.6 *`lu62_get_readfds`

`lu62_get_readfds` returns `select` readfds for sockets with outstanding verb responses. Users who want to use `select` to multiplex SunLink LU6.2 socket events with events from other devices can use this verb to include the active SunLink LU6.2 sockets (identified by their port descriptors) in the `select` readfds. You should then add your own file descriptors to the readfds using FD_SET and issue select. If select indicates a pending read on a SunLink LU6.2 socket, issue lu62_wait_server to read the response.

Synopsis

```
#include <sys/types.h>
int lu62_get_readfds(fd_set *readfds_p);
```

Parameters

readfds_p

(returned). Specifies the address of an fd_set structure to receive the socket fds for all channels with outstanding verbs.

Return Value

The verb always returns LU62_OK.

See Also

lu62_wait_server.

7.7 *lu62_open

A program opens a connection to the SunLink SNA PU2.1 9.1 server using the lu62_open verb. A connection associates your program with a specific LU defined to a SunLink SNA PU2.1 9.1 server. Your program can connect to many LUs in the SunLink SNA PU2.1 9.1 server. It can also connect to multiple SunLink SNA PU2.1 9.1 servers.

Each connection is identified by a port_id. All subsequent non-conversation verbs issued to the selected LU designate this port_id.

Synopsis

```
int lu62_open(lu62_open_req_t *rqp);
```

Request Structure

```
typedef struct {
    char           host[MAXHOSTNAMELEN+1];          /* so */
    char           lu_name[LU62_LU_NAME_LEN+1];     /* s */
    char           tp_name[LU62_TP_NAME_LEN+1];     /* so */
    lu62_processing_mode_e processing_mode;         /* s */
    bit32          return_code;                     /* r */
    bit32          port_id;                         /* r */
    int            port_desc;                       /* r */
} lu62_open_req_t;
```

The `lu62_open_req_t` request structure members are:

host

(supplied/optional). Specifies the TCP/IP hostname of the SunLink SNA PU2.1 9.1 server host. Hostnames are configured in the Unix network configuration file, `/etc/hosts`, or are maintained by NIS. *server* is supplied as an ASCII (null-terminated) string. If *host* is not supplied, localhost is assumed.

`lu_name`

(supplied/optional). Specifies the name of the logical unit in the SunLink SNA PU2.1 9.1 server with which this connection is to be associated. This name corresponds to the `LU_NAME` parameter of the LU definition. `lu_name` is supplied as an ASCII (null-terminated) string. If not specified, it will get assigned by the Gateway. It can be examined using `get_tp_properties`.

tp_name

(supplied/optional). Specifies the name of the local transaction program. A tp_name is required if the program requires any special privileges, such as the ability to issue control operator verbs. Such privileges are associated with local TPs via the configuration file. See the TP directive in the configuration. tp_name is supplied as an ASCII (null-terminated) string.

processing_mode

(supplied). Specifies the initial processing mode of the LU connection as either:

- PM_BLOCKING
- PM_NON_BLOCKING

If processing_mode is set to PM_BLOCKING, lu62_open does not return until open processing is successfully completed (*return_code* = LU62_OK) or fails. If processing_mode is set to PM_NON_BLOCKING and initial parameter checks pass, return_code is set to LU62_OPERATION_INCOMPLETE and the lu62_wait_server must be issued to receive the eventual return.

The specified processing_mode remains in effect for the LU control channel until lu62_set_processing_mode is issued or the connection is closed. Note, the processing mode of conversations allocated or accepted over the LU connection is specified separately on the respective lu62_allocate and lu62_accept verbs.

return_code

(returned). Specifies the result of call execution. The return_code variable may have one of the following values:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - LU62_HOST_UNKNOWN
  - LU62_SERVER_UNKNOWN

- `LU62_LU_NAME_REQD`
- `LU62_BAD_LU_NAME`
- `LU62_BAD_TP_NAME`
- `LU62_BAD_PROCESSING_MODE`

`port_id`

(returned). Specifies the port identifier assigned to the connection. All subsequent non-conversation verbs issued to the selected LU designate this `port_id`.

`port_desc`

(returned). Specifies the file descriptor associated with the socket connection. The `port_desc` is available for users who want to perform their own select processing.

State Changes

Not applicable.

Usage Notes

1. `lu62_open` must be issued before any communication with the SunLink SNA PU2.1 9.1 server can take place. When the connection is first opened, a control channel is set up. All LU-level verbs (`lu62_close`, control operator verbs, etc.) are transported on this control channel. Multiple conversations may be multiplexed over each connection. Each time a conversation is allocated or accepted, a new conversation channel is established.

2. You may connect to LUs supported by multiple SunLink SNA PU2.1 9.1 servers. For each LU that you require access to, a separate connection is required. Multiple conversations may be multiplexed over each connection.

See Also

Handling Connections to the SunLink SNA PU2.1 9.1 server.

7.8 *lu62_set_processing_mode

lu62_set_processing_mode is used to set the processing mode of a communications channel to blocking or non-blocking.

A communications channel is multiplexed over the socket connection between the client program and the SunLink SNA PU2.1 9.1 server. The channel is either the control channel, which carries LU-level verbs, or a conversation channel, which carries conversation verbs.

The processing mode affects any verb that requires an interaction with the SunLink SNA PU2.1 9.1 server. If the processing mode is set to PM_BLOCKING, such verbs do not return until the server responds. If the processing mode is set to PM_NON_BLOCKING and initial parameter checks pass, the verb's return_code is set to LU62_OPERATION_INCOMPLETE and lu62_wait_server must be issued to receive the eventual return.

The processing mode cannot be changed when a verb response is outstanding.

Synopsis

```
int lu62_set_processing_mode(bit32 id, lu62_processing_mode_e
pmode);
```

Parameters

id

(supplied). Specifies the channel identifier. For the control channel, this is the *port_id* returned by lu62_open. For a conversation channel, this is the *conv_id* returned by lu62_allocate or lu62_accept.

pmode

(supplied). Specifies the required processing mode as either:

- PM_BLOCKING
- PM_NON_BLOCKING

RETURN VALUE. The verb returns an integer error code:

- LU62_OK, indicates that the processing mode was successfully updated.

- LU62_ERROR, indicates that the verb failed. *lu62_errno* is set to indicate the reason for failure:
  - LU62_CONV_ID_UNKNOWN
  - LU62_VERB_IN_PROGRESS

Usage Notes

See Also

lu62_open, lu62_accept, lu62_allocate, lu62_mc_allocate, lu62_wait_server and Handling Multiple Concurrent Conversations.

7.9 *lu62_wait_server

lu62_wait_server is used to wait for and receive a response to an outstanding verb, that is, a verb that was issued on a PM_NON_BLOCKING channel and that returned LU62_OPERATION_INCOMPLETE.

Synopsis

```
#include <sys/time.h>
int lu62_wait_server(struct timeval *timeout, bit32 *id_p);
```

Parameters

timeout

(supplied). Specifies the time to wait for a response. If NULL, lu62_wait_server will wait indefinitely. If a pointer to a zeroed timeval structure, lu62_wait_server polls all channels with an outstanding response to see if a response has been received, but does not wait for one.

`id_p`

(returned). Specifies the location to receive the identifier of the channel on which a response has been received. This may be the `port_id` of an LU control channel or the `conv_id` of a conversation channel.

RETURN VALUE. The verb returns an integer error code as described below.

- `LU62_OK`, indicates that a response was successfully received.
- `LU62_ERROR`, indicates that the verb has failed. *lu62_errno* is set to indicate the reason for failure:
  - `LU62_NO_VERB_IN_PROGRESS`
  - `LU62_WAIT_TIMEOUT`.

Usage Notes

When multiple connections to SunLink PU2.1 SNA server(s) are maintained, the program should retain the port_ids returned by `lu62_open` call so that it can use `lu62_close` to take down the connections gracefully.

See Also

Handling Multiple Concurrent Conversations.

Basic Conversation Verbs 8

This chapter describes the basic conversation verbs. Table 8-1 lists the SunLink P2P LU6.2 9.1 basic conversation verbs. Detailed man pages follow.

Table 8-1 SunLU6.2 Basic Conversation Verbs

Verb	**Function**
`lu62_allocate`	Initiates a conversation with a remote TP
`lu62_confirm`	Issues a confirmation request to the remote TP
`lu62_confirmed`	Issues a confirmation response
`lu62_deallocate`	Terminates a conversation
`lu62_flush`	Forces transmission of data in the send buffer
`lu62_get_attributes`	Returns information about a conversation
`lu62_post_on_receipt`	Sets receive posting conditions for a conversation
`lu62_prep_to_receive`	Changes the conversation from send to receive state
`lu62_receive_and_wait`	Waits for information to arrive and then receives it
`lu62_receive_immediate`	Receives available information but does not wait
`lu62_request_to_send`	Requests the turn

Table 8-1 SunLU6.2 Basic Conversation Verbs

Verb	Function
lu62_send_data	Sends data on the conversation
lu62_send_error	Notifies the remote program of a detected error
lu62_test	Tests a conversation for posting

8.1 lu62_allocate

lu62_allocate is used to initiate a basic or mapped conversation with a remote (partner) transaction program. A session is assigned for the exclusive use of the local and remote programs for the duration of the conversation. A conversation id is assigned to the conversation. This conversation id is used to identify the conversation in all subsequent verb issuances.

Synopsis

```
int lu62_allocate(lu62_allocate_t *rqp);
```

Request Structure

The lu62_allocate_t request structure members are shown in Code Example 8-1.

Code Example 8-1

```
typedef struct {
    bit32             port_id;                          /* s */
    bit32             tp_id;                            /* s */
     char               unique_session_name              /* so */
                            [LU62_UNIQUE_SESSION_NAME_LEN+1];
    char              lu_name[LU62_LU_NAME_LEN+1];         /* s */
    char              mode_name[LU62_MODE_NAME_LEN+1];     /* s */
    char               remote_tp_name[LU62_TP_NAME_LEN+1];     /*
s */
    bit32             conv_grp_id;                      /* s */
```

Code Example 8-1 (Continued)

```
    lu62_processing_mode_e processing_mode;                /*
s */
    lu62_conv_type_e   type;                             /* s */
    lu62_flush_e      flush;                             /* s */
    lu62_return_control_e return_control;                /* s */
    lu62_sync_level_e   sync_level;                      /* s */
    lu62_pip_presence_e pip_presence;                    /* s */
    lu62_security_e    security;                         /* s */
    char           user_id[LU62_MAX_USER_ID_LEN+1];      /* s */
    char           passwd[LU62_MAX_PASSWD_LEN+1];        /* s */
    char           profile[LU62_MAX_PROFILE_LEN+1];      /* s */
    bit32            conv_id;                            /* r */
    int                luw_len;                        /* s/r */
    bit8               luw[LU62_MAX_LUW_LEN];          /* s/r */
    bit32            return_code;                        /* r */
} lu62_allocate_t;
```

8.1.1 `lu62_allocate_t` *Request Structure Members*

The following subsections describe the `lu62_allocate_t` request structure members:

`port_id`

(supplied) Specifies the `port_id` for the LU connection. The `port_id` is returned by `lu62_open`.

`tp_id`

(Supplied / optional.) Specifies the id of a registered TP for which a conversation has been accepted. This parameter is used when *security* = `SECURITY_SAME`, see below. The `tp_id` is returned by `lu62_accept`.

`unique_session_name`

Used to specify the actual node used from the configuration instead of `lu_name` and `node`. To use unique session names, the TP must not use an `lu_name` in the previous open.

lu_name

(supplied) Specifies the locally known name of the partner LU at which the remote transaction program, `remote_tp_name`, is located. `lu_name` is supplied as an ASCII (null-terminated) string. It corresponds to the `PTNR_LU_NAME` or `NO_LU_NAME` parameter in the configuration.

mode_name

(supplied) Specifies the name of the required mode. The conversation is allocated on a session of this mode. `mode_name` is supplied as an ASCII (null-terminated) string and is translated to EBCDIC by the SunLink SNA PU2.1 9.1 server. It corresponds to the `MODE NAME` parameter in the configuration.

remote_tp_name

(supplied) Specifies the name of the transaction program to which conversation attachment is required. `remote_tp_name` is supplied as an ASCII (null-terminated) string and is translated to `EBCDIC` by the SunLink SNA PU2.1 9.1 server.

conv_grp_id

(ignored) Reserved for future use.

processing_mode

(supplied) Specifies the initial processing mode of the conversation from one of the following:

- `PM_BLOCKING`
- `PM_NON_BLOCKING`

If `processing_mode` is set to `PM_BLOCKING`, `lu62_allocate` does not return until a conversation is successfully allocated (`return_code = LU62_OK`) or the verb fails. If `processing_mode` is set to `PM_NON_BLOCKING` and initial parameter checks pass, `return_code` is set to `LU62_OPERATION_INCOMPLETE` and `lu62_wait_server` must be issued to receive the eventual return.

The specified `processing_mode` remains in effect for the allocated conversation until `lu62_set_processing_mode` is issued or the conversation terminates.

`type`

(supplied) Specifies the conversation type from one of the following:

- `CONVERSATION_BASIC`
- `CONVERSATION_MAPPED`

`flush`

(supplied) Specifies whether the allocation request is sent to the remote LU as soon as a session is allocated for the conversation, or whether the allocation request is retained until another flush condition arises. `flush` is set to one of the following:

- `FLUSH_NO`
- `FLUSH_YES`

`return_control`

(supplied) Specifies when, in relation to the allocation of a session for the conversation, the local LU returns control to the program. `return_control` is set to one of the following:

- `RC_WHEN_SESSION_ALLOCATED`
- `RC_IMMEDIATE`

Note – `LU62_OPERATION_INCOMPLETE` will still be returned when the processing mode is non-blocking.

- `RC_WHEN_CONWINNER_ALLOCATED`

`sync_level`

(supplied) Specifies the synchronization level for the conversation from one of the following:

- `SYNC_LEVEL_NONE`
- `SYNC_LEVEL_CONFIRM`
- `SYNC_LEVEL_SYNCPT`

The selected `sync_level` must correspond to the `TP SYNC_LEVEL` configuration of the remote TP.

pip_presence

(supplied) Specifies whether the pip_presence field of the FMH-5 Attach request is to be set or not by one of the following parameters:

- PIP_NOT_PRESENT
- PIP_PRESENT

If PIP_PRESENT is set, the FMH-5 Attach request is built with the pip_presence indicator set. The caller is obliged to build the PIP variable (see Chapter 11 of the *IBM SNA Formats* manual and send it using the lu62_send_data verb.

security

(supplied) Specifies the access security information that the partner LU requires to verify the identity of the conversation initiator and validates access to the remote program and its resources. security may be one of the following:

- SECURITY_NONE
  Security access information is omitted.
- SECURITY_SAME
  Specifies to use known (and already verified) user_id (and profile) information. If user_id (and profile) are supplied, these values are used. Otherwise, if tp_id is supplied, information is taken from the first allocation request accepted for that TP, or if tp_id is not supplied, the Unix user_id is sent.
- SECURITY_PROGRAM
  Indicates that the required user_id, password (and profile) are included in this allocate request.

user_id

(Supplied/conditional.) If security is SECURITY_NONE or SECURITY_PROGRAM, a user_id is required. user_id is specified as an ASCII (null-terminated) string and is translated to EBCDIC by the SunLink SNA PU2.1 9.1 server.

passwd

(Supplied/conditional). If security is SECURITY_PROGRAM, passwd is required. passwd is specified as an ASCII (null-terminated) string and is translated to EBCDIC by the SunLink SNA PU2.1 9.1 server.

profile

(Supplied / conditional.) profile is specified as an ASCII (null-terminated) string and is translated to EBCDIC by the SunLink SNA PU2.1 9.1 server.

conv_id

(returned) Specifies the identifier of the allocated conversation. All subsequent verbs issued on this conversation require this identifier.

luw

(Supplied / returned.) This extension to [TPRM] is provided for SYNC_LEVEL_SYNCPT (see Appendix F). If supplied, luw contains the complete Logical Unit of work identifier to be set in the FMH-5 Attach (see Chapter 11 of the *IBM SNA Formats* manual). If luw is not supplied for SYNC_LEVEL_SYNCPT, the SunLink SNA PU2.1 9.1 server will generate the LUW on the caller’s behalf.

luw_len

(supplied / returned) Length of logical unit of work.

return_code

(returned) Specifies the result of verb execution. return_code values are affected by the value of the return_control parameter. The following return_codes can occur with all values of return_control:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - LU62_PORT_ID_UNKNOWN
  - LU62_BAD_LU_NAME
  - LU62_BAD_MODE_NAME
  - LU62_BAD_REMOTE_TP_NAME
  - LU62_BAD_PROCESSING_MODE
  - LU62_BAD_CONV_TYPE
  - LU62_BAD_FLUSH_TYPE
  - LU62_BAD_RETURN_CONTROL

- — LU62_BAD_SYNC_LEVEL
- — LU62_BAD_SECURITY
- — LU62_BAD_SECURITY_PROGRAM
- — LU62_BAD_USERID
- — LU62_BAD_PASSWD
- — LU62_BAD_PROFILE
- — LU62_LU_NAME_REQD
- — LU62_MODE_NAME_REQD
- — LU62_REMOTE_TP_NAME_REQD
- — LU62_UNKNOWN_TP

- LU62_PROGRAM_STATE_CHECK
  - — LU62_TP_NOT_STARTED
- LU62_ALLOCATION_ERROR
  - — LU62_SYNC_LEVEL_NOT_SUPPORTED_BY_LU

If return_control is set to RC_WHEN_SESSION_ALLOCATED or RC_WHEN_CONWINNER_ALLOCATED, the following additional return_codes are possible:

- LU62_PARAMETER_ERROR
  - — LU62_UNKNOWN_MODE
  - — LU62_UNKNOWN_PARTNER_LU
- LU62_ALLOCATION_ERROR
  - — LU62_ALLOCATION_FAILURE_NO_RETRY
  - — LU62_ALLOCATION_FAILURE_RETRY

If return_control is set to RC_IMMEDIATE, the following additional return_codes are possible:

- LU62_PARAMETER_ERROR
  - — LU62_UNKNOWN_MODE
  - — LU62_UNKNOWN_PARTNER_LU
- LU62_UNSUCCESSFUL (no session is available)

State Changes

When return_code is LU62_OK, the conversation enters Send state.

Usage Notes

- Session contention occurs when the two LUs both attempt to allocate a conversation on the session at the same time. Contention is resolved by making one LU the contention-winner, and the other the contention-loser. The contention-winner is guaranteed access to the session; the contention-loser must first ask permission of the contention-winner LU before it attempts to allocate a conversation on the session. See the MODE directive in the configuration for more information on session limits, conwinnners, and conlosers.
- An allocation error resulting from the local LU's failure to obtain a session for the conversation is reported on the `lu62_allocate` call. An allocation error resulting from the remote LU's rejection of the allocation request is reported on a subsequent conversation call.

See Also

Section 5.3, "Allocating Conversations," provides an example of how this verb is used.

8.2 lu62_confirm

`lu62_confirm` sends a request for confirmation to the remote transaction program and waits for a reply. In normal circumstances, the remote program issues an `lu62_confirmed` verb in response. The LU flushes the conversation's send buffer as a function of this verb.

Synopsis

```
int lu62_confirm(lu62_confirm_t *rqp);
```

Request Structure

```
typedef struct {
    bit32          conv_id;                           /* s */
    bit32          return_code;                       /* r */
    bit32          request_to_send_received;          /* r */
} lu62_confirm_t;
```

8.2.1 `u62_confirm_t` *Request Structure Members*

The following subsections describe the `lu62_confirm_t` request structure members:

`conv_id`

(supplied) Specifies the id of the conversation to use. `conv_id` is returned by `lu62_allocate or lu62_accept`.

`return_code`

(returned) Specifies the result of verb execution from one of the following:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  - `LU62_CONV_ID_UNKNOWN`
  - `LU62_BAD_SYNC_LEVEL`
- `LU62_PROGRAM_STATE_CHECK`
  - the conversation is not is Send state
  - `LU62_PIP_PENDING` - the conversation is in Send state following an `lu62_(mc_)_allocate` verb in which `pip_presence` was indicated. A basic send, `lu62_send_data`, is required to send the PIP Variable.
  - `LU62_VERB_IN_PROGRESS`
- `LU62_ALLOCATION_ERROR`
- `LU62_DEALLOCATE_ABEND_PROG`
- `LU62_DEALLOCATE_ABEND_TIMER`
- `LU62_DEALLOCATE_ABEND_SVC`
- `LU62_PROG_ERROR_PURGING`
  - `LU62_RESOURCE_FAILURE_NO_RETRY`
  - `LU62_RESOURCE_FAILURE_RETRY`
  - `LU62_SVC_ERROR_PURGING`

`request_to_send_received`

(returned) Indicates whether or not a request-to-send indication has been received from the remote program:

- TRUE, indicates that a request-to-send indication was received.

- FALSE, indicates that a request-to-send indication was not received.

State Changes

This verb can only be issued in Send state. No state change occurs.

Usage Notes

1. This verb is used to synchronize local and remote processing:
   - The initiating program may issue this verb immediately following `lu62_allocate` to ensure that the remote program is available and attached before sending any data.
   - The sending program may issue this verb as a request for acknowledgment of the data it sent to the remote program. The remote program issues `lu62_confirmed` to positively acknowledge receipt, or `lu62_send_error` to indicate that it encountered an error.

2. When `request_to_send_received` is TRUE, the remote program is requesting that the local program “give up the turn”, that is, that it enter Receive state, thereby placing the remote program in Send state. The local program enters Receive state by issuing `lu62_prep_to_receive` or `lu62_receive_and_wait`. The remote program issues `lu62_receive_immediate` or `lu62_receive_and_wait` to receive the resulting send indication (`what_received` = `WR_SEND`).

See Also

`lu62_confirmed`, `lu62_send_error`

Section 5.8, “Basic Conversations,” illustrates how programs can be synchronized using confirmation requests.

8.3 lu62_confirmed

`lu62_confirmed` sends a confirmation reply in response to a confirmation request from the remote transaction program. The local program issues this verb when it receives a confirmation request. (See the `what_received` parameter of the `lu62_receive_and_wait` and `lu62_receive_immediate` verbs). This verb can only be issued as a reply to a confirmation request; it cannot be issued at any other time.

Synopsis

```
int lu62_confirmed(lu62_confirmed_t *rqp);
```

Request Structure

```
typedef struct {
    bit32          conv_id;                          /* s */
    bit32          return_code;                      /* r */
} lu62_confirmed_t;
```

8.3.1 lu62_confirmed_t Request Structure Members

The following subsections describe the lu62_confirmed_t request structure members:

conv_id

(supplied) Specifies the id of the conversation to use. conv_id is returned by lu62_allocate or lu62_accept.

return_code

(returned) Specifies the result of verb execution from one of the following:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  — LU62_CONV_ID_UNKNOWN
- LU62_PROGRAM_STATE_CHECK
  — the conversation is not in Confirm, Confirm_Send or Confirm_Deallocate state
  — LU62_VERB_IN_PROGRESS

State Changes

The state change depends on the value of the *what_received* parameter of the preceding lu62_receive_and_wait or lu62_receive_immediate verb:

- Receive state is entered when what_received = WR_CONFIRM
- Send state is entered when what_received = WR_CONFIRM_SEND
- Deallocate state is entered when what_received = WR_CONFIRM_DEALLOCATE.

Usage Notes

The local and remote programs use the lu62_confirm and lu62_confirmed verbs to synchronize their processing. For example, the remote program can request acknowledgment that the data it sent was received by the local program. The local program issues lu62_confirmed to provide a positive acknowledgment or lu62_send_error to send a negative acknowledgment.

See Also

lu62_receive_and_wait, lu62_receive_immediate, lu62_confirm, and lu62_send_error.

Section 5.8, "Basic Conversations," illustrates how programs can be synchronized using confirmation requests.

8.4 lu62_deallocate

lu62_deallocate deallocates the specified conversation from the transaction program. The deallocation can include the function of the lu62_flush or lu62_confirm verb, depending on the value of the type parameter.

Synopsis

```
int lu62_deallocate(lu62_deallocate_t *rqp,"
```

Request Structure

```
typedef struct {
    bit32       conv_id;                           /* s */
    lu62_deallocate_type_e type;                   /* s */
    char        *log_data;                         /* s */
    bit32       return_code;                       /* r */
} lu62_deallocate_t;
```

8.4.1 `lu62_deallocate_t` *Request Structure Members*

The following subsections describe the `lu62_deallocate_t` request structure members:

`conv_id`

(supplied) Specifies the id of the conversation to use. `conv_id` is returned by `lu62_allocate` or `lu62_accept`.

`type`

(supplied) Specifies the type of deallocation to perform and can be one of the following:

- DA_SYNC_LEVEL
  Deallocation processing is dependent on the conversation synchronization level (as specified by the `sync_level` parameter of the `lu62_allocate` verb):
  - SYNC_LEVEL_NONE
    Perform the function of the `lu62_flush` verb and deallocate the conversation normally.
  - SYNC_LEVEL_CONFIRM
    Perform the function of the `lu62_confirm` verb and, if successful, deallocate the conversation normally. DA_SYNC_LEVEL is not supported when the `sync_level` is SYNC_LEVEL_SYNCPT.
- DA_FLUSH
  Perform the function of the `lu62_flush` verb and deallocate the conversation normally.

- DA_CONFIRM
  Perform the function of the lu62_confirm verb and, if successful, deallocate the conversation normally. This deallocation *type* can only be used on conversations with sync_level = SYNC_LEVEL_CONFIRM.
- DA_ABEND_PROG, DA_ABEND_SVC, DA_ABEND_TIMER
  Perform the function of the lu62_flush verb and deallocate the conversation abnormally. Logical record truncation can occur if the program is in Send state; data purging can occur in Receive state.
- DA_UNBIND
  Forces the session to be deactivated by the SunLink SNA PU2.1 9.1 server. This extension to [TPRM] is provided for SYNC_LEVEL_SYNCPT (see Appendix F).
- DA_LOCAL
  Deallocate the conversation locally. This type of deallocation can only be specified, and must be specified, if the conversation is in Deallocate state.

log_data

(supplied / conditional / optional). On deallocation of type DA_ABEND*, this parameter specifies any product-unique error information that is to be placed in the system error logs of the LUs supporting this conversation. If supplied, log_data is specified as an ASCII (null-terminated) string and is translated to EBCDIC by the SunLink SNA PU2.1 9.1 server.

return_code

(returned) Specifies the result of verb execution. return_code is dependent on the deallocation type.

When the function of the lu62_flush verb is performed (see above), return_code can be one of the following:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - LU62_CONV_ID_UNKNOWN
  - LU62_BAD_DEALLOCATE_TYPE
- LU62_PROGRAM_STATE_CHECK
  - The conversation is not in Send state.
  - The conversation is in Send state but is in the process of sending a logical record.

- — LU62_VERB_IN_PROGRESS

When the function of the lu62_confirm verb is performed (see above), return_code can be one of the following:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - — LU62_CONV_ID_UNKNOWN
- LU62_PROGRAM_STATE_CHECK
  - — The conversation is not in Send state.
  - — The conversation is in Send state but is in the process of sending a logical record.
  - — LU62_VERB_IN_PROGRESS
- LU62_ALLOCATION_ERROR
- LU62_DEALLOCATE_ABEND_PROG
- LU62_DEALLOCATE_ABEND_TIMER
- LU62_DEALLOCATE_ABEND_SVC
- LU62_PROG_ERROR_PURGING
- LU62_SVC_ERROR_PURGING
- LU62_RESOURCE_FAILURE_NO_RETRY
- LU62_RESOURCE_FAILURE_RETRY

When deallocation type is DA_ABEND* or DA_UNBIND, return_code can be one of the following:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - — LU62_CONV_ID_UNKNOWN
- LU62_PROGRAM_STATE_CHECK
  - — The conversation is not in Send, Receive, or Confirm state.
  - — LU62_VERB_IN_PROGRESS

When deallocation type is DA_LOCAL, return_code can be one of the following:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK

  - LU62_CONV_ID_UNKNOWN
- LU62_PROGRAM_STATE_CHECK
  - The conversation is not in Deallocate state.

LU62_VERB_IN_PROGRESS

State Changes

When `return_code` is LU62_OK, the conversation is Reset.

Usage Notes

1. The deallocation type DA_SYNC_LEVEL causes the conversation deallocation to be performed based on the conversation's synchronization level.
2. If the deallocation type is DA_LOCAL, or DA_SYNC_LEVEL and the `sync_level` is SYNC_LEVEL_NONE, the conversation is unconditionally deallocated. The remote program `lu62_receive*` `return_code` is LU62_DEALLOCATE_NORMAL, which causes it to enter Deallocate state. In Deallocate state, the remote program issues `lu62_deallocate`(DA_LOCAL) to end the conversation.
3. If the deallocation type is DA_CONFIRM, or DA_SYNC_LEVEL and the `sync_level` is SYNC_LEVEL_CONFIRM, the function of the `lu62_confirm` verb is performed prior to deallocation. The remote program receives `what_received` = WR_CONFIRM_DEALLOCATE, and may issue an `lu62_confirmed` verb in response. In this case the conversation is deallocated when the local LU receives the confirmation response. If, however, the remote program issues `lu62_send_error`, the conversation remains allocated.
4. The deallocation types DA_ABEND* are intended to be used to unconditionally deallocate the conversation, irrespective of its synchronization level or state. If, however, the conversation is operating in non-blocking mode, and an operation is incomplete, an attempt to `lu62_deallocate` (DA_ABEND*) will cause an LU62_PROGRAM_STATE_ERROR. In this situation use the `lu62_abort` verb to abandon the conversation.

See Also

`lu62_receive_and_wait`, `lu62_receive_immediate`, and `lu62_abort`.

8.5 lu62_flush

lu62_flush flushes the local LU's conversation send buffer. Any buffered information is sent to the remote LU. Information buffered by the LU can come from lu62_allocate (*flush* = FLUSH_NO), lu62_send_data, or lu62_send_error.

Synopsis

```
int lu62_flush(lu62_flush_t *rqp);
```

Request Structure

```
typedef struct {
    bit32       conv_id;                              /* s */
    bit32       return_code;                          /* r */
} lu62_flush_t;
```

8.5.1 lu62_flush_t *Request Structure Members*

The following subsections describe the lu62_flush_t request structure members:

conv_id

(supplied) Specifies which id conversation to use. conv_id is returned by lu62_allocate or lu62_accept.

return_code

(returned) Specifies the result of verb execution from one of the following:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - LU62_CONV_ID_UNKNOWN
  - LU62_BAD_FLUSH_TYPE
- LU62_PROGRAM_STATE_CHECK

— The conversation is not in Send state
— LU62_VERB_IN_PROGRESS

State Changes

This verb can only be issued in Send state. No state change occurs.

Usage Notes

1. Normally the LU buffers the data from consecutive `lu62_send_data` verbs until it has completely filled the current request unit (RU), or the local program issues a verb that causes an end-of-chain to be sent to the remote LU. Only then does it send the data to the remote LU. In this way transmission overhead is minimized. The `lu62_flush` verb enables the local program to force buffer transmission.
2. The LU flushes its buffer only if it has something to send. Nothing is sent if the buffer is empty.

See Also

`lu62_allocate`, `lu62_send_data`, and `lu62_send_error`.

8.6 `lu62_get_attributes`

`lu62_get_attributes` is used to provide information regarding the specified conversation.

Synopsis

```
int lu62_get_attributes(lu62_get_attributes_t *rqp);
```

Request Structure

```
typedef struct {
    bit32       conv_id;                            /* s */
    bit32       return_code;                        /* r */
    char         unique_session_name                    /* r */
                   [LU62_UNIQUE_SESSION_NAME_LEN+1];
    char        partner_lu_name[LU62_LU_NAME_LEN+1];    /* r */
    char        mode_name[LU62_MODE_NAME_LEN+1];        /* r */
    bit8        partner_qlu_name[LU62_NQ_LU_NAME_LEN+1];/* r */
    int         partner_qlu_name_len;               /* r */
    lu62_sync_level_e sync_level;                   /* r */
    lu62_conv_state_e conv_state;                   /* r */
    int         conv_corr_len;                      /* r */
    bit8        conv_corr[LU62_MAX_CONV_CORR_LEN];       /* r */
    bit32       conv_grp_id;                        /* r */
    int         sess_id_len;                        /* r */
    bit8        sess_id[LU62_MAX_SESS_ID_LEN];      /* r */
    int         luw_len;                            /* r */
    bit8        luw[LU62_MAX_LUW_LEN];              /* r */
} lu62_get_attributes_t;
```

8.6.1 `lu62_get_attributes_t` Request Structure Members

The following subsections describe the `lu62_get_attributes_t` request structure members:

`conv_id`

(supplied) Specifies the id of the conversation to use. `conv_id` is returned by `lu62_allocate or lu62_accept`.

`return_code`

(returned) Specifies the result of verb execution as one of the following:

- `LU62_OK`
- `LU62_PARAMETER_CHECK`
  - — `LU62_CONV_ID_UNKNOWN`

unique_session_name

Used to specify the actual node used from the configuration instead of lu_name and *node*. Furthermore to use unique session names, the TP cannot have used an lu_name in the previous open.

partner_lu_name

(returned) Specifies the name of the partner LU at which the remote transaction program is located. partner_lu_name corresponds to the PTNR_LU NAME parameter in the configuration.

mode_name

(returned) Specifies the name of the selected mode. The conversation is allocated on a session of this mode. mode_name corresponds to the MODE NAME parameter in the configuration.

partner_qlu_name_len

Length of partner_qlu_len in bytes.

partner_qlu_name

(returned) Specifies the fully qualified name of the partner LU at which the remote transaction program is located. partner_qlu_name corresponds to the PTNR_LU NQ_LU_NAME parameter in the configuration.

sync_level

(returned) Specifies the synchronization level for the conversation as one of the following:

- SYNC_LEVEL_NONE
- SYNC_LEVEL_CONFIRM
- SYNC_LEVEL_SYNCPT

conv_state

(returned) Specifies the current state of the conversation as one of the following:

- CONV_RESET
- CONV_SEND

- CONV_RECEIVE
- CONV_CONFIRM
- CONV_CONFIRM_SEND
- CONV_CONFIRM_DEALLOCATE
- CONV_DEALLOCATE

conv_grp_id

(ignored) Reserved for future use.

conv_corr_len
conv_corr

(ignored) Reserved for future use.

sess_id_len

Length of session id.

sess_id

(returned) Returns the assigned session identifier. The *sess_id* is returned as binary data. (Contrast this to `lu62_display_mode` and `lu62_deactivate_session` in which `session_id` is an ASCII-hex string).

*luw_len*luw

(returned) This extension to [TPRM] is provided for `SYNC_LEVEL_SYNCPT` (see Appendix F). `luw` contains the complete Logical Unit of Work Identifier as set in the FMH-5 Attach that initiated the conversation (see Chapter 11 of the *IBM SNA Formats* manual.

State Changes

No state change occurs.

See Also

Section 5.4, “Accepting Conversations,” provides an example of the use of this verb.

8.7 lu62_post_on_receipt

lu62_post_on_receipt causes the LU to post the specified conversation when information is available to be received. The information can be data, conversation status, or a request for confirmation. When the conservation is posted, the information is retrieved using lu62_receive_and_wait or lu62_receive_immediate. Programs can issue the lu62_wait verb to wait for posting to occur. Alternatively, programs can issue lu62_test to poll a conversation to see if it is posted.

Synopsis

```
int lu62_post_on_receipt(lu62_post_on_receipt_t *rqp);
```

Request Structure

```
typedef struct {
    bit32       conv_id;                                    /* s */
    int         length;                                     /* s */
    lu62_fill_e fill;                                       /* s */
    bit32       return_code;                                /* r */
} lu62_post_on_receipt_t;
```

8.7.1 lu62_post_on_receipt_t *Request Structure Members*

The following subsections describe the lu62_post_on_receipt_t request structure members:

conv_id

(supplied) Specifies the id of the conversation to use. conv_id is returned by lu62_allocate or lu62_accept.

length

(supplied) Specifies the maximum amount of data the program can receive.

`fill`

(supplied) Specifies when posting is to occur as one of the following:

- `FILL_LL`
  Posting occurs when a complete or truncated logical record is received, or the length value is satisfied, whichever occurs first.
- `FILL_BUFFER`
  Posting occurs when length data is received or end of data occurs, whichever occurs first.

`return_code`

(returned) Specifies the result of verb execution as one of the following:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  - `LU62_CONV_ID_UNKNOWN`
  - `LU62_BAD_LENGTH`
  - `LU62_BAD_FILL_TYPE`
- `LU62_PROGRAM_STATE_CHECK`
  - the conversation is not is Receive state
  - `LU62_VERB_IN_PROGRESS`

State Changes

This verb can only be issued in Receive state. No state change occurs.

Usage Notes

1. The `lu62_post_on_receipt, lu62_wait`, and `lu62_test` verbs together, provide the architected solution for handling multiple conversations in a non-blocking manner as mentioned in the *IBM SNA Transaction Programmer's Reference Manual*. This solution, however, handles receive processing only. All conversations must still wait for one conversation to confirm an operation, or for another to deallocate. An alternative approach is to use `lu62_set_processing_mode` to set your conversations into `PM_NON_BLOCKING` mode. In this mode, all verbs that require interaction with the SunLink SNA PU2.1 9.1 server return as `LU62_OPERATION_INCOMPLETE` as soon as a request is sent to the server. Other conversations can then be processed. When the program is ready,

it issues `lu62_wait_server` to wait for an outstanding operation to complete. Thus `lu62_receive_and_wait` may be used to wait for conversation data or status, without blocking other conversations.

2. Posting occurs when the LU has any information that would satisfy a receive verb (issued with the same length and fill parameters). Refer to `lu62_receive_and_wait` for a description of what information can be received (`what_received`).

3. Posting remains in effect until the conversation is posted, posting is reset, or posting is canceled.

Posting is reset when one of the following verbs is issued on the conversation after the conversation is posted:

- `lu62_deallocate (DA_ABEND*)`
- `lu62_receive_and_wait`
- `lu62_receive_immediate`
- `lu62_send_error`
- `lu62_test`
- `lu62_wait`

Posting is canceled when any of the following verbs is issued on the conversation *before* the conversation is posted:

- `lu62_deallocate (DA_ABEND*)`
- `lu62_receive_and_wait`
- `lu62_receive_immediate`
- `lu62_send_error`

See Also

`lu62_receive_and_wait`, `lu62_receive_immediate`, `lu62_test`, `lu62_wait`, `lu62_wait_server`

8.8 lu62_prep_to_receive

`lu62_prep_to_receive` changes the conversation from Send to Receive state, in preparation to receive data.

Synopsis

```
int lu62_prep_to_receive(lu62_prep_to_receive_t *rqp);
```

Request Structure

```
typedef struct {
    bit32       conv_id;                                 /* s */
    lu62_prep_to_rcv_type_e type;                        /* s */
    lu62_locks_elocks;                                   /* s */
    bit32       return_code;                             /* r */
} lu62_prep_to_receive_t;
```

8.8.1 `lu62_prep_to_receive_t` *Request Structure Members*

The following subsections describe the request structure members of the `lu62_prep_to_receive_t`.

`conv_id`

(supplied) Specifies the id of the conversation to use. `conv_id` is returned by `lu62_allocate` or `lu62_accept`.

`type`

(supplied) Specifies the type of prepare to receive, to perform one of:

- PR_SYNC_LEVEL
  Processing is dependent on the conversation synchronization level (as specified by the *sync*`_level` parameter of the `lu62_allocate` verb):
  - SYNC_LEVEL_NONE
    Perform the function of the `lu62_flush` verb and enter Receive state.
  - SYNC_LEVEL_CONFIRM
    Perform the function of the `lu62_confirm` verb and, if successful, enter Receive state. PR_SYNC_LEVEL is not supported when the `sync_level` is SYNC_LEVEL_SYNCPT.

- `PR_FLUSH`
  Perform the function of the `lu62_flush` verb and enter Receive state.

- `PR_CONFIRM`
  Perform the function of the `lu62_confirm` verb and, if successful, enter Receive state. This type can only be used on conversations with `sync_level =SYNC_LEVEL_CONFIRM`.

`locks`

(supplied/conditional) Specifies when control is to be returned to the local program. This parameter is only relevant if `type` = `PR_CONFIRM`. It may be one of:

- `LOCKS_SHORT`
  Control is returned immediately after the confirmation response is received from the remote program.
- `LOCKS_LONG`
  Control is returned when information, such as data or status, is received from the remote program following the confirmation response.

`return_code`

(returned) Specifies the result of verb execution. `return_code` is dependent on the type.

The following `return_codes` can be returned for all values of the type parameter:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  - `LU62_CONV_ID_UNKNOWN`
  - `LU62_BAD_PREP_TO_RCV_TYPE`
  - `LU62_BAD_LOCKS_TYPE`
- `LU62_PROGRAM_STATE_CHECK`
  - The conversation is not in Send state
  - The conversation is in Send state but is in the process of sending a logical record

— LU62_PIP_PENDING - the conversation is in Send state following an lu62_(mc_)_allocate verb in which pip_presence was indicated. A basic send, lu62_send_data, is required to send the PIP Variable.

— LU62_VERB_IN_PROGRESS

When the function of the lu62_confirm verb is performed (see above), additional return_codes are possible:

- LU62_OK
- LU62_ALLOCATION_ERROR
- LU62_DEALLOCATE_ABEND_PROG
- LU62_DEALLOCATE_ABEND_TIMER
- LU62_DEALLOCATE_ABEND_SVC
- LU62_PROG_ERROR_PURGING
- LU62_SVC_ERROR_PURGING
- LU62_RESOURCE_FAILURE_NO_RETRY
- LU62_RESOURCE_FAILURE_RETRY

State Changes

When return_code is LU62_OK, the conversation is in Receive state.

Usage Notes

1. When type = PR_SYNC_LEVEL, send control is transferred to the remote program based on the synchronization level of the conversation. Thus, if the synchronization level is SYNC_LEVEL_CONFIRM, a confirmation response is required before handing over send control.

2. When type = PR_FLUSH, or type = PR_SYNC_LEVEL and the synchronization level is SYNC_LEVEL_NONE, send control is transferred to the remote program without a confirmation. The remote program's lu62_receive* verb returns with a what_received value of WR_SEND.

3. When type = PR_CONFIRM, or type = PR_SYNC_LEVEL and the synchronization level is SYNC_LEVEL_CONFIRM, a confirmation response is required before handing over send control. The remote program's lu62_receive* verb returns with a what_received value of WR_CONFIRM_SEND.

See Also

`lu62_confirmed`, `lu62_receive_and_wait`, `lu62_receive_immediate`

8.9 `lu62_receive_and_wait`

`lu62_receive_and_wait` waits for information to be received on the specified conversation. The information can be data, conversation status, or a request for confirmation—an indication of the type of information received is returned.

Synopsis

```
int lu62_receive_and_wait(lu62_receive_t *rqp);
```

Request Structure

```
typedef struct {
    bit32       conv_id;                            /* s */
    lu62_fill_e fill;                               /* s */
    int         length;                             /* s/r */
    bit32       return_code;                        /* r */
    bit32       request_to_send_received;           /* r */
    bit8        *data;                              /* r */
    lu62_what_received_e what_received;             /* r */
    char        map_name[LU62_MAP_NAME_LEN+1];      /* r */
} lu62_receive_t;
```

8.9.1 `lu62_receive_t` *Request Structure Members*

The following subsections describe the `lu62_receive_t` request structure members:

`conv_id`

(supplied) Specifies the id of the conversation to use. `conv_id` is returned by `lu62_allocate` or `lu62_accept`.

`fill`

(supplied) Specifies whether the program requires to receive data as a logical record, or independently of logical record format. `fill` may be one of:

- `FILL_LL`
  The verb returns when a complete or truncated logical record is received, or the length value is satisfied, whichever occurs first.
- `FILL_BUFFER`
  The verb returns when length data is received or end of data occurs, whichever occurs first.

`length`

(supplied / returned) On input, this parameter specifies the maximum amount of data the program can receive. On return, and if data is received, the parameter is set with the amount of data received. If no data is received, this parameter is unchanged. `return_code` (returned) specifies the result of verb execution, one of the following:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  - `LU62_CONV_ID_UNKNOWN`
  - `LU62_BAD_LENGTH`
  - `LU62_BAD_FILL_TYPE`
  - `LU62_NULL_DATA`
- `LU62_PROGRAM_STATE_CHECK`
  - The conversation is not in Send or Receive state.
  - The conversation is in Send state but is in the process of sending a logical record.
  - `LU62_PIP_PENDING` - the conversation is in Send state following an `lu62_(mc_)_allocate` verb in which `pip_presence` was indicated. A basic send, `lu62_send_data`, is required to send the PIP Variable.
  - `LU62_VERB_IN_PROGRESS`
- `LU62_ALLOCATION_ERROR`
- `LU62_DEALLOCATE_ABEND_PROG`
- `LU62_DEALLOCATE_ABEND_TIMER`
- `LU62_DEALLOCATE_ABEND_SVC`

- LU62_DEALLOCATE_NORMAL
- LU62_PROG_ERROR_NO_TRUNC
- LU62_PROG_ERROR_PURGING
- LU62_PROG_ERROR_TRUNC (only in Receive state)
- LU62_SVC_ERROR_NO_TRUNC
- LU62_SVC_ERROR_PURGING
- LU62_SVC_ERROR_TRUNC (only in Receive state)
- LU62_RESOURCE_FAILURE_NO_RETRY
- LU62_RESOURCE_FAILURE_RETRY.

request_to_send_received

(returned) Indicates whether or not a request-to-send indication is received from the remote program:

- TRUE, indicates that a request-to-send indication was received.
- FALSE, indicates that a request-to-send indication was not received.

data

(supplied / returned) Specifies the buffer into which any received data is to be written. The buffer should be at least length bytes long. If what_received indicates that information other than data was received, nothing is written into this buffer.

what_received

(returned) Indicates the type of information that was received from one of the following:

- WR_DATA
  Indicated when FILL_BUFFER is specified and data is received.
- WR_DATA_COMPLETE
  Indicated when FILL_LL is specified and a complete logical record is received (or the remaining portion thereof).
- WR_DATA_INCOMPLETE
  Indicated when FILL_LL is specified and less than a complete logical record is received. The local program must issue at least one more lu62_receive_and_wait verb to receive the remaining data.

- *WR_PS_DATA_COMPLETE
  Indicated when FILL_LL is specified and a PS Header is received on a SYNC_LEVEL_SYNCPT conversation (or the remaining portion thereof). This extension to the *IBM SNA Transaction Programmer's Reference Manual* is provided for SYNC_LEVEL_SYNCPT (see Appendix F).
- *WR_PS_DATA_INCOMPLETE
  Indicated when FILL_LL is specified and less than a complete PS Header is received on a SYNC_LEVEL_SYNCPT conversation. The local program must issue at least one more lu62_receive_and_wait verb to receive the remaining data. This extension is provided for SYNC_LEVEL_SYNCPT (see Appendix F).
- WR_LL_TRUNCATED
  Indicated when FILL_LL is specified and the 2 bye LL field of a logical record is truncated after the first byte. The local program should issue another lu62_receive_and_wait verb to determine why the truncation occurred (the remote program or LU has issued lu62_send_error or lu62_deallocate (type = DA_ABEND*).
- WR_SEND
  Indicates that the remote program has entered Receive state. The local program transitions to Send state.
- WR_CONFIRM
  Indicates that the remote program has issued lu62_confirm. The local program may respond by issuing lu62_confirmed.
- WR_CONFIRM_SEND
  Indicates that the remote program requires a confirmation response before entering Receive state. The local program may respond by issuing lu62_confirmed.
- WR_CONFIRM_DEALLOCATE
  Indicates that the remote program requires a confirmation response before deallocating the conversation. The local program may terminate the conversation by issuing lu62_confirmed.

map_name

(ignored) Mapped conversations only.

State Changes

If the return_code is LU62_OK, the state changes according to the initial state and the value of the what_received parameter:

- Receive state is entered when the verb is issued in Send state and what_received = WR_DATA, WR_DATA_COMPLETE, WR_DATA_INCOMPLETE, WR_PS_DATA_COMPLETE, WR_PS_DATA_INCOMPLETE, or WR_LL_TRUNCATED.
- Send state is entered when what_received = WR_SEND.
- Confirm state is entered when what_received = WR_CONFIRM, WR_CONFIRM_SEND, or WR_CONFIRM_DEALLOCATE.

No state change occurs when the verb is issued in Receive state and what_received = WR_DATA, WR_DATA_COMPLETE, WR_DATA_INCOMPLETE, WR_PS_DATA_COMPLETE, WR_PS_DATA_INCOMPLETE or WR_LL_TRUNCATED.

Usage Notes

1. lu62_receive_and_wait receives only one type of information at a time. It may receive data, status, or a confirmation request, as indicated by the value of what_received.
2. When lu62_receive_and_wait is issued in Send state, an implicit lu62_prep_to_receive (PR_FLUSH) is executed by the local LU.
3. lu62_receive_and_wait includes posting. If posting is already active, the post conditions (length, fill) are superceded by those specified by this verb.
4. When fill = FILL_LL and what_received = WR_DATA_INCOMPLETE, either the length of the logical record exceeds the maximum length of the user's data buffer, or the logical record has been truncated by the remote program issuing a lu62_send_error, or lu62_deallocate (DA_ABEND*). The local program must issue another lu62_receive_and_wait to determine which of the aforementioned conditions occur first.
5. When fill = FILL_BUFFER and the length received is less than that requested, then the LU must have received end-of-chain. The end-of-chain condition corresponds to a change of state of the remote program, that is, a change to Send, Confirm or Deallocate states.
6. The request-to-send notification is usually received when the local program is in Send state, and is reported to the program via the request_to_send_received parameter of the lu62_send_data or

`lu62_send_error` verb. The notification can also be received, however, when the conversation is in Receive state. This can occur under three different circumstances:

- When the local program enters Receive state and the remote program issues `lu62_request_to_send` before it enters Send state.
- When the remote program enters Receive state using `lu62_prep_to_receive` (not `lu62_receive_and_wait`), and then issues `lu62_request_to_send` before the local program enters Send state. This can occur because the request-to-send is transmitted as an expedited request and can therefore arrive ahead of the request, carrying the send indication. Potentially, the local program cannot distinguish this condition from the first. This ambiguity is avoided if the remote program waits until it receives information from the local program before it issues `lu62_request_to_send`.
- When the remote program issues `lu62_request_to_send` in Send state. This can be used to signal the local program that data is about to be sent. The local program issues `lu62_test` (`test` = `TEST_REQUEST_TO_SEND_RECEIVED`) to poll the local LU for this situation. Only when the result is TRUE does the local program issue `lu62_receive_and_wait.`

See Also

`lu62_post_on_receipt`

Section 5.8, “Basic Conversations,” illustrates the use of this verb.

8.10 `lu62_receive_immediate`

`lu62_receive_immediate` requests any information that is available for the specified conversation. In contrast to `lu62_receive_and_wait`, it does not wait for information to arrive. The information can be data, conversation status, or a request for confirmation, an indication of the type of information received is returned.

Synopsis

```
int lu62_receive_immediate(lu62_receive_t *rqp);
```

Request Structure

```
typedef struct {
    bit32       conv_id;                                /* s */
    lu62_fill_e fill;                                   /* s */
    int         length;                               /* s/r */
    bit32       return_code;                            /* r */
    bit32       request_to_send_received;               /* r */
    bit8        *data;                                  /* r */
    lu62_what_received_e what_received;                  /* r */
    char        map_name[LU62_MAP_NAME_LEN+1];          /* r */
} lu62_receive_t;
```

8.10.1 `lu62_receive_t` *Request Structure Members*

The following subsections describe the `lu62_receive_t` request structure members.

`conv_id`

(supplied) Specifies which id conversation to use. `conv_id` is returned by `lu62_allocate` or `lu62_accept`.

`fill`

(supplied) Specifies whether the program requires to receive data as a logical record, or independently of a logical record format. `fill` may be:

- `FILL_LL`
  The verb returns when a complete or truncated logical record is received, or the length value is satisfied, whichever occurs first.
- `FILL_BUFFER`
  The verb returns when length data is received or end-of-data occurs, whichever occurs first.

`length`

(supplied / returned) On input, this parameter specifies the maximum amount of data the program can receive. On return, and if data is received, the parameter is set with the amount of data received. If no data is received, this parameter is unchanged.

`return_code`

(returned) Specifies the result of verb execution:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  - — `LU62_CONV_ID_UNKNOWN`
  - — `LU62_BAD_LENGTH`
  - — `LU62_BAD_FILL_TYPE`
  - — `LU62_NULL_DATA`
- `LU62_PROGRAM_STATE_CHECK`
  - — the conversation is not in Receive state
  - — `LU62_VERB_IN_PROGRESS`
- `LU62_ALLOCATION_ERROR`
- `LU62_DEALLOCATE_ABEND_PROG`
- `LU62_DEALLOCATE_ABEND_TIMER`
- `LU62_DEALLOCATE_ABEND_SVC`
- `LU62_DEALLOCATE_NORMAL`
- `LU62_PROG_ERROR_NO_TRUNC`
- `LU62_PROG_ERROR_PURGING`
- `LU62_PROG_ERROR_TRUNC` (only in Receive state)
- `LU62_SVC_ERROR_NO_TRUNC`
- `LU62_SVC_ERROR_PURGING`
- `LU62_SVC_ERROR_TRUNC` (only in Receive state)
- `LU62_RESOURCE_FAILURE_NO_RETRY`
- `LU62_RESOURCE_FAILURE_RETRY`
- `LU62_UNSUCCESSFUL` (there is nothing to receive)

request_to_send_received

(returned) Indicates whether or not a request-to-send indication is received from the remote program:

- TRUE, indicates that a request-to-send indication was received.
- FALSE, indicates that a request-to-send indication was been received.

data

(supplied / returned) Specifies the buffer into which any received data is to be written. The buffer should be at least as many bytes long as specified in the lu62_receive_t request structure. If what_received indicates that information other than data has been received, nothing is written into this buffer.

what_received

(returned) Indicates the type of information that is received:

- WR_DATA
  Is indicated when FILL_BUFFER is specified and data is received.
- WR_DATA_COMPLETE
  Is indicated when FILL_LL is specified and a complete logical record is received (or the remaining portion thereof).
- WR_DATA_INCOMPLETE
  Is indicated when FILL_LL is specified and less than a complete logical record is received. The local program must issue at least one more lu62_receive_immediate verb to receive the remaining data.
- WR_PS_DATA_COMPLETE
  Is indicated when FILL_LL is specified and a PS header is received on a SYNC_LEVEL_SYNCPT conversation (or the remaining portion thereof). This extension to the *IBM SNA Transaction Programmer's Reference Manual* is provided for SYNC_LEVEL_SYNCPT (see Appendix F).
- *WR_PS_DATA_INCOMPLETE
  Is indicated when FILL_LL is specified and less than a complete PS header is received on a SYNC_LEVEL_SYNCPT conversation. The local program must issue at least one more lu62_receive_immediate verb to receive the remaining data. This extension to *IBM SNA Transaction Programmer's Reference Manual* is provided for SYNC_LEVEL_SYNCPT (see Appendix F).

- WR_LL_TRUNCATED
  Is indicated when FILL_LL is specified and the 2-bye LL field of a logical record is truncated after the first byte. The local program should issue another lu62_receive_immediate verb to determine why the truncation occurred (the remote program or LU issued lu62_send_error or lu62_deallocate (type = DA_ABEND*).
- WR_SEND
  Indicates that the remote program has entered Receive state. The local program transitions to Send state.
- WR_CONFIRM
  Indicates that the remote program has issued lu62_confirm. The local program may respond by issuing lu62_confirmed.
- WR_CONFIRM_SEND
  Indicates that the remote program requires a confirmation response before entering Receive state. The local program may respond by issuing lu62_confirmed.
- WR_CONFIRM_DEALLOCATE
  Indicates that the remote program requires a confirmation response before deallocating the conversation. The local program may terminate the conversation by issuing lu62_confirmed.

map_name

(ignored) Mapped conversations only.

State Changes

If the return_code is LU62_OK, the state changes according to the initial state and the value of the what_received parameter:

- Receive state is entered when the verb is issued in Send state and what_received = WR_DATA, WR_DATA_COMPLETE, WR_DATA_INCOMPLETE, WR_PS_DATA_COMPLETE, WR_PS_DATA_INCOMPLETE, or WR_LL_TRUNCATED.
- Send state is entered when what_received = WR_SEND.
- Confirm state is entered when what_received = WR_CONFIRM, WR_CONFIRM_SEND, or WR_CONFIRM_DEALLOCATE.

No state change occurs when the verb is issued in Receive state and what_received = WR_DATA, WR_DATA_COMPLETE, WR_DATA_INCOMPLETE, WR_PS_DATA_COMPLETE, WR_PS_DATA_INCOMPLETE, or WR_LL_TRUNCATED.

Usage Notes

1. lu62_receive_immediate receives only one type of information at a time. It may receive data, status, or a confirmation request, as indicated by the value of what_received.

2. lu62_receive_immediate resets or cancels posting. If posting is active and the conversation has been posted, posting is reset. If posting is active and the conversation is not posted, posting is canceled.

3. When fill = FILL_LL and what_received= WR_DATA_INCOMPLETE, either the length of the logical record exceeds the maximum length of the user's data buffer, or the logical record is truncated by the remote program issuing a lu62_send_error or lu62_deallocate (DA_ABEND*). The local program must issue another lu62_receive_immediate to determine whichever of the aforementioned conditions occurs first.

4. When fill = FILL_BUFFER, the program receives whatever data is available, up to the amount requested by the length parameter. If the length received is less than that requested, then either the LU received the end-of-chain or less than the requested amount of data is available. The end-of-chain condition corresponds to a change of state of the remote program, that is, a change to Send, Confirm or Deallocate states.

5. The request-to-send notification is usually received when the local program is in Send state, and is reported to the program via the request_to_send_received parameter of the lu62_send_data or lu62_send_error verb. The notification can also be received, however, when the conversation is in Receive state. This can occur under three different circumstances:
   - When the local program enters Receive state and the remote program issues lu62_request_to_send before it enters Send state.
   - When the remote program enters Receive state using lu62_prep_to_receive (not lu62_receive_and_wait), and then issues lu62_request_to_send before the local program enters Send state. This can occur because the request-to-send is transmitted as an

expedited request and can therefore arrive ahead of the request that carries the send indication. Potentially, the local program cannot distinguish this condition from the first. This ambiguity is avoided if the remote program waits until it receives information from the local program before it issues `lu62_request_to_send`.

- When the remote program issues `lu62_request_to_send` in Send state. This can be used to signal the local program that data is about to be sent. The local program issues `lu62_test` (`TEST_REQUEST_TO_SEND_RECEIVED`) to poll the local LU for this situation. Only when the result is TRUE does the local program issue `lu62_receive_immediate`.

See Also

`lu62_post_on_receipt`

8.11 `lu62_request_to_send`

`lu62_mc_request_to_send` sends a notification to the remote program to indicate that the local program would like to enter Send state. The conversation remains in its current state, however, until a send indication is received from the remote program.

Synopsis

```
int lu62_request_to_send(lu62_request_to_send_t *rqp);
```

Request Structure

```
typedef struct {
    bit32       conv_id;                              /* s */
    bit32       return_code;                          /* r */
} lu62_request_to_send_t;
```

8.11.1 `lu62_request_to_send_t` *Request Structure Members*

The following subsections describe the `lu62_request_to_send_t` request structure members:

`conv_id`

(supplied) Specifies the id of the conversation to be used. `conv_id` is returned by `lu62_allocate` or `lu62_accept`.

`return_code`

(returned) Specifies the result of verb execution, which is one of the following:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
    - `LU62_CONV_ID_UNKNOWN`
- `LU62_PROGRAM_STATE_CHECK`
    - The conversation is not is Receive, Confirm, or Send state
    - `LU62_VERB_IN_PROGRESS`

State Changes

No state change occurs.

Usage Notes

1. The remote program is informed of the arrival of a request-to-send notification by means of the `request_to_send_received` parameter returned by `lu62_confirm`, `lu62_receive_and_wait`, `lu62_receive_immediate`, `lu62_send_data`, and `lu62_send_error`. The remote program may also poll the LU to determine if a request-to-send notification was received using `lu62_test` (`TEST_REQUEST_TO_SEND_RECEIVED`). When the remote program receives the request-to-send notification, it issues `lu62_receive_and_wait` or `lu62_prep_to_receive` to enter Receive state and thereby places the local program in Send state. The local program enters Send state when it issues an `lu62_receive_and_wait` or `lu62_receive_immediate` and receives the send indication.

2. The remote LU retains the request-to-send notification until the remote program issues one of the verbs identified above. Additional request-to-send notifications are discarded until the retained notification is passed to the remote program.

See Also

`lu62_confirm`, `lu62_prep_to_receive`, `lu62_receive_and_wait`, `lu62_receive_immediate`, `lu62_send_data`, `lu62_send_error`, `lu62_test`

`lu62_send_data`, which is used to send data to the remote program. The data consists of logical records. The amount of data is specified independently of the data format.

Synopsis

```
int lu62_send_data(lu62_send_data_t *rqp);
```

Request Structure

```
typedef struct {
    bit32          conv_id;                          /* s */
    bit8           *data;                            /* s */
    int            length;                           /* s */
    char           map_name[LU62_MAP_NAME_LEN+1];    /* s */
    lu62_fmh_data_e fmh_data;                        /* s */
    lu62_encrypt_e encrypt;                          /* s */
    lu62_flush_eflush;                               /* s */
    bit32          return_code;                      /* r */
    bit32          request_to_send_received;         /* r */
} lu62_send_data_t;
```

8.11.2 `lu62_send_data_t` *Request Structure Members*

The following subsections describe the `lu62_send_data_t` request structure members:

`conv_id`

(supplied) Specifies the id of the conversation to use. `conv_id` is returned by `lu62_allocate` or `lu62_accept`.

`data`

(supplied) The address of the user buffer containing the data to be sent. The data is structured in logical records, each comprising a two-byte LL field followed by a data field. The length of the data can range from 0 to 32765 bytes. The LL field contains the 15-bit binary length of the record. The high order bit is passed transparently by the sending LU and is used to support the mapped conversations. The length of the record includes the length of the LL field.

`length`

(supplied) Specifies the length of the data to be sent. This data length is not related to the length of a logical record (unless the data field contains one and only one complete logical record). A data length of zero is permitted. No data is sent, but all other parameters retain their meaning.

`map_name`

(ignored) Mapped conversations only.

`fmh_data`

(ignored) Mapped conversations only.

`encrypt`

(ignored) Reserved for future use.

`flush`

(supplied) Specifies whether the supplied data is to be sent to the remote program immediately or buffered in the local LU's send buffer. *flush* is set to one of the following:

- `FLUSH_NO`
- `FLUSH_YES`

`return_code`

(returned) Specifies the result of verb execution, one of the following:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  - `LU62_CONV_ID_UNKNOWN`
  - `LU62_NULL_DATA`
  - `LU62_BAD_LENGTH`
- `LU62_PROGRAM_STATE_CHECK`
  - The conversation is not is Send state.
  - `LU62_VERB_IN_PROGRESS`
- `LU62_ALLOCATION_ERROR`
- `LU62_DEALLOCATE_ABEND_PROG`
- `LU62_DEALLOCATE_ABEND_TIMER`
- `LU62_DEALLOCATE_ABEND_SVC`
- `LU62_PROG_ERROR_PURGING`
- `LU62_SVC_ERROR_PURGING`
- `LU62_RESOURCE_FAILURE_NO_RETRY`
- `LU62_RESOURCE_FAILURE_RETRY`

`request_to_send_received`

(returned) Indicates whether or not a request-to-send indication is received from the remote program:

- TRUE, indicates that a request-to-send indication was received.
- FALSE, indicates that a request-to-send indication was not received.

State Changes

No state change occurs.

Usage Notes

1. The data sent by the program during a basic conversation consists of logical records. The logical records are independent of the length of data as specified by the length parameter. The data can contain one or more complete records, the beginning of a record, the middle of a record, or the end of a record. The following combinations of data are also possible:
   - One or more complete records, followed by the beginning of a record
   - The end of a record, followed by one or more complete records
   - The end of a record, followed by one or more complete records, followed by the beginning of a record; and the end of a record, followed by the beginning of a record

2. The program must finish sending a logical record before issuing any of the following calls:
   - `lu62_confirm`
   - `lu62_deallocate, DA_FLUSH, DA_CONFIRM`, or `DA_SYNC_LEVEL`
   - `lu62_prep_to_receive`
   - `lu62_receive_and_wait`

   A program finishes sending a logical record when it sends a complete record or when it truncates an incomplete record.

3. A complete logical record contains the 2-byte-long LL field and all bytes of the data field, as determined by the logical-record length. If the data field length is zero, the complete logical record contains only the 2-byte-long length field. An incomplete logical record consists of any amount of data less than a complete record. It can consist of only the first byte of the LL field, the 2-byte-long LL field plus all of the data field except the last byte, or any amount between. A logical record is incomplete until the last byte of the data field is sent, or until the second byte of the LL field is sent if the data field is zero.

4. A program can truncate an incomplete logical record by issuing the `lu62_send_error` verb. `lu62_send_error` causes the LU to flush its send buffer, which includes sending the truncated record. The LU then treats the first two bytes of data specified in the next `lu62_send_data` as the LL field. Issuing `lu62_deallocate` with `type` set to `DA_ABEND*` also truncates an incomplete logical record.

5. The local LU buffers the data to be sent to the remote LU until it accumulates a sufficient amount of data for transmission (from one or more `lu62_send_data` verbs), or until the local program issues a call that causes the LU to flush its send buffer. The amount of data sufficient for transmission depends on the characteristics of the session allocated for the conversation, and varies from one session to another.

6. When `request_to_send_received` is TRUE, the remote program requests that the local program "give up the turn", that is, enter Receive state, thereby placing the remote program in Send state. The local program enters Receive state by issuing `lu62_prep_to_receive or lu62_receive_and_wait`. The remote program issues `lu62_receive_immediate` or `lu62_receive_and_wait` to receive the resulting Send indication (`what_received` = `WR_SEND`).

See Also

`lu62_send_error`

Section 5.8, "Basic Conversations," illustrates the use of this verb.

8.12 `lu62_send_error`

`lu62_send_error` is used by a program to inform the remote program that the local program detected an error during a conversation. If the conversation is in Send state, `lu62_send_error` forces the LU to flush its send buffer.

When this call completes successfully, the local program is in Send state and the remote program is in Receive state. Further action is defined by program logic.

Synopsis

```
int lu62_send_error(lu62_send_error_t *rqp);
```

Request Structure

```
typedef struct {
    bit32          conv_id;                      /* s */
    lu62_prog_type_etype;                        /* s */
    char           *log_data;                    /* s */
    int            error_direction;              /* s */
    bit32          return_code;                  /* r */
    bit32          request_to_send_received;     /* r */
} lu62_send_error_t;
```

8.12.1 `lu62_send_error_t` *Request Structure Members*

The following subsections describe the `lu62_send_error_t` request structure members:

`conv_id`

(supplied) Specifies the id of the conversation to be used. `conv_id` is returned by `lu62_allocate or lu62_accept`.

`type`

(supplied) Specifies the program type, one of:

- `PROG`—Indicates that an end-user application error is being reported.
- `PROG_SVC`—Indicates that an LU services error is being reported.

`log_data`

(supplied/optional) If supplied, this parameter specifies any product-unique error information that is to be placed in the system error logs of the LUs that support this conversation. If supplied, `log_data` is specified as an ASCII (null-terminated) string and is translated to EBCDIC by the SunLink SNA PU2.1 9.1 server.

`error_direction`

(ignored) Reserved for use by CPI-C.

return_code

(returned) Specifies the result of verb execution. return_code may be one of the following:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - — LU62_CONV_ID_UNKNOWN
  - — LU62_BAD_PROG_TYPE
  - — LU62_BAD_LOG_DATA
- LU62_PROGRAM_STATE_CHECK
- The conversation is not is Send, Receive, or Confirm state.
- LU62_VERB_IN_PROGRESS

If the verb is issued in Send state, return_code can additionally be one of the following:

- LU62_ALLOCATION_ERROR
- LU62_DEALLOCATE_ABEND_PROG
- LU62_DEALLOCATE_ABEND_TIMER
- LU62_DEALLOCATE_ABEND_SVC
- LU62_PROG_ERROR_PURGING
- LU62_SVC_ERROR_PURGING
- LU62_RESOURCE_FAILURE_NO_RETRY
- LU62_RESOURCE_FAILURE_RETRY

If the verb is issued in Receive state, return_code can additionally be:

- LU62_DEALLOCATE_NORMAL
- LU62_RESOURCE_FAILURE_NO_RETRY
- LU62_RESOURCE_FAILURE_RETRY

request_to_send_received

(returned) Indicates whether or not a request-to-send indication is received from the remote program:

- TRUE, indicates that a request-to-send indication was received
- FALSE, indicates that a request-to-send indication was not received

State Changes

When `return_code` is `LU62_OK`:

- Send state is entered when the verb is issued in Receive or Confirm states.
- No state change occurs when the verb is issued in Send state.

Usage Notes

1. The LU sends the error notification to the remote LU immediately (during the processing of this call).

2. Log data is buffered by the local LU until it is implicitly or explicitly flushed.

3. The issuance of `lu62_send_error` is reported to the remote program as one of the following return codes:
   - `LU62_PROG_ERROR_TRUNC` or `LU62_SVC_ERROR_TRUNC`

     The local program issued `lu62_send_error` in Send state after sending an incomplete logical record (see `lu62_send_data`). The record is truncated.
   - `LU62_PROG_ERROR_NO_TRUNC` or `LU62_SVC_ERROR_NO_TRUNC`

     The local program issued `lu62_send_error` in Send state after sending a complete logical record; or before sending any record. No truncation occurs.
   - `LU62_PROG_ERROR_PURGING` or `LU62_SVC_ERROR_PURGING`

     The local program issued `lu62_send_error` in Receive state. All information sent by the remote program and not yet received by the local program is purged; or the local program issues `lu62_send_error` in Confirm state, in which case no purging occurs.

4. When `lu62_send_error` is issued in Receive state, incoming information is also purged. Because of this purging, the `return_code` of `LU62_DEALLOCATE_NORMAL` is reported instead of:
   - `LU62_ALLOCATION_ERROR`
   - `LU62_DEALLOCATED_ABEND_PROG`
   - `LU62_DEALLOCATED_ABEND_SVC`
   - `LU62_DEALLOCATED_ABEND_TIMER`

Similarly, a `return_code` of `LU62_OK` is reported instead of:

- `LU62_PROG_ERROR_NO_TRUNC`
- `LU62_PROG_ERROR_PURGING`
- `LU62_PROG_ERROR_TRUNC`

The following types of incoming information are also purged:

- Data sent with the `lu62_send_data` call
- Confirmation request sent with the `lu62_confirm`, `lu62_prep_to_receive`, or `lu62_deallocate` verbs

If the confirmation request was sent with type set to `DA_CONFIRM` or `DA_SYNC_LEVEL`, the deallocation request is also purged.

The request-to-send notification is not purged. This notification is reported to the program when it issues a call that includes the `request_to_send_received` parameter.

5. When `request_to_send_received` is TRUE, the remote program is requesting that the local program "give up the turn", that is, that it enter Receive state, thereby placing the remote program in Send state. The local program enters Receive state by issuing `lu62_prep_to_receive` or `lu62_receive_and_wait`. The remote program issues `lu62_receive_immediate` or `lu62_receive_and_wait` to receive the resulting Send indication (`what_received` = `WR_SEND`).
6. The program can use this call for various application-level functions. For example, the program can issue this call to truncate an incomplete logical record it is sending; to inform the remote program of an error detected in data received, or to reject a confirmation request.
7. `lu62_send_error` resets or cancels posting. If posting is active and the conversation is posted, posting is reset. If posting is active and the conversation is not posted, posting is canceled.

See Also

`lu62_confirm`, `lu62_receive_and_wait`, `lu62_receive_immediate`

8.13 `lu62_test`

`lu62_test` tests the specified conversation to see if it was posted or a request-to-send notification was received.

Synopsis

```
int lu62_test(lu62_test_t *rqp);
```

Request Structure

```
typedef struct {
    bit32          conv_id;                           /* s */
    lu62_test_type_etest;                             /* s */
    bit32        return_code;                         /* r */
} lu62_test_t;
```

8.13.1 `lu62_test_t` *Request Structure Members*

The following subsections describe the `lu62_test_t` request structure members:

`conv_id`

(supplied) Specifies the id of the conversation to use. `conv_id` is returned by `lu62_allocate` or `lu62_accept`.

`test`

(supplied) Specifies the condition to be tested, as either:

- `TEST_POSTED`
- `TEST_REQUEST_TO_SEND_RECEIVED`

`return_code`

(returned) Specifies the result of verb execution.

If *test* = `TEST_POSTED`, `return_code` can be one of the following:

- `LU62_OK_DATA`, data is available to be received
- `LU62_OK_NO_DATA`, information other than data (that is, status or a confirmation request) is available to be received
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`

- LU62_CONV_ID_UNKNOWN
  - LU62_BAD_TEST_TYPE
- LU62_PROGRAM_STATE_CHECK
  — The conversation is not in Receive state.
  — LU62_VERB_IN_PROGRESS
- LU62_POSTING_NOT_ACTIVE
- LU62_UNSUCCESSFUL (no information is available)
- LU62_ALLOCATION_ERROR
- LU62_DEALLOCATE_ABEND_PROG
- LU62_DEALLOCATE_ABEND_TIMER
- LU62_DEALLOCATE_ABEND_SVC
- LU62_DEALLOCATE_NORMAL
- LU62_PROG_ERROR_NO_TRUNC
- LU62_PROG_ERROR_PURGING
- LU62_PROG_ERROR_TRUNC
- LU62_SVC_ERROR_NO_TRUNC
- LU62_SVC_ERROR_PURGING
- LU62_SVC_ERROR_TRUNC
- LU62_RESOURCE_FAILURE_NO_RETRY
- LU62_RESOURCE_FAILURE_RETRY

If *test* = TEST_REQUEST_TO_SEND_RECEIVED, return_code can be:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  — LU62_CONV_ID_UNKNOWN
  — LU62_BAD_TEST_TYPE
- LU62_PROGRAM_STATE_CHECK
  — The conversation is not is Send or Receive state.
  — LU62_VERB_IN_PROGRESS
  — LU62_UNSUCCESSFUL (request-to-send not received)

State Changes

No state change occurs.

Usage Notes

1. See the Usage Notes for `lu62_post_on_receipt` for a discussion of using `lu62_test` (`TEST_POSTED`) in conjunction with `lu62_post_on_receipt` and `lu62_wait` to provide non-blocking receive processing.

2. If the `return_code` to `lu62_test` (POSTED) indicates that information is available to be received (`LU62_DATA` or `LU62_NO_DATA`), the local program issues `lu62_receive_and_wait` or `lu62_receive_immediate` to receive the information.

3. If the `return_code` to `lu62_test` (`TEST_POSTED`) is `LU62_UNSUCCESSFUL`, posting remains active for the conversation.

4. If the `return_code` to `lu62_test` (`TEST_REQUEST_TO_SEND_RECEIVED`) is `LU62_OK`, a request-to-send notification is received by the local LU. `LU62_UNSUCCESSFUL` indicates that a request-to-send notification was not received.

See Also

`lu62_post_on_receipt`, `lu62_receive_and_wait`,
`lu62_receive_immediate`, `lu62_test`

Mapped Conversation Verbs 9

This chapter describes the SunLink P2P LU6.2 9.1 mapped conversation verbs See Table 9-1 for a list of verbs, along with their function. Detailed man pages follow.

Table 9-1 SunLink LU6.2 Mapped Conversation Verbs

Verb	Function
`lu62_mc_allocate`	Initiates a conversation with a remote TP
`lu62_mc_confirm`	Issues a confirmation request to the remote TP
`lu62_mc_confirmed`	Issue a confirmation responses
`lu62_mc_deallocate`	Terminates a conversation
`lu62_mc_flush`	Forces transmission of data in the send buffer
`lu62_mc_get_attributes`	Returns information about a conversation
`lu62_mc_post_on_receipt`	Sets receives posting conditions for a conversation
`lu62_mc_prep_to_receive`	Changes the conversation from Send to Receive state
`lu62_mc_receive_and_wait`	Waits for information to arrive and then receives it
`lu62_mc_receive_immediate`	Receives available information but does not wait
`lu62_mc_request_to_send`	Requests the turn

Table 9-1 SunLink LU6.2 Mapped Conversation Verbs *(Continued)*

Verb	**Function**
`lu62_mc_send_data`	Sends data on the conversation
`lu62_mc_send_error`	Notifies the remote program of a detected error
`lu62_mc_test`	Tests a conversation for posting

9.1 `lu62_mc_allocate`

`lu62_mc_allocate` is used to initiate a mapped conversation with a remote (partner) transaction program. A session is assigned for the exclusive use of the local and remote programs for the duration of the conversation. A conversation id is assigned to the mapped conversation. This conversation id is used to identify the conversation in all subsequent verbs that are issued.

Synopsis

```
int lu62_mc_allocate(lu62_allocate_t *rqp);
```

Request Structure

Code Example 9-1

```
typedef struct {
bit32                   port_id;                            /*s*/
bit32                   tp_id;                              /*s*/
char                    unique_session_name                 /*s*/
                          [LU62_UNIQUE_SESSION_NAME_LEN+1]; /*s*/
char                    lu_name[LU62_LU_NAME_LEN+1];        /*s*/
char                    mode_name[LU62_MODE_NAME_LEN+1];    /*s*/
char                    remote_tp_name[LU62_TP_NAME_LEN+1]; /*s*/
bit32                   conv_grp_id;                        /*s*/
lu62_processing_mode_e processing_mode;                     /*s*/
lu62_conv_type_e        type;                               /*s*/
lu62_flush_e            flush;                              /*s*/
lu62_return_control_e   return_control;                     /*s*/
lu62_sync_level_e       sync_level;                         /*s*/
lu62_pip_presence_e     pip_presence;                       /*s*/
lu62_security_e         security;                           /*s*/
char                    user_id[LU62_MAX_USER_ID_LEN+1];    /*s*/
char                    passwd[LU62_MAX_PASSWD_LEN+1];      /*s*/
char                    profile[LU62_MAX_PROFILE_LEN+1];    /*s*/
bit32                   conv_id;                            /*r*/
int                     luw_len                             /*r*/
bit8                    luw[LU62_MAX_LUW_LEN];              /*r*/
bit32                   return_code;                        /*r*/
} lu62_allocate_t;
```

9.1.1 lu62_allocate_t *Request Structure Members*

The lu62_allocate_t request structure members are described in the following subsections:

port_id

(supplied) Specifies the port_id of the LU connection to use. The port_id is returned by lu62_open.

tp_id

(supplied / optional) Specifies the id of a registered TP for which a conversation is accepted. This parameter is used when *security* = SECURITY_SAME (see below). The tp_id is returned by lu62_accept.

unique_session_name

Used to specify the actual node used from the configuration instead of lu_name and node. Furthermore, to use unique session names, the TP cannot use an lu_name in the previous open.

lu_name

(supplied) Specifies the locally known name of the partner LU at which the remote transaction program, remote_tp_name, is located. lu_name is supplied as an ASCII (null-terminated) string. It corresponds to the PTNR_LU NAME or NO_LU_NAME parameter in the configuration.

mode_name

(supplied) Specifies the name of the required mode. The conversation is allocated on a session of this mode. mode_name is supplied as an ASCII (null-terminated) string and is translated to EBCDIC by the SunLink SNA PU2.1 9.1 server. It corresponds to the MODE NAME parameter in the configuration.

remote_tp_name

(supplied) Specifies the name of the transaction program to which conversation attachment is required. `remote_tp_name` is supplied as an ASCII (null-terminated) string and is translated to EBCDIC by the SunLink SNA PU2.1 9.1 server.

conv_grp_id

(ignored) Reserved for future use.

processing_mode

(supplied) Specifies the initial processing mode of the conversation from one of the following:

- `PM_BLOCKING`
- `PM_NON_BLOCKING`

If `processing_mode` is set to `PM_BLOCKING`, `lu62_mc_allocate` does not return until a conversation is successfully allocated (`return_code` = `LU62_OK`) or the verb fails for some reason. If `processing_mode` is set to `PM_NON_BLOCKING` and initial parameter checks pass, `return_code` is set to `LU62_OPERATION_INCOMPLETE` and `lu62_wait_server` must be issued to receive the eventual return.

The specified `processing_mode` remains in effect for the allocated conversation until `lu62_set_processing_mode` is issued or the conversation terminates.

type

(ignored) Basic conversations only.

flush

(supplied) Specifies whether the allocation request is sent to the remote LU as soon as a session is allocated for the conversation, or whether the allocation request is retained until some other flush condition arises. `flush` is set to one of the following:

- `FLUSH_NO`
- `FLUSH_YES`

return_control

(supplied) Specifies when, in relation to the allocation of a session for the conversation, the local LU is to return control to the program. `return_control` is set to one of the following:

- `RC_WHEN_SESSION_ALLOCATED`
- `RC_IMMEDIATE`

Note – `LU62_OPERATION_INCOMPLETE` will still be returned when the processing mode is non-blocking.

- `RC_WHEN_CONWINNER_ALLOCATED`

sync_level

(supplied) Specifies the synchronization level for the conversation from one of the following:

- `SYNC_LEVEL_NONE`
- `SYNC_LEVEL_CONFIRM`
- `SYNC_LEVEL_SYNCPT`

The selected `sync_level` must correspond to the `TP SYNC_LEVEL` configuration of the remote TP.

pip_presence

(supplied) Specifies whether the `pip_presence` field of the FMH-5 Attach request is to be set or not:

- `PIP_NOT_PRESENT`
- `PIP_PRESENT`

If `PIP_PRESENT`, the FMH-5 attach request is built with the `pip_presence` indicator set. The caller is obliged to build the PIP variable (see Chapter 11 of the *IBM SNA Formats* manual and send it using the `lu62_send_data` verb.

security

(supplied) Specifies the access security information that the partner LU requires to verify the identity of the conversation initiator and validate access to the remote program and its resources. `security` may be:

- SECURITY_NONE—Security access information is omitted
- SECURITY_SAME—Specifies to use known (and already verified) user_id (and profile) information. If user_id (and profile) are supplied, these values are used. Otherwise, if tp_id is supplied, information is taken from the first allocation request accepted for that TP, or if tp_id is not supplied, the Unix user_id is sent.
- SECURITY_PROGRAM—Indicates that the required user_id, password (and profile) are included in this allocate request.

user_id

(supplied / conditional) If *security* is SECURITY_NONE or SECURITY_PROGRAM, a user_id is required. user_id is specified as an ASCII (null-terminated) string and is translated to EBCDIC by the SunLink SNA PU2.1 9.1 server.

passwd

(supplied / conditional) If *security* is SECURITY_PROGRAM, passwd is required. passwd is specified as an ASCII (null-terminated) string and is translated to EBCDIC by the SunLink SNA PU2.1 9.1 server.

profile

(supplied / conditional) profile is specified as an ASCII (null-terminated) string and is translated to EBCDIC by the SunLink SNA PU2.1 9.1 server.

conv_id

(returned) Specifies the identifier of the allocated conversation. All subsequent verbs issued on this conversation require this identifier.

luw_len
luw

(supplied / returned) This extension to the *IBM SNA Transaction Programmer's Reference Manual* is provided for SYNC_LEVEL_SYNCPT (see Appendix F). If supplied, luw contains the complete Logical Unit of Work Identifier to be set in the FMH-5 Attach (see Chapter 11 of the *IBM SNA Formats* manual). If luw is not supplied for SYNC_LEVEL_SYNCPT, the SunLink SNA PU2.1 9.1 server will generate the luw on the caller's behalf.

return_code

(returned) Specifies the result of verb execution. return_code values are affected by the value of the return_control parameter. The following return_codes can occur with all values of return_control:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - LU62_PORT_ID_UNKNOWN
- LU62_BAD_LU_NAME
- LU62_BAD_MODE_NAME
- LU62_BAD_REMOTE_TP_NAME
- LU62_BAD_PROCESSING_MODE
- LU62_BAD_CONV_TYPE
- LU62_BAD_FLUSH_TYPE
- LU62_BAD_RETURN_CONTROL
- LU62_BAD_SYNC_LEVEL
- LU62_BAD_SECURITY
- LU62_BAD_SECURITY_PROGRAM
- LU62_BAD_USERID
- LU62_BAD_PASSWD
- LU62_BAD_PROFILE
- LU62_LU_NAME_REQD
- LU62_MODE_NAME_REQD
- LU62_REMOTE_TP_NAME_REQD
- LU62_UNKNOWN_TP
- LU62_PROGRAM_STATE_CHECK
  - LU62_TP_NOT_STARTED
- LU62_ALLOCATION_ERROR
  - LU62_SYNC_LEVEL_NOT_SUPPORTED_BY_LU

If return_control is set to RC_WHEN_SESSION_ALLOCATED or RC_WHEN_CONWINNER_ALLOCATED, the following additional return_codes are possible:

- LU62_PARAMETER_ERROR
  - LU62_UNKNOWN_MODE
  - LU62_UNKNOWN_PARTNER_LU

- `LU62_ALLOCATION_ERROR`
  - `LU62_ALLOCATION_FAILURE_NO_RETRY`
  - `LU62_ALLOCATION_FAILURE_RETRY`

If `return_control` is set to `RC_IMMEDIATE`, the following additional `return_codes` are possible:

- `LU62_PARAMETER_ERROR`
  - `LU62_UNKNOWN_MODE`
  - `LU62_UNKNOWN_PARTNER_LU`
- `LU62_UNSUCCESSFUL` (no session is available).

State Changes

When `return_code` is `LU62_OK`, the conversation enters Send state.

Usage Notes

1. Session contention occurs when both LUs attempt to allocate a conversation on the session at the same time. Contention is resolved by making one LU the contention-winner, and the other the contention-loser. The contention-winner is guaranteed access to the session; the contention-loser must first ask permission of the contention-winner LU before it attempts to allocate a conversation on the session. See the MODE directive in the configuration for more information on session limits, conwinnners and conlosers.

2. An allocation error resulting from the local LU's failure to obtain a session for the conversation is reported on the `lu62_mc_allocate` call. An allocation error resulting from the remote LU's rejection of the allocation request is reported on a subsequent conversation call.

See Also

Section 5.3, “Allocating Conversations,” which provides an example of the use of this verb.

9.2 `lu62_mc_confirm`

`lu62_mc_confirm` sends a request for confirmation to the remote transaction program and waits for a reply. In normal circumstances, the remote program issues an `lu62_mc_confirmed` verb in response. The LU flushes the conversation's send buffer as a function of this verb.

Synopsis

```
int lu62_mc_confirm(lu62_confirm_t *rqp);
```

Request Structure

```
typedef struct {
    bit32       conv_id;                             /* s */
    bit32       return_code;                         /* r */
    bit32       request_to_send_received;            /* r */
} lu62_confirm_t;
```

9.2.1 `lu62_confirm_t` *Request Structure Members*

The following subsections describe the `lu62_confirm_t` request structure members:

`conv_id`

(supplied) Specifies the id of the conversation to use. `conv_id` is returned by `lu62_mc_allocate` or `lu62_accept`.

`return_code`

(returned) Specifies the result of verb execution from one of the following:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  - `LU62_CONV_ID_UNKNOWN`
  - `LU62_BAD_SYNC_LEVEL`
- `LU62_PROGRAM_STATE_CHECK`
  - the conversation is not is Send state
  - `LU62_PIP_PENDING` - the conversation is in Send state following an `lu62_(mc_)_allocate` verb in which `pip_presence` was indicated. A basic send, `lu62_send_data`, is required to send the PIP Variable.
  - `LU62_VERB_IN_PROGRESS`
- `LU62_ALLOCATION_ERROR`

- `LU62_DEALLOCATE_ABEND`
- `LU62_FMH_DATA_NOT_SUPPORTED`
- `LU62_PROG_ERROR_PURGING`
- `LU62_RESOURCE_FAILURE_NO_RETRY`
- `LU62_RESOURCE_FAILURE_RETRY`

`request_to_send_received`

(returned) Indicates whether or not a request-to-send indication is received from the remote program:

- `TRUE`, indicates that a request-to-send indication was received.
- `FALSE`, indicates that a request-to-send indication was not received.

State Changes

This verb can only be issued in Send state. No state change occurs.

Usage Notes

1. This verb is used to synchronize local and remote processing:
   - The initiating program may issue this verb immediately following `lu62_mc_allocate` to ensure that the remote program is available and attached before sending any data.
   - The sending program may issue this verb as a request for acknowledgment of the data it sent to the remote program. The remote program issues `lu62_mc_confirmed` to positively acknowledge receipt, or `lu62_mc_send_error` to indicate that it encountered an error.
2. When `request_to_send_received` is `TRUE`, the remote program is requesting that the local program to "give up the turn" and enter Receive state, thereby placing the remote program in Send state. The local program enters Receive state by issuing `lu62_mc_prep_to_receive` or `lu62_mc_receive_and_wait`. The remote program issues `lu62_mc_receive_immediate` or `lu62_mc_receive_and_wait` to receive the resulting send indication (`what_received` = `WR_SEND`).

See Also

`lu62_mc_confirmed`, `lu62_mc_send_error`

Section 5.8, “Basic Conversations,” illustrates how programs may be synchronized using confirmation requests.

9.3 lu62_mc_confirmed

lu62_mc_confirmed sends a confirmation reply in response to a confirmation request from the remote transaction program. The local program issues this verb when it receives a confirmation request. (See the what_received parameter of the lu62_mc_receive_and_wait and lu62_mc_receive_immediate verbs). This verb can only be issued as a reply to a confirmation request and cannot be issued at any other time.

Synopsis

```
int lu62_mc_confirmed(lu62_confirmed_t *rqp);
```

Request Structure

```
typedef struct {
    bit32          conv_id;                        /* s */
    bit32          return_code;                    /* r */
} lu62_confirmed_t;
```

9.3.1 lu62_confirmed_t *Request Structure Members*

The following subsections describe the lu62_confirmed_t request structure members:

conv_id

(supplied) Specifies the id of the conversation to use. conv_id is returned by lu62_mc_allocate or lu62_accept.

return_code

(returned) Specifies the result of verb execution from one of the following:

- LU62_OK
- LU62_OPERATION_INCOMPLETE

- LU62_PARAMETER_CHECK
  - — LU62_CONV_ID_UNKNOWN
- LU62_PROGRAM_STATE_CHECK
  - — Conversation is not in Confirm, Confirm_Send, or Confirm_Deallocate state
  - — LU62_VERB_IN_PROGRESS

State Changes

The state change depends on the value of the what_received parameter of the preceding lu62_mc_receive_and_wait or lu62_mc_receive_immediate verb:

- Receive state is entered when what_received = WR_CONFIRM.
- Send state is entered when what_received = WR_CONFIRM_SEND.
- Deallocate state is entered when what_received = WR_CONFIRM_DEALLOCATE.

Usage Notes

1. The local and remote programs use the lu62_mc_confirm and lu62_mc_confirmed verbs to synchronize their processing. For example, the remote program can request an acknowledgment that the data it sent was received by the local program. The local program issues lu62_mc_confirmed to provide a positive acknowledgment or lu62_mc_send_error, to send a negative acknowledgment.

See Also

lu62_mc_receive_and_wait, lu62_mc_receive_immediate, lu62_mc_confirm, lu62_mc_send_error.

Section 5.8, “Basic Conversations,” illustrates how programs may be synchronized using confirmation requests.

9.4 lu62_mc_deallocate

lu62_mc_deallocate deallocates the specified conversation from the transaction program. The deallocation can include the function of the lu62_mc_flush or lu62_mc_confirm verb, depending on the value of the type parameter.

Synopsis

```
int lu62_mc_deallocate(lu62_deallocate_t *rqp);
```

Request Structure

```
typedef struct {
    bit32          conv_id;                          /* s */
    lu62_deallocate_type_e type;                     /* s */
    char           *log_data;                        /* s */
    bit32          return_code;                      /* r */
} lu62_deallocate_t;
```

The lu62_deallocate_t request structure members are:

conv_id

(supplied) Specifies the id of the conversation to use. conv_id is returned by lu62_mc_allocate or lu62_accept.

type

(supplied) Specifies the type of deallocation to perform one of the following:

- DA_SYNC_LEVEL
  Deallocation processing is dependent on the conversation synchronization level (as specified by the sync_level parameter of the lu62_mc_allocate verb):
  - — SYNC_LEVEL_NONE: Performs the function of the lu62_mc_flush verb and deallocates the conversation normally

- — SYNC_LEVEL_CONFIRM: Performs the function of the lu62_mc_confirm verb and, if successful, deallocates the conversation normally
- — DA_SYNC_LEVEL is not supported when the sync_level is SYNC_LEVEL_SYNCPT

- DA_FLUSH
  Performs the function of the lu62_mc_flush verb and deallocates the conversation normally
- DA_CONFIRM
  Performs the function of the lu62_mc_confirm verb and, if successful, deallocates the conversation normally. This deallocation type can only be used on conversations with sync_level =SYNC_LEVEL_CONFIRM.
- DA_ABEND
  Performs the function of the lu62_mc_flush verb and deallocates the conversation abnormally. Data purging can occur in Receive state
- DA_UNBIND
  Forces the session to be deactivated by the SunLink SNA PU2.1 9.1 server. This extension to the *IBM SNA Transaction Programmer's Reference Manual* is provided for SYNC_LEVEL_SYNCPT (see Appendix F)
- DA_LOCAL
  Deallocates the conversation locally. This type of deallocation can only be specified, and must be specified, if the conversation is in Deallocate state

log_data

(ignored) Basic conversations only.

return_code

(returned) Specifies the result of verb execution. return_code is dependent on the deallocation type.

When the function of the lu62_mc_flush verb is performed (see above), return_code can be:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - LU62_CONV_ID_UNKNOWN
  - LU62_BAD_DEALLOCATE_TYPE
- LU62_PROGRAM_STATE_CHECK
  - the conversation is not in Send state
  - LU62_VERB_IN_PROGRESS

When the function of the lu62_mc_confirm verb is performed (see above), return_code can be:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - LU62_CONV_ID_UNKNOWN
  - LU62_BAD_SYNC_LEVEL
- LU62_PROGRAM_STATE_CHECK
  - the conversation is not in Send state
  - LU62_VERB_IN_PROGRESS
- LU62_ALLOCATION_ERROR
- LU62_DEALLOCATE_ABEND
- LU62_FMH_DATA_NOT_SUPPORTED
- LU62_PROG_ERROR_PURGING
- LU62_RESOURCE_FAILURE_NO_RETRY
- LU62_RESOURCE_FAILURE_RETRY

When deallocation type is DA_ABEND or DA_UNBIND, return_code can be:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - LU62_CONV_ID_UNKNOWN
- LU62_PROGRAM_STATE_CHECK
  - The conversation is not in Send, Receive, or Confirm state.

  — LU62_VERB_IN_PROGRESS

When deallocation type is DA_LOCAL, return_code can be:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - LU62_CONV_ID_UNKNOWN
- LU62_PROGRAM_STATE_CHECK
  - The conversation is not in Deallocate state.
  - LU62_VERB_IN_PROGRESS

State Changes

When return_code is LU62_OK, the conversation is Reset.

Usage Notes

1. The deallocation type DA_SYNC_LEVEL causes the conversation deallocation to be performed based on the conversation's synchronization level.
2. If the deallocation type is DA_LOCAL, or DA_SYNC_LEVEL and the sync_level is SYNC_LEVEL_NONE, the conversation is unconditionally deallocated. The remote program lu62_receive* return_code is LU62_DEALLOCATE_NORMAL, which causes it to enter Deallocate state. In Deallocate state, the remote program issues lu62_mc_deallocate(DA_LOCAL) to end the conversation.
3. If the deallocation type is DA_CONFIRM, or DA_SYNC_LEVEL and the sync_level is SYNC_LEVEL_CONFIRM, the function of the lu62_mc_confirm verb is performed prior to deallocation. The remote program receives what_received = WR_CONFIRM_DEALLOCATE, and may issue an lu62_mc_confirmed verb in response. In this case the conversation is deallocated when the local LU receives the confirmation response. If the remote program issues lu62_mc_send_error, the conversation remains allocated.
4. The deallocation type DA_ABEND is intended to be used to unconditionally deallocate the conversation irrespective of its synchronization level or state. If the conversation is operating in non-blocking mode, and an operation is

incomplete, an attempt to `lu62_mc_deallocate(DA_ABEND)` will cause an `LU62_PROGRAM_STATE_ERROR`. In this situation use the `lu62_abort` verb to abandon the conversation.

See Also

`lu62_mc_receive_and_wait`, `lu62_mc_receive_immediate`, `lu62_abort`

9.5 lu62_mc_flush

`lu62_mc_flush` flushes the local LU's conversation send buffer. Any buffered information is sent to the remote LU. Information buffered by the LU can come from `lu62_mc_allocate` (`flush` = `FLUSH_NO`), `lu62_mc_send_data`, or `lu62_mc_send_error`.

Synopsis

```
int lu62_mc_flush(lu62_flush_t *rqp);
```

Request Structure

```
typedef struct {
    bit32          conv_id;                           /* s */
    bit32          return_code;                       /* r */
} lu62_flush_t;
```

9.5.1 lu62_flush_t *Request Structure Members*

The following subsections describe the `lu62_flush_t` request structure members:

conv_id

(supplied) Specifies the id of the conversation to use. `conv_id` is returned by `lu62_mc_allocate` or `lu62_accept`.

`return_code`

(returned) Specifies the result of verb execution as one of the following:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  - `LU62_CONV_ID_UNKNOWN`
  - `LU62_BAD_FLUSH_TYPE`
- `LU62_PROGRAM_STATE_CHECK`
  - The conversation is not is Send state.
  - `LU62_VERB_IN_PROGRESS`

State Changes

This verb can only be issued in Send state. No state change occurs.

Usage Notes

1. Normally, the LU buffers the data from consecutive `lu62_mc_send_data` verbs until it has completely filled the current request unit (RU), or the local program issues a verb that causes an end-of-chain to be sent to the remote LU. Only then it sends the data to the remote LU. This way the transmission overhead is minimized. The `lu62_mc_flush` verb enables the local program to force buffer transmission.

2. The LU flushes its buffer only if it has something to send. Nothing is sent if the buffer is empty.

See Also

`lu62_mc_allocate`, `lu62_mc_send_data`, `lu62_mc_send_error`

9.6 `lu62_mc_get_attributes`

`lu62_mc_get_attributes` is used to provide information regarding the specified conversation.

Synopsis

```
int lu62_mc_get_attributes(lu62_get_attributes_t *rqp);
```

Request Structure

```
typedef struct {
    bit32          conv_id;                                 /* s */
    bit32          return_code;                             /* r */
    char            unique_session_name                         /* r */
                      [LU62_UNIQUE_SESSION_NAME_LEN+1];
    char           partner_lu_name[LU62_LU_NAME_LEN+1];     /* r */
    char           mode_name[LU62_MODE_NAME_LEN+1];         /* r */
    bit8           partner_qlu_name[LU62_NQ_LU_NAME_LEN+1]; /* r */
    int            partner_qlu_name_len;                    /* r */
    lu62_sync_level_e   sync_level;                         /* r */
    lu62_conv_state_e   conv_state;                         /* r */
    int            conv_corr_len;                           /* r */
    bit8           conv_corr[LU62_MAX_CONV_CORR_LEN];       /* r */
    bit32          conv_grp_id;                             /* r */
    int            sess_id_len;                             /* r */
    bit8           sess_id[LU62_MAX_SESS_ID_LEN];           /* r */
    int            luw_len;                                 /* r */
    bit8           luw[LU62_MAX_LUW_LEN];                   /* r */
} lu62_get_attributes_t;
```

The `lu62_get_attributes_t` request structure members are:

conv_id

(supplied) Specifies the id of the conversation to use. `conv_id` is returned by `lu62_mc_allocate` or `lu62_accept`.

return_code

(returned) Specifies the result of verb execution, as one of the following:

- LU62_OK
- LU62_PARAMETER_CHECK

— LU62_CONV_ID_UNKNOWN

unique_session_name

Used to specify the actual node used from the configuration instead of lu_name and node. Furthermore, to use unique session names, the TP cannot have used an lu_name in the previous open.

partner_lu_name

(returned) Specifies the name of the partner LU at which the remote transaction program is located. partner_lu_name corresponds to the PTNR_LU NAME parameter in the configuration.

mode_name

(returned) Specifies the name of the selected mode. The conversation is allocated on a session of this mode. mode_name corresponds to the MODE NAME parameter in the configuration.

partner_qlu_name_len
partner_qlu_name

(returned) Specifies the fully qualified name of the partner LU at which the remote transaction program is located. partner_qlu_name corresponds to the PTNR_LU NQ_LU_NAME parameter in the configuration.

sync_level

(supplied) Specifies the synchronization level for the conversation, as one of the following:

- SYNC_LEVEL_NONE
- SYNC_LEVEL_CONFIRM
- SYNC_LEVEL_SYNCPT

conv_state

(supplied) Specifies the current state of the conversation, as one of the following:

- CONV_RESET
- CONV_SEND
- CONV_RECEIVE

- CONV_CONFIRM
- CONV_CONFIRM_SEND
- CONV_CONFIRM_DEALLOCATE
- CONV_DEALLOCATE

conv_grp_id

(ignored) Reserved for future use.

conv_corr_len
conv_corr

(ignored) Reserved for future use.

sess_id_len
sess_id

(returned) Returns the assigned session identifier. The sess_id is returned as binary data. (Contrast this to lu62_display_mode and lu62_deactivate_session in which session_id is an ASCII-hex string).

*luw_len
*luw

(returned) This extension to the *IBM SNA Transaction Programmer's Reference Manual* is provided for SYNC_LEVEL_SYNCPT (see Appendix F). luw contains the complete Logical Unit of Work Identifier as set in the FMH-5 Attach that initiated the conversation (see Chapter 11 of the *IBM SNA Formats* manual.

State Changes

No state change occurs.

See Also

Section 5.4, "Accepting Conversations," provides an example of how this verb is used.

9.7 lu62_mc_post_on_receipt

lu62_mc_post_on_receipt causes the LU to post the specified conversation when information is available to be received. The information can be data, conversation status, or a request for confirmation. When the conservation is posted, the information is retrieved using lu62_mc_receive_and_wait or lu62_mc_receive_immediate. Programs can issue the lu62_wait verb to wait for posting to occur. Alternatively, programs can issue lu62_mc_test to poll a conversation to see if it is posted.

Synopsis

```
int lu62_mc_post_on_receipt(lu62_post_on_receipt_t *rqp);
```

Request Structure

```
typedef struct {
    bit32          conv_id;                          /* s */
    int            length;                           /* s */
    lu62_fill_e    fill;                             /* s */
    bit32          return_code;                      /* r */
} lu62_post_on_receipt_t;
```

9.7.1 lu62_post_on_receipt_t *Request Structure Members*

The following subsections describe the lu62_post_on_receipt_t request structure members:

conv_id

(supplied) Specifies the id of the conversation to use. conv_id is returned by lu62_mc_allocate or lu62_accept.

length

(supplied) Specifies the maximum amount of data the program can receive.

`fill`

(ignored) Basic conversations only.

`return_code`

(returned) Specifies the result of verb execution from one of the following:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  - `LU62_CONV_ID_UNKNOWN`
  - `LU62_BAD_LENGTH`
- `LU62_PROGRAM_STATE_CHECK`
  - Conversation is not is Receive state.
  - `LU62_VERB_IN_PROGRESS`

State Changes

This verb can only be issued in Receive state. No state change occurs.

Usage Notes

1. The `lu62_mc_post_on_receipt`, `lu62_wait`, and `lu62_mc_test` verbs together provide the *IBM SNA Transaction Programmer's Reference Manual* architected solution for handling multiple conversations in a non-blocking manner. This solution, however, handles receive processing only. All conversations must still wait for one conversation to confirm an operation, or for another to deallocate. An alternative approach is to use `lu62_set_processing_mode` to set your conversations into `LU62_NON_BLOCKING` mode. In this mode, all verbs that require interaction with the SunLink SNA PU2.1 9.1 server return as `LU62_OPERATION_INCOMPLETE` as soon as a request is sent to the server. Other conversations can then be processed. When the program is ready, it issues `lu62_wait_server` to wait for an outstanding operation to complete. Thus, `lu62_mc_receive_and_wait` may be used to wait for conversation data or status without blocking other conversations.

2. Posting occurs when the LU has any information that would satisfy a receive verb. Refer to `lu62_mc_receive_and_wait` for a description of what information can be received (`what_received`).

3. Posting remains in effect until the conversation is posted, posting is reset, or posting is cancelled.

   Posting is reset when one of the following verbs is issued on the conversation after the conversation is posted:
   - `lu62_mc_deallocate (DA_ABEND)`
   - `lu62_mc_receive_and_wait`
   - `lu62_mc_receive_immediate`
   - `lu62_mc_send_error`
   - `lu62_mc_test`
   - `lu62_wait`

   Posting is cancelled when any of the following verbs is issued on the conversation before the conversation is posted:
   - `lu62_mc_deallocate (DA_ABEND)`
   - `lu62_mc_receive_and_wait`
   - `lu62_mc_receive_immediate`
   - `lu62_mc_send_error`

See Also

`lu62_mc_receive_and_wait`, `lu62_mc_receive_immediate`, `lu62_mc_test`, `lu62_wait`, `lu62_wait_server`

9.8 lu62_mc_prep_to_receive

`lu62_mc_prep_to_receive` changes the conversation from Send to Receive state, in preparation to receive data.

Synopsis

```
int lu62_mc_prep_to_receive(lu62_prep_to_receive_t *rqp);
```

Request Structure

```
typedef struct {
    bit32          conv_id;                            /* s */
    lu62_prep_to_rcv_type_e type;                      /* s */
    lu62_locks_e   locks;                              /* s */
    bit32          return_code;                        /* r */
} lu62_prep_to_receive_t;
```

9.8.1 `lu62_prep_to_receive_t` *Request Structure Members*

The following subsections describe the `lu62_prep_to_receive_t` request structure members.

`conv_id`

(supplied) Specifies the id of the conversation to use. `conv_id` is returned by `lu62_mc_allocate` or `lu62_accept`.

`type`

(supplied) Specifies the type of prepare to receive in order to perform one of the following:

- `PR_SYNC_LEVEL`
  Processing is dependent on the conversation synchronization level (as specified by the `sync_level` parameter of the `lu62_mc_allocate` verb):
  - `SYNC_LEVEL_NONE`, performs the function of the `lu62_mc_flush` verb and enters Receive state
  - `SYNC_LEVEL_CONFIRM`, performs the function of the `lu62_mc_confirm` verb and, if successful, enters Receive state
  - `PR_SYNC_LEVEL` is not supported when the `sync_level` is `SYNC_LEVEL_SYNCPT`
- `PR_FLUSH`
  Performs the function of the `lu62_mc_flush` verb and enters Receive state

- PR_CONFIRM
  Perform the function of the lu62_mc_confirm verb and, if successful, enters Receive state. This type can only be used on conversations with sync_level =SYNC_LEVEL_CONFIRM

locks

(supplied / conditional) Specifies when control is to be returned to the local program. This parameter is only relevant if type = PR_CONFIRM. It may be one of the following:

- LOCKS_SHORT
  Control is returned immediately after the confirmation response is received from the remote program
- LOCKS_LONG
  Control is returned when information, such as data or status, is received from the remote program following the confirmation response

return_code

(returned) Specifies the result of verb execution. return_code is dependent on the type.

The following return_codes can be returned for all values of the type parameter:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - LU62_CONV_ID_UNKNOWN
  - LU62_BAD_PREP_TO_RCV_TYPE
  - LU62_BAD_LOCKS_TYPE
- LU62_PROGRAM_STATE_CHECK
  - Conversation is not in Send state.
  - LU62_PIP_PENDING - the conversation is in Send state following an lu62_(mc_)_allocate verb in which pip_presence was indicated. A basic send, lu62_send_data, is required to send the PIP Variable
  - LU62_VERB_IN_PROGRESS

When the function of the `lu62_mc_confirm` verb is performed (see above), additional `return_codes` are possible:

- `LU62_OK`
- `LU62_ALLOCATION_ERROR`
- `LU62_DEALLOCATE_ABEND`
- `LU62_FMH_DATA_NOT_SUPPORTED`
- `LU62_PROG_ERROR_PURGING`
- `LU62_RESOURCE_FAILURE_NO_RETRY`
- `LU62_RESOURCE_FAILURE_RETRY`

State Changes

When `return_code` is `LU62_OK`, the conversation is in Receive state.

Usage Notes

1. When `type` = `PR_SYNC_LEVEL`, send control is transferred to the remote program based on the synchronization level of the conversation. Thus, if the synchronization level is `SYNC_LEVEL_CONFIRM`, a confirmation response is required before handing over send control.

2. When `type` = `PR_FLUSH`, or `type` =`PR_SYNC_LEVEL` and the synchronization level is `SYNC_LEVEL_NONE`, send control is transferred to the remote program without a confirmation. The remote program's `lu62_mc_receive*` verb returns with a `what_received` value of `WR_SEND`.

3. When `type` = `PR_CONFIRM`, or `type` = `PR_SYNC_LEVEL` and the synchronization level is `SYNC_LEVEL_CONFIRM`, a confirmation response is required before handing over send control. The remote program's `lu62_mc_receive*` verb returns with a `what_received` value of `WR_CONFIRM_SEND`.

See Also

`lu62_mc_confirmed`, `lu62_mc_receive_and_wait`, `lu62_mc_receive_immediate`

9.9 lu62_mc_receive_and_wait

lu62_mc_receive_and_wait waits for information to be received on the specified conversation. The information can be data, conversation status, or a request for confirmation. An indication of the type of information received is returned.

Synopsis

```
int lu62_mc_receive_and_wait(lu62_receive_t *rqp);
```

Request Structure

```
typedef struct {
    bit32          conv_id;                          /* s */
    lu62_fill_e    fill;                             /* s */
    int            length;                           /* s/r */
    bit32          return_code;                      /* r */
    bit32          request_to_send_received;         /* r */
    bit8           *data;                            /* r */
    lu62_what_received_e what_received;              /* r */
    char           map_name[LU62_MAP_NAME_LEN+1];    /* r */
} lu62_receive_t;
```

9.9.1 lu62_receive_t *Request Structure Members*

The following subsections describe the lu62_receive_t request structure members:

conv_id

(supplied) Specifies the id of the conversation to use. conv_id is returned by lu62_mc_allocate or lu62_accept.

fill

(ignored) Basic conversations only.

`length`

(supplied / returned) On input, this parameter specifies the maximum amount of data the program can receive. On return, and if data is received, the parameter is set with the amount of data received. If no data is received, this parameter is unchanged.

`return_code`

(returned) Specifies the result of verb execution, as one of the following:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
    - `LU62_CONV_ID_UNKNOWN`
    - `LU62_BAD_LENGTH`
    - `LU62_NULL_DATA`
- `LU62_PROGRAM_STATE_CHECK`
    - The conversation is not in *Send* or Receive state.
    - `LU62_PIP_PENDING` - the conversation is in Send state following an `lu62_(mc_)_allocate` verb in which `pip_presence` was indicated. A basic send, `lu62_send_data`, is required to send the PIP Variable.
    - `LU62_PIP_PENDING` - the conversation is in Receive state following an `lu62_accept` verb in which `pip_presence` was indicated. A basic receive, `lu62_receive_and_wait`, is required to receive the PIP variable.
    - `LU62_VERB_IN_PROGRESS`
- `LU62_ALLOCATION_ERROR`
- `LU62_DEALLOCATE_ABEND`
- `LU62_FMH_DATA_NOT_SUPPORTED`
- `LU62_DEALLOCATE_NORMAL`
- `LU62_PROG_ERROR_NO_TRUNC`
- `LU62_PROG_ERROR_PURGING`
- `LU62_RESOURCE_FAILURE_NO_RETRY`
- `LU62_RESOURCE_FAILURE_RETRY`

request_to_send_received

(returned) Indicates whether or not a request-to-send indication is received from the remote program:

- TRUE, indicates that a request-to-send indication was received.
- FALSE, indicates that a request-to-send indication was not received.

data

(Supplied / returned.) Specifies the buffer into which any received data is to be written. The buffer should be at least *length* bytes long. If `what_received` indicates that information other than data has been received, nothing is written into this buffer.

what_received

(returned) Indicates the type of information that is received as one of the following:

- `WR_DATA_COMPLETE`
  A complete data record is received (or the remaining portion thereof).
- `WR_DATA_TRUNCATED`
  An incomplete data record is received and the LU discards the remainder of the data record.
- `WR_DATA_INCOMPLETE`
  Less than a complete data record is received and the LU retained the remainder of the data record. The local program must issue at least one more `lu62_mc_receive_and_wait` verb to receive the remaining data.
- `*WR_PS_DATA_COMPLETE`
  A PS header is received on a `SYNC_LEVEL_SYNCPT` conversation (or the remaining portion thereof). The complete PS header is returned, including the (invalid) LL length bytes. This extension to *IBM SNA Transaction Programmer's Reference Manual* is provided for `SYNC_LEVEL_SYNCPT` (see Appendix F).
- *`WR_PS_DATA_INCOMPLETE`
  Less than a complete PS header is received on a `SYNC_LEVEL_SYNCPT` conversation. The local program must issue at least one more `lu62_receive_and_wait` verb to receive the remaining data. This extension to *IBM SNA Transaction Programmer's Reference Manual* is provided for `SYNC_LEVEL_SYNCPT` (see Appendix F).

- WR_FMH_DATA_COMPLETE
  A complete GDS user-control variable is received (or its remaining portion).
- WR_FMH_DATA_INCOMPLETE
  Less than a complete GDS user-control variable is received and the LU retained the remainder of the record. The local program must issue at least one more lu62_mc_receive_and_wait verb to receive the remaining data.
- WR_FMH_DATA_TRUNCATED
  An incomplete GDS variable is received and the LU discards the remainder of the record.
- WR_SEND
  The remote program has entered Receive state. The local program transitions to Send state.
- WR_CONFIRM
  The remote program issued lu62_mc_confirm. The local program may respond by issuing lu62_mc_confirmed.
- WR_CONFIRM_SEND
  The remote program requires a confirmation response before entering Receive state. The local program may respond by issuing lu62_mc_confirmed.
- WR_CONFIRM_DEALLOCATE
  The remote program requires a confirmation response before deallocating the conversation. The local program may terminate the conversation by issuing lu62_mc_confirmed.

map_name

(ignored) Reserved for future use.

State Changes

If the return_code is LU62_OK, the state changes according to the initial state and the value of the what_received parameter:

- Receive state is entered when the verb is issued in Send state, and when the following conditions are present:

  what_received = WR_DATA_COMPLETE, WR_DATA_INCOMPLETE, WR_PS_DATA_COMPLETE, WR_PS_DATA_INCOMPLETE, WR_FMH_DATA_COMPLETE, WR_FMH_DATA_INCOMPLETE

- Send state is entered when `what_received` = `WR_SEND`
- Confirm state is entered when `what_received` = `WR_CONFIRM`, `WR_CONFIRM_SEND`, or `WR_CONFIRM_DEALLOCATE`

No state change occurs when the verb is issued in Receive state and in the following conditions:

`what_received` = `WR_DATA_COMPLETE`, `WR_DATA_INCOMPLETE`, `WR_PS_DATA_COMPLETE`, `WR_PS_DATA_INCOMPLETE`, `WR_FMH_DATA_COMPLETE`, `WR_FMH_DATA_INCOMPLETE`, or `WR_FMH_DATA_TRUNCATED`.

Usage Notes

1. `lu62_mc_receive_and_wait` receives only one type of information at a time. It may receive data, status, or a confirmation request, as indicated by the value of `what_received`.
2. When `lu62_mc_receive_and_wait` is issued in Send state, an implicit `lu62_mc_prep_to_receive (PR_FLUSH)` is executed by the local LU.
3. When `what_received` = `WR_DATA_INCOMPLETE` or `WR_FMH_DATA_INCOMPLETE`, the length of the data record exceeds the maximum length of the user's data buffer. The local program must issue at least on more `lu62_mc_receive_and_wait` to receive the remainder of the data.
4. `lu62_mc_receive_and_wait` includes posting. If posting is already active, this verb supersedes the prior `lu62_mc_post_on_receipt`.
5. The request-to-send notification is usually received when the local program is in Send state and is reported to the program via the `request_to_send_received` parameter of the `lu62_mc_send_data` or `lu62_mc_send_error` verb. The notification can also be received, however, when the conversation is in Receive state. This can occur under three different circumstances:

- When the local program enters Receive state and the remote program issues `lu62_mc_request_to_send` before it enters Send state
- When the remote program enters Receive state using `lu62_mc_prep_to_receive` (not `lu62_mc_receive_and_wait`), and then issues `lu62_mc_request_to_send` before the local program enters

Send state. This can occur because the request-to-send is transmitted as an expedited request and can therefore arrive ahead of the request that carries the send indication. Potentially, the local program cannot distinguish this condition from the first. This ambiguity is avoided if the remote program waits until it receives information from the local program before it issues `lu62_mc_request_to_send`.

- When the remote program issues `lu62_mc_request_to_send` in Send state. This can be used to signal the local program that data is about to be sent. The local program issues `lu62_mc_test` (`TEST_REQUEST_TO_SEND_RECEIVED`) to poll the local LU for this situation. Only when the result is TRUE does the local program issue `lu62_mc_receive_and_wait`.

See Also

`lu62_mc_post_on_receipt`

Section 5.7, "Mapped Conversations," illustrates the use of this verb.

9.10 `lu62_mc_receive_immediate`

`lu62_mc_receive_immediate` requests any information that is available for the specified conversation. In contrast to `lu62_mc_receive_and_wait`, it does not wait for information to arrive. The information can be data, conversation status, or a request-for-confirmation; an indication of the type of information received is returned.

Synopsis

```
int lu62_mc_receive_immediate(lu62_receive_t *rqp);
```

Request Structure

```
typedef struct {
    bit32          conv_id;                          /* s */
    lu62_fill_e    fill;                             /* s */
    int            length;                           /* s/r */
    bit32          return_code;                      /* r */
    bit32          request_to_send_received;         /* r */
    bit8           *data;                            /* r */
    lu62_what_received_e what_received;              /* r */
    char           map_name[LU62_MAP_NAME_LEN+1];    /* r */
} lu62_receive_t;
```

9.10.1 `lu62_receive_t` *Request Structure Members*

The following subsections describe the `lu62_receive_t` request structure members:

`conv_id`

(supplied) Specifies the id of the conversation to use. `conv_id` is returned by `lu62_mc_allocate` or `lu62_accept`.

`fill`

(ignored) Basic conversations only.

`length`

(supplied/returned) On input, this parameter specifies the maximum amount of data the program can receive. On return, and if data has been received, the parameter is set with the amount of data received. If no data is received, this parameter is unchanged.

`return_code`

(returned) Specifies the result of verb execution as one of the following:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`

- — LU62_CONV_ID_UNKNOWN
- — LU62_BAD_LENGTH
- — LU62_NULL_DATA

- LU62_PROGRAM_STATE_CHECK
  - — The conversation is not in Receive state
  - — LU62_PIP_PENDING - the conversation is in Receive state following an lu62_accept verb in which pip_presence was indicated. A basic receive, lu62_receive_and_wait, is required to receive the PIP variable
  - — LU62_VERB_IN_PROGRESS
- LU62_ALLOCATION_ERROR
- LU62_DEALLOCATE_ABEND
- LU62_FMH_DATA_NOT_SUPPORTED
- LU62_DEALLOCATE_NORMAL
- LU62_PROG_ERROR_NO_TRUNC
- LU62_PROG_ERROR_PURGING
- LU62_RESOURCE_FAILURE_NO_RETRY
- LU62_RESOURCE_FAILURE_RETRY
- LU62_UNSUCCESSFUL (there is nothing to receive)

request_to_send_received

(returned) Indicates whether or not a request-to-send indication is received from the remote program:

- TRUE, indicates that a request-to-send indication was received.
- FALSE, indicates that a request-to-send indication was not received.

data

(supplied / returned) Specifies the buffer into which any received data is to be written. The buffer should be at least *length* bytes long. If what_received indicates that information other than data is received, nothing is written into this buffer.

what_received

(returned) Indicates the type of information that is received as one of the following:

- WR_DATA_COMPLETE
  A complete data record is received (or its remaining portion)
- WR_DATA_TRUNCATED
  An incomplete data record is received and the LU discarded the remainder of the data record
- WR_DATA_INCOMPLETE
  Less than a complete data record is received and the LU retained the remainder of the data record. The local program must issue at least one more lu62_mc_receive_and_wait verb to receive the remaining data
- *WR_PS_DATA_COMPLETE
  A PS header is received on a SYNC_LEVEL_SYNCPT conversation (or the remaining portion thereof). The complete PS header is returned, including the (invalid) LL length bytes. This extension to the *IBM SNA Transaction Programmer's Reference Manual* is provided for SYNC_LEVEL_SYNCPT (see Appendix F)
- *WR_PS_DATA_INCOMPLETE
  Less than a complete PS header is received on a SYNC_LEVEL_SYNCPT conversation. The local program must issue at least one more lu62_receive_immediate verb to receive the remaining data. This extension to the *IBM SNA Transaction Programmer's Reference Manual* is provided for SYNC_LEVEL_SYNCPT (see Appendix F)
- WR_FMH_DATA_COMPLETE
  A complete GDS user-control variable is received (or its remaining portion)
- WR_FMH_DATA_INCOMPLETE
  Less than a complete GDS user-control variable is received and the LU retained the remainder of the record. The local program must issue at least one more lu62_mc_receive_and_wait verb to receive the remaining data
- WR_FMH_DATA_TRUNCATED
  An incomplete GDS variable is received and the LU discarded the remainder of the record
- WR_SEND
  The remote program entered Receive state. The local program transitions to Send state
- WR_CONFIRM
  The remote program issued lu62_mc_confirm. The local program may respond by issuing lu62_mc_confirmed

- WR_CONFIRM_SEND
  The remote program requires a confirmation response before entering Receive state. The local program may respond by issuing lu62_mc_confirmed
- WR_CONFIRM_DEALLOCATE
  The remote program requires a confirmation response before deallocating the conversation. The local program may terminate the conversation by issuing lu62_mc_confirmed

map_name

(ignored) Mapped conversations only.

State Changes

If the return_code is LU62_OK, the state changes according to the initial state and the value of the what_received parameter:

- Receive state is entered when the verb is issued in Send state and what_received = WR_DATA_COMPLETE, WR_DATA_INCOMPLETE, WR_PS_DATA_COMPLETE, WR_PS_DATA_INCOMPLETE, WR_FMH_DATA_COMPLETE, WR_FMH_DATA_INCOMPLETE
- Send state is entered when what_received = WR_SEND
- Confirm state is entered when what_received = WR_CONFIRM, WR_CONFIRM_SEND, or WR_CONFIRM_DEALLOCATE

No state change occurs when what_received = WR_DATA_COMPLETE, WR_DATA_INCOMPLETE, WR_PS_DATA_COMPLETE, WR_PS_DATA_INCOMPLETE, WR_FMH_DATA_COMPLETE, WR_FMH_DATA_INCOMPLETE, or WR_FMH_DATA_TRUNCATED

Usage Notes

1. lu62_mc_receive_immediate receives only one type of information at a time. It may receive data, status, or a confirmation request, as indicated by the value of what_received.

2. When `what_received` = `WR_DATA_INCOMPLETE` or `WR_FMH_DATA_INCOMPLETE`, the length of the data record exceeds the maximum *length* of the user's data buffer. The local program must issue at least on more `lu62_mc_receive_immediate` to receive the remainder of the data.

3. `lu62_receive_immediate` resets or cancels posting. If posting is active and the conversation is posted, posting is reset. If posting is active and the conversation is not posted, posting is cancelled.

4. The request-to-send notification is usually received when the local program is in Send state, and is reported to the program via the `request_to_send_received` parameter of the `lu62_mc_send_data` or `lu62_mc_send_error` verb. The notification can also be received, however, when the conversation is in Receive state. This can occur under three different circumstances:
   - When the local program enters Receive state and the remote program issues `lu62_mc_request_to_send` before it enters Send state.
   - When the remote program enters Receive state using `lu62_mc_prep_to_receive` (not `lu62_mc_receive_and_wait`), and then issues `lu62_mc_request_to_send` before the local program enters Send state. This can occur because the request-to-send is transmitted as an expedited request and can therefore arrive ahead of the request carrying the send indication. Potentially, the local program cannot distinguish this condition from the first. This ambiguity is avoided if the remote program waits until it receives information from the local program before it issues `lu62_mc_request_to_send.`
   - When the remote program issues `lu62_mc_request_to_send` in Send state. This can be used to signal the local program that data is about to be sent. The local program issues `lu62_mc_test` (`TEST_REQUEST_TO_SEND_RECEIVED`) to poll the local LU for this situation. Only when the result is TRUE does the local program issue `lu62_mc_receive_immediate.`

See Also

`lu62_mc_post_on_receipt`

9.11 lu62_mc_request_to_send

lu62_mc_request_to_send sends a notification to the remote program to indicate that the local program wants to enter Send state. The conversation remains in its current state, however, until a send indication is received from the remote program.

Synopsis

```
int lu62_mc_request_to_send(lu62_request_to_send_t *rqp);
```

Request Structure

```
typedef struct {
    bit32           conv_id;                            /* s */
    bit32           return_code;                        /* r */
} lu62_request_to_send_t;
```

9.11.1 lu62_request_to_send_t *Request Structure Members*

The following subsections describe the lu62_request_to_send_t request structure members.

conv_id

(supplied) Specifies the id of the conversation to use. conv_id is returned by lu62_mc_allocate or lu62_accept.

return_code

(returned) Specifies the result of verb execution from one of the following:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  — LU62_CONV_ID_UNKNOWN

- `LU62_PROGRAM_STATE_CHECK`
  - — The conversation is not is Receive, Confirm, or Send state.
  - — `LU62_VERB_IN_PROGRESS`

State Changes

No state change occurs.

Usage Notes

1. The remote program is informed of the arrival of a request-to-send notification by means of the `request_to_send_received` parameter returned by `lu62_mc_confirm`, `lu62_mc_receive_and_wait`, `lu62_mc_receive_immediate`, `lu62_mc_send_data`, and `lu62_mc_send_error`. The remote program may also poll the LU to determine if a request-to-send notification was received using `lu62_mc_test` (`TEST_REQUEST_TO_SEND_RECEIVED`). When the remote program receives the request-to-send notification, it issues `lu62_mc_receive_and_wait` or `lu62_mc_prep_to_receive` to enter Receive state and thereby places the local program in Send state. The local program enters Send state when it issues an `lu62_mc_receive_and_wait` or `lu62_mc_receive_immediate` and receives the send indication.
2. The remote LU retains the request-to-send notification until the remote program issues one of the verbs identified above. Additional request-to-send notifications are discarded until the retained notification is passed to the remote program.

See Also

`lu62_mc_confirm`, `lu62_mc_prep_to_receive`, `lu62_mc_receive_and_wait`, `lu62_mc_receive_immediate`, `lu62_mc_send_data`, `lu62_mc_send_error`, `lu62_mc_test`

9.12 lu62_mc_send_data

lu62_mc_send_data is used to send one data record to the remote program. The program can specify whether the data record includes FM headers.

Synopsis

```
int lu62_mc_send_data(lu62_send_data_t *rqp);
```

Request Structure

```
typedef struct {
    bit32          conv_id;                         /* s */
    bit8           *data;                           /* s */
    int            length;                          /* s */
    char           map_name[LU62_MAP_NAME_LEN+1];   /* s */
    lu62_fmh_data_efmh_data;                        /* s */
    lu62_encrypt_e encrypt;                         /* s */
    lu62_flush_e   flush;                           /* s */
    bit32          return_code;                     /* r */
    bit32          request_to_send_received;        /* r */
} lu62_send_data_t;
```

9.12.1 u62_send_data_t *Request Structure Members*

The following subsections describe the lu62_send_data_t request structure members:

conv_id

(supplied) Specifies the id of the conversation to use. conv_id is returned by lu62_mc_allocate or lu62_accept.

data

(supplied) The address of the user buffer containing the data to be sent.

`length`

(supplied) Specifies the length of the data record.

`map_name`

(ignored) Reserved for future use.

`fmh_data`

(ignored) Specifies whether the data record contains FM headers from one of the following:

- `FMH_NO`
- `FMH_YES`

The following extensions to the *IBM SNA Transaction Programmer’s Reference Manual* are provided for sync-point recovery (see
,”Appendix F
,"

- `*FMH_ELN`
- `*FMH_CS`

`encrypt`

(ignored) Reserved for future use.

`flush`

(supplied) Specifies whether the supplied data is to be sent to the remote program immediately or buffered in the local LU's send buffer. `flush` is set to one of the following:

- `FLUSH_NO`
- `FLUSH_YES`

`return_code`

(returned) Specifies the result of verb execution from one of the following:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  — `LU62_CONV_ID_UNKNOWN`

  - LU62_NULL_DATA
  - LU62_BAD_LENGTH
  - LU62_BAD_FMH_DATA_TYPE
- LU62_PROGRAM_STATE_CHECK
  - The conversation is not is Send state
  - LU62_PIP_PENDING - the conversation is in Send state following an `lu62_(mc_)_allocate` verb in which `pip_presence` is indicated. A basic send, `lu62_send_data`, is required to send the PIP variable.
  - LU62_VERB_IN_PROGRESS
- LU62_ALLOCATION_ERROR
- LU62_DEALLOCATE_ABEND
- LU62_FMH_DATA_NOT_SUPPORTED
- LU62_PROG_ERROR_PURGING
- LU62_RESOURCE_FAILURE_NO_RETRY
- LU62_RESOURCE_FAILURE_RETRY

`request_to_send_received`

(returned) Indicates whether or not a request-to-send indication is received from the remote program:

- TRUE, indicates that a request-to-send indication was received
- FALSE, indicates that a request-to-send indication was not received

State Changes

No state change occurs.

Usage Notes

1. One data record at a time may be sent on mapped conversations. Unlike the logical records sent on basic conversations, data records contain only data.

2. Since one complete data record is sent, the sending program cannot truncate a data record.

3. The local LU buffers the data to be sent to the remote LU until it accumulates a sufficient amount of data for transmission (from one or more `lu62_mc_send_data` verbs), or until the local program issues a call that

causes the LU to flush its send buffer. The amount of data sufficient for transmission depends on the characteristics of the session allocated for the conversation, and varies from one session to another.

4. When `request_to_send_received` is TRUE, the remote program requests that the local program "give up the turn", i.e., that it enter Receive state, thereby placing the remote program in Send state. The local program enters Receive state by issuing `lu62_mc_prep_to_receive` or `lu62_mc_receive_and_wait`. The remote program issues `lu62_mc_receive_immediate` or `lu62_mc_receive_and_wait` to receive the resulting Send indication (`what_received` = `WR_SEND`).

See Also

`lu62_mc_send_error`

Section 5.7, “Mapped Conversations,” illustrates the use of this verb.

9.13 `lu62_mc_send_error`

`lu62_mc_send_error` is used by a program to inform the remote program that the local program detected an error during a conversation. If the conversation is in Send state, `lu62_mc_send_error` forces the LU to flush its send buffer.

When this call completes successfully, the local program is in Send state and the remote program is in Receive state. Further action is defined by program logic.

Synopsis

```
int lu62_mc_send_error(lu62_send_error_t *rqp);
```

Request Structure

```
typedef struct {
    bit32              conv_id;                     /* s */
    lu62_prog_type_e  type;                        /* s */
    char               *log_data;                   /* s */
    int                error_direction;             /* s */
    bit32               return_code;                /* r */
    bit32               request_to_send_received;   /* r */
} lu62_send_error_t;
```

9.13.1 lu62_send_error_t *Request Structure Members*

The lu62_send_error_t request structure members are:

conv_id

(supplied) Specifies the id of the conversation to use. conv_id is returned by lu62_mc_allocate or lu62_accept.

type

(ignored) Basic conversations only.

log_data

(ignored) Basic conversations only.

error_direction

(ignored) Reserved for use by CPI-C.

return_code

(returned) Specifies the result of verb execution. return_code may be one of the folowing:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - LU62_CONV_ID_UNKNOWN
- LU62_PROGRAM_STATE_CHECK
  - The conversation is not Send, Receive, or Confirm state
  - LU62_VERB_IN_PROGRESS

If the verb is issued in Send state, return_code can additionally be one of the following:

- `LU62_ALLOCATION_ERROR`
- `LU62_DEALLOCATE_ABEND`
- `LU62_PROG_ERROR_PURGING`
- `LU62_RESOURCE_FAILURE_NO_RETRY`
- `LU62_RESOURCE_FAILURE_RETRY`

If the verb is issued in Receive state, `return_code` can additionally be one of the following:

- `LU62_DEALLOCATE_NORMAL`
- `LU62_RESOURCE_FAILURE_NO_RETRY`
- `LU62_RESOURCE_FAILURE_RETRY`

`request_to_send_received`

(returned) Indicates whether or not a request-to-send indication is received from the remote program:

- TRUE, indicates that a request-to-send indication was received
- FALSE, indicates that a request-to-send indication was not received

State Changes

When `return_code` is `LU62_OK`:

- Send state is entered when the verb is issued in Receive or Confirm states
- No state change occurs when the verb is issued in Send state.

Usage Notes

1. The LU sends the error notification to the remote LU immediately (during the processing of this call).
2. The issuance of `lu62_mc_send_error` is reported to the remote program as one of the following return codes:
   - `LU62_PROG_ERROR_NO_TRUNC`

   The local program issued `lu62_mc_send_error` in Send state. No truncation can occur on mapped conversations.
   - `LU62_PROG_ERROR_PURGING`

The local program issued `lu62_mc_send_error` in Receive state. All information sent by the remote program and not yet received by the local program is purged; or the local program issued `lu62_mc_send_error` in Confirm state in which case no purging occurrs.

3. When `lu62_mc_send_error` is issued in Receive state, incoming information is also purged. Because of this purging, the `return_code` of `LU62_DEALLOCATE_NORMAL` is reported instead of:
   - `LU62_ALLOCATION_ERROR`
   - `LU62_DEALLOCATE_ABEND`

   Similarly, a `return_code` of `LU62_OK` is reported instead of:
   - `LU62_FMH_DATA_NOT_SUPPORTED`
   - `LU62_PROG_ERROR_NO_TRUNC`
   - `LU62_PROG_ERROR_PURGING`

   The following types of incoming information are also purged:
   - Data sent with the `lu62_mc_send_data` call
   - Confirmation request sent with the `lu62_mc_confirm`, `lu62_mc_prep_to_receive`, or `lu62_mc_deallocate` verbs

   If the confirmation request was sent with type set to `DA_CONFIRM` or `DA_SYNC_LEVEL`, the deallocation request is also purged.

   The request-to-send notification is not purged. This notification is reported to the program when it issues a call that includes the `request_to_send_received` parameter.

4. When `request_to_send_received` is TRUE, the remote program requests that the local program "give up the turn", i.e., that it enter Receive state, thereby placing the remote program in Send state. The local program enters Receive state by issuing `lu62_mc_prep_to_receive` or `lu62_mc_receive_and_wait`. The remote program issues `lu62_mc_receive_immediate` or `lu62_mc_receive_and_wait` to receive the resulting Send indication (`what_received` = `WR_SEND`).

5. The program can use this verb for various application-level functions. For example, the program may issue this verb to inform the remote program of an error detected in data records it received, or to reject a confirmation request.

6. lu62_mc_send_error resets or cancels posting. If posting is active and the conversation is posted, posting is reset. If posting is active and the conversation is not posted, the posting is cancelled.

See Also

lu62_mc_confirm, lu62_mc_receive_and_wait, lu62_mc_receive_immediate

9.14 lu62_mc_test

lu62_mc_test tests the specified conversation to see if it has been posted or if a request-to-send notification was received.

Synopsis

```
int lu62_mc_test(lu62_test_t *rqp);
```

Request Structure

```
typedef struct {
    bit32        conv_id;                                 /* s */
    lu62_test_type_e   test;                              /* s */
    bit32        return_code;                             /* r */
} lu62_test_t;
```

9.14.1 lu62_test_t *Request Structure Members*

The following subsections describe the lu62_test_t request structure members:

conv_id

(supplied) Specifies the id of the conversation to use. conv_id is returned by lu62_mc_allocate or lu62_accept.

`test`

(supplied) Specifies the condition to be tested, which is one of the following:

- `TEST_POSTED`
- `TEST_REQUEST_TO_SEND_RECEIVED`

`return_code`

(returned) Specifies the result of verb execution.

If `test` = `TEST_POSTED`, `return_code` can be one of the following:

- `LU62_OK_DATA`, a data record is available to be received
- `LU62_OK_NO_DATA`, information other than data (that is, status or a confirmation request) is available to be received
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  - `LU62_CONV_ID_UNKNOWN`
  - `LU62_BAD_TEST_TYPE`
- `LU62_PROGRAM_STATE_CHECK`
  - The conversation is not is Receive state.
  - `LU62_VERB_IN_PROGRESS`
- `LU62_POSTING_NOT_ACTIVE`
- `LU62_UNSUCCESSFUL (no information is available)`
- `LU62_ALLOCATION_ERROR`
- `LU62_DEALLOCATE_ABEND`
- `LU62_FMH_DATA_NOT_SUPPORTED`
- `LU62_DEALLOCATE_NORMAL`
- `LU62_PROG_ERROR_NO_TRUNC`
- `LU62_PROG_ERROR_PURGING`
- `LU62_PROG_ERROR_TRUNC`
- `LU62_RESOURCE_FAILURE_NO_RETRY`
- `LU62_RESOURCE_FAILURE_RETRY`

If `test` = `TEST_REQUEST_TO_SEND_RECEIVED`, `return_code` can be one of the following:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  - `LU62_CONV_ID_UNKNOWN`
  - `LU62_BAD_TEST_TYPE`
- `LU62_PROGRAM_STATE_CHECK`
  - The conversation is not is Send or Receive state.
  - `LU62_VERB_IN_PROGRESS`
- `LU62_UNSUCCESSFUL` (request-to-send not received)

State Changes

No state change occurs.

Usage Notes

1. See the usage notes for `lu62_mc_post_on_receipt` for a discussion of the use of `lu62_mc_test` (`TEST_POSTED`) in conjunction with `lu62_mc_post_on_receipt` and `lu62_wait` to provide non-blocking receive processing.
2. If the `return_code` to `lu62_mc_test` (POSTED) indicates that information is available to be received (`LU62_DATA` or `LU62_NO_DATA`), the local program should issue `lu62_mc_receive_and_wait` or `lu62_mc_receive_immediate` to receive the information.
3. If the `return_code` to `lu62_mc_test` (`TEST_POSTED`) is `LU62_UNSUCCESSFUL`, posting remains active for the conversation.
4. If the `return_code` to `lu62_mc_test` (`TEST_REQUEST_TO_SEND_RECEIVED`) is `LU62_OK`, a request-to-send notification is received by the local LU. `LU62_UNSUCCESSFUL` indicates that a request-to-send notification was not received.

See Also

`lu62_mc_post_on_receipt`, `lu62_mc_receive_and_wait`, `lu62_mc_receive_immediate`, `lu62_mc_test`

Type-Independent Verbs 10

Type-independent verbs are used on both basic and mapped conversations. Table 10-1 lists the type-independent verbs. Detailed man pages follow.

Table 10-1 SunLink LU6.2 Type-Independent Verbs

Verb Function	Function
*lu62_abort	Aborts conversation processing
lu62_accept	Listens for and accepts an incoming conversation
lu62_get_tp_properties	Returns information about the TPs
lu62_get_type	Returns the conversation type
*lu62_listen	Listens for an incoming conversation
lu62_register_tp	Registers a local TP for incoming conversations
*lu62_unregister_tp	Unregisters a local TP for incoming conversations
*lu62_send_ps_data	Sends Presentation Services data (sync-point)
lu62_wait	Waits for a conversation to be posted

10.1 *lu62_abort

lu62_abort is used to abort a conversation. It terminates any incomplete operation, i.e., a previous verb that received a return_code of LU62_OPERATION_INCOMPLETE, and causes the conversation (if active) to be deallocated (type=DA_ABEND_SVC).

lu62_abort may be issued in either blocking or non-blocking mode. It is designed, however, to be issued in non-blocking mode whenever it is necessary to abandon an incomplete operation. lu62_abort is the only way to terminate a conversation that is in this state. In this situation, an lu62_deallocate (type=DA_ABEND) verb would be refused by sending a return_code of LU62_PROGRAM_STATE_CHECK; lu62_abort overrides this state checking.

Synopsis

```
int lu62_abort(lu62_abort_t *rqp);
```

Request Structure

```
typedef struct {
    bit32     conv_id;                               /* s */
    bit32     return_code;                           /* r */
} lu62_abort_t;
```

10.1.1 lu62_abort_t *Request Structure Members*

The following subsections describe the lu62_abort_t request structure members:

conv_id

(supplied) Specifies the id of the conversation to use. conv_id is returned by lu62_allocate or lu62_accept.

return_code

(returned) Specifies the result of verb execution of one of the following:

- LU62_OK
- LU62_PARAMETER_CHECK
  - LU62_CONV_ID_UNKNOWN

State Changes

When `return_code` is `LU62_OK`, the conversation enters Reset state.

Usage Notes

When a program exits, the SunLink SNA PU2.1 9.1 server automatically aborts any active conversations. It is not necessary, therefore, to abort conversation(s) if your program is going to exit.

See Also

`lu62_wait_server`

10.2 *`lu62_accept`

`lu62_accept` is used to accept an incoming conversation. A client program can allocate and accept multiple conversations. To accept conversations, however, the client program must first register with its local LU to receive incoming conversations for designated local TPs using the `lu62_register_tp` verb.

Synopsis

```
int lu62_accept(lu62_accept_t *rqp);
```

Request Structure

```
typedef struct {
    bit32    port_id;                                /* s */
    bit32    own_tp_instance;                        /* s/r */
    lu62_processing_mode_e processing_mode;          /* s */
    bit32    conv_id;                                /* r */
    bit32    tp_id;                                  /* r */
    lu62_pip_presence_e pip_presence;                /* r */

    bit32     return_code;                           /* r */
} lu62_accept_t;
```

10.2.1 lu62_accept_t Request Structure Members

The following subsections describe the lu62_accept_t request structure members:

port_id

(supplied) Specifies the port_id of the LU connection to use. The port_id is returned by lu62_open.

processing_mode

(supplied) Specifies the initial processing mode of the conversation of one of the following:

- PM_BLOCKING
- PM_NON_BLOCKING

If processing_mode is set to PM_BLOCKING, lu62_accept does not return until a conversation is accepted (return_code = LU62_OK) or the verb fails for some reason. If processing_mode is set to PM_NON_BLOCKING and initial parameter checks pass, return_code is set to LU62_OPERATION_INCOMPLETE and lu62_wait_server must be issued to receive the eventual return.

The specified processing_mode remains in effect for the allocated conversation until lu62_set_processing_mode is issued or the conversation terminates.

own_tp_instance

(supplied / returned) Specifies the SunLink SNA PU2.1 9.1 server's identifier for the TP instance.

conv_id

(returned) Specifies the identifier of the accepted conversation. All subsequent verbs issued on this conversation require this identifier.

tp_id

(returned) Specifies the id of the registered TP for which the conversation is accepted.

pip_presence

(returned) Specifies whether the `pip_presence` field of the received FMH-5 attach request is set or not, as shown below.

- `PIP_NOT_PRESENT`
- `PIP_PRESENT`

If `PIP_PRESENT`, the caller is obliged to issue `lu62_receive_and_wait` (or `lu62_receive_immediate)` to receive the PIP variable (see Chapter 11 of the *IBM SNA Formats* manual).

return_code

(returned) Specifies the result of verb execution of one of the following:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  - `LU62_PORT_ID_UNKNOWN`
  - `LU62_BAD_PROCESSING_MODE`
  - `LU62_NO_TP_REGISTERED`
  - `LU62_UNKNOWN-TP`

State Changes

When `return_code` is `LU62_OK`, the conversation enters Send state.

Usage Notes

A program must issue `lu62_register_tp` for each local TP it supports. Wild card TP names may be specified. If multiple TPs are registered, determine which TP the conversation is for by correlating the `tp_id` returned by lu62_accept with one returned by lu62_register_tp. Alternatively, issue lu62_get_tp_properties on the newly accepted conversation and examine the returned tp_name.

See Also

lu62_register_tp, lu62_listen

Section 5.4, “Accepting Conversations,” provides an example of the use of this verb.

10.3 lu62_get_tp_properties

lu62_get_tp_properties returns information pertaining to the transaction program issuing the verb.

Synopsis

```
int lu62_get_tp_properties(lu62_get_tp_properties_t *rqp);
```

Request Structure

Table 10-2 Request Structure

```
typedef struct {
   bit32          conv_id;                          /* s */
   bit32          return_code;                      /* r */
    bit32         own_tp_instance;                  /* r */
   char           tp_name[LU62_TP_NAME_LEN+1];      /* r */
   bit8           qlu_name[LU62_NQ_LU_NAME_LEN+1];  /* r */
   int            qlu_name_len;                     /* r */
   char           user_id[LU62_MAX_USER_ID_LEN+1];  /* r */
   int            user_id_len;                      /* r */
    char          profile[LU62_MAX_PROFILE_LEN+1];  /* r */
   int            profile_len;                      /* r */
} lu62_get_tp_properties_t;
```

10.3.1 lu62_get_tp_properties_t *Request Structure Members*

The following subsections describe the lu62_get_tp_properties_t request structure members:

conv_id

(supplied) Specifies the id of the conversation to use. conv_id is returned by lu62_allocate or lu62_accept.

return_code

(returned) Specifies the result of verb execution of one of the following:

- LU62_OK
- LU62_PARAMETER_CHECK
  — LU62_CONV_ID_UNKNOWN

`own_tp_instance`

(returned) Specifies the SunLink SNA PU2.1 9.1 server's identifier for the TP instance.

`tp_name`

(returned) Specifies the name of the local transaction program. `tp_name` is an ASCII (null-terminated) string.

`qlu_name_len`

See `qlue_name_len` description.

`qlu_name`

(returned) Specifies the fully qualified name of the local LU. `qlu_name` is an ASCII (null-terminated) string and corresponds to the LU `NQ_LU_NAME` parameter in the configuration.

`user_id_len`

See `user_id` description.

`user_id`

(returned) The security `user_id`, if any, that was carried on the allocation request that initiated execution of the local program. `user_id` is an ASCII (null-terminated) string.

`profile_len`

See `profile` below.

`profile`

(returned) The security profile, if any, that was carried on the allocation request that initiated execution of the local program. `profile` is an ASCII (null-terminated) string.

State Changes

No state change occurs.

Usage Notes

If `lu62_get_tp_properties` is issued on a conversation that was allocated by the local program, a `tp_name` is known only if the allocation request specified `SECURITY_SAME`, and a `tp_id` was provided.

See Also

`lu62_allocate`, `lu62_mc_allocate`

Section 5.4, "Accepting Conversations," provides an example of the use of this verb.

10.4 `lu62_get_type`

`lu62_get_type` returns the conversation type, mapped or basic.

Synopsis

```
int lu62_get_type(lu62_get_type_t *rqp);
```

Request Structure

```
typedef struct {
    bit32          conv_id;                          /* s */
    lu62_conv_type_etype;                            /* r */
    bit32          return_code;                      /* r */
} lu62_get_type_t;
```

10.4.1 `lu62_get_type_t` *Request Structure Members*

The following subsections describe the `lu62_get_type_t` request structure members:

`conv_id`

(supplied) Specifies the id of the conversation to use. `conv_id` is returned by `lu62_allocate` or `lu62_accept`.

type

(returned) Specifies the conversation type from one of the following:

- `CONVERSATION_BASIC`
- `CONVERSATION_MAPPED`

`return_code`

(returned) Specifies the result of verb execution type from one of the following:

- `LU62_OK`
- `LU62_PARAMETER_CHECK`
  - — `LU62_CONV_ID_UNKNOWN`

State Changes

No state change occurs.

Usage Notes

The configuration allows local TPs to be configured to accept both mapped and basic conversations. If your program is such a TP, issue this verb when `lu62_accept` returns to determine the conversation type.

See Also

`lu62_allocate, lu62_mc_allocate`

Section 5.4, "Accepting Conversations," provides an example of the use of this verb.

10.5 *`lu62_listen`

`lu62_listen` is used to receive indication of an incoming conversation and is designed to let you build your own transaction dispatcher. It is used much like `lu62_accept` except that the SunLink SNA PU2.1 9.1 server does not allocate the conversation to the listening program. Instead, it holds the conversation until a subsequent `lu62_accept` is received. The listening program registers with its local LU to receive incoming conversations for designated local TPs using the `lu62_register_tp` verb.

`lu62_listen` is also used for sync-point recovery. Indication of an "implied forget" is sent to a listening sync-point manager (TP 06f2), see Appendix F.

Synopsis

```
int lu62_listen(lu62_listen_t *rqp);
```

Request Structure

```
typedef struct {
    bit32     port_id;                                /* s */
    lu62_processing_mode_e processing_mode;           /* s */
    bit32     conv_id;                                /* r */
/* Accept parameters */
    lu62_pip_presence_e pip_presence;                 /* r */
    lu62_conv_type_e type;                            /* r */
    char      uunique_session_name
              [LU62_UNIQUE_SESSION_NAME_LEN+1];      /* r */
/* Conversation Attributes - see lu62_get_attributes */
    char      partner_lu_name[LU62_LU_NAME_LEN+1];    /* r */
    char      mode_name[LU62_MODE_NAME_LEN+1];        /* r */
    bit8      partner_qlu_name[LU62_NQ_LU_NAME_LEN+1];/* r */
    int       partner_qlu_name_len                    /* r */
    lu62_sync_level_e sync_level;                     /* r */
    int       conv_corr_len;                          /* r */
    bit8      conv_corr[LU62_MAX_CONV_CORR_LEN];      /* r */
    bit32     conv_grp_id;                            /* r */
/* TP Properties - see lu62_get_tp_properties */
    bit32     tp_id;                                  /* r */
    bit32     own_tp_instance;                        /* r */
    char      tp_name[LU62_TP_NAME_LEN+1]             /* r */
    bit8      qlu_name[LU62_NQ_LU_NAME_LEN+1];        /* r */
    int       qlu_name_len;                           /* r */
    char      user_id[LU62_MAX_USER_ID_LEN+1];        /* r */
    int       user_id_len;                            /* r */
    char      profile[LU62_MAX_PROFILE_LEN+1];        /* r */
    int       profile_len;                            /* r */
    bit32     return_code;                            /* r */
/* Sync Level sync-point additions */
    lu62_response_type_e response_type;               /* r */
    int       sess_id_len;                            /* r */
    bit8      sess_id[LU62_MAX_SESS_ID_LEN];          /* r */
    int       luw_len;                                /* r */
    bit8      luw[LU62_MAX_LUW_LEN];                  /* r */
} lu62_listen_t;
```

10.5.1 `lu62_listen_t` *Request Structure Members*

The following subsections describe the `lu62_listen_t` request structure members:

`port_id`

(supplied) Specifies the `port_id` of the LU connection to use. The `port_id` is returned by `lu62_open`.

`processing_mode`

(supplied) Specifies the processing mode for listening from one of the following:

- `PM_BLOCKING`
- `PM_NON_BLOCKING`

If `processing_mode` is set to `PM_BLOCKING`, `lu62_listen` does not return until a conversation or implied forget is indicated (`return_code` = `LU62_OK`) or the verb fails for some reason. If `processing_mode` is set to `PM_NON_BLOCKING` and initial parameter checks pass, `return_code` is set to `LU62_OPERATION_INCOMPLETE` and `lu62_wait_server` must be issued to receive the eventual return.

`conv_id`

(returned) Specifies a temporary conversation identifier for `lu62_listen` verbs issued in `PM_NON_BLOCKING` mode.

`pip_presence`

(returned) Specifies whether the `pip_presence` field of the received FMH-5 attach request is set or not:

- `PIP_NOT_PRESENT`
- `PIP_PRESENT`

`type`

(returned) Specifies the conversation type as one of the following:

- `CONVERSATION_BASIC`
- `CONVERSATION_MAPPED`

unique_session_name

Used to specify the actual node used from the configuration instead of lu_name and node. The TP cannot use an lu_name in the previous open if unique session names are used.

partner_lu_name

(returned) Specifies the name of the partner LU at which the remote transaction program is located. partner_lu_name corresponds to the PTNR_LU NAME parameter in the configuration.

mode_name

(returned) Specifies the name of the selected mode. The conversation is allocated on a session of this mode. mode_name corresponds to the MODE NAME parameter in the configuration.

partner_qlu_name_len

See partner_qlu_name below.

partner_qlu_name

(returned) Specifies the fully qualified name of the partner LU at which the remote transaction program is located. partner_qlu_name corresponds to the PTNR_LU NQ_LU_NAME parameter in the configuration.

sync_level

(returned) Specifies the synchronization level for the conversation type from one of the following:

- SYNC_LEVEL_NONE

- SYNC_LEVEL_CONFIRM
  — SYNC_LEVEL_SYNCPT

conv_corr_len

See conv_corr below.

`conv_corr`

(ignored) Reserved for future use.

`conv_grp_id`

(ignored) Reserved for future use.

`tp_id`

(returned) Specifies the id of the registered TP for which the indicated conversation is received.

`own_tp_instance`

(returned) Specifies the SunLink SNA PU2.1 9.1 server's identifier for the TP instance. This value is used in a subsequent `lu62_accept` verb.

`tp_name`

(returned) Specifies the name of the local transaction program. `tp_name` is an ASCII (null-terminated) string.

`qlu_name_len`

See `qlu_name` below.

`qlu_name`

(returned) Specifies the fully qualified name of the local LU. `qlu_name` is an ASCII (null-terminated) string and corresponds to the LU `NQ_LU_NAME` parameter in the configuration.

`user_id_len`

See `user_id` below.

`user_id`

(returned) The security `user_id`, if any, that was carried on the allocation request that initiated execution of the local program. `user_id` is an ASCII (null-terminated) string.

profile_len

See profile below.

profile

(returned) The security profile, if any, that was carried on the allocation request that initiated execution of the local program. profile is an ASCII (null-terminated) string.

return_code

(returned) Specifies the result of verb execution from one of the following:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - — LU62_PORT_ID_UNKNOWN
  - — LU62_BAD_PROCESSING_MODE
  - — LU62_NO_TP_REGISTERED

response_type

(returned) Specifies the type of indication received from one of the following:

- LISTEN_ATTACH
  Indicates that an incoming conversation (FMH5 attach) was received for a registered TP
- LISTEN_FORGET
  Sent to the sync-point manager (TP 06f2) to indicate that an "implied forget" has occurred on the session identified by sess_id (see Appendix F).

sess_id_len

See sess_id below.

`sess_id`

(returned) Returns the assigned session identifier. The `sess_id` is returned as binary data. (Contrast this to `lu62_display_mode` and `lu62_deactivate_session` in which `session_id` is an ASCII-hex string).

`*luw_len`

See `*luw` below.

`*luw`

(returned) This extension to the *IBM SNA Transaction Programmer's Reference Manual* is provided for `SYNC_LEVEL_SYNCPT` (see Appendix F). `luw` contains the complete Logical Unit of Work Identifier as set in the FMH-5 attach that initiated the conversation (see Chapter 11 of the *IBM SNA Formats* manual).

State Changes

Not applicable.

Usage Notes

A program must issue `lu62_register_tp` for each local TP it supports. Wild card TP names may be specified.

See Also

`lu62_register_tp, lu62_accept`

Section 5.5, "Transaction Dispatch Using lu62_listen," provides an example of the use of this verb.

10.6 *lu62_register_tp

lu62_register_tp is used to register local TP names (including wild-carded TP names) with the SunLink SNA PU2.1 9.1 server, thus specifying that it will accept incoming conversations for those TPs. The TP name must already be configured on the local LU. A program may register many local TP names and, when a conversation is accepted, use lu62_get_tp_properties to determine the identity of the TP.

Synopsis

```
int lu62_register_tp(lu62_register_tp_t *rqp);
```

Request Structure

```
typedef struct {
    bit32       port_id;                            /* s */
    char        tp_name[LU62_TP_NAME_LEN+1];        /* s */
    bit32       tp_id;                              /* r */
    bit32       return_code;                        /* r */
} lu62_register_tp_t;
```

10.6.1 lu62_register_t *Request Structure Members*

The following subsections describe the lu62_register_t request structure members:

port_id

(supplied) Specifies the port_id of the LU connection to use. The port_id is returned by lu62_open.

tp_name

(supplied) Specifies the name of the local transaction program. This corresponds to the TP NAME parameter in the configuration. tp_name is an ASCII (null-terminated) string.

`tp_id`

(returned) Specifies the id of the newly registered TP.

`return_code`

(returned) Specifies the result of verb execution from one of the following:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
- `LU62_PORT_ID_UNKNOWN`
- `LU62_TP_ALREADY_REGISTERED`
- `LU62_PARAMETER_ERROR`
- `LU62_TP_UNKNOWN`

State Changes

Not applicable.

Usage Notes

1. A program must issue `lu62_register_tp` for each local TP for which it is prepared to handle conversations. When a conversation is attached to the local TP, `lu62_accept` returns the `tp_id` of the local TP. When multiple local TPs are registered, correlate this `tp_id` with those returned by the `lu62_register_tp` calls to determine which TP the conversation is for. Alternatively, issue `lu62_get_tp_properties` on the newly accepted conversation and examine the returned `tp_name`.

2. A program can specify a wild-carded TP name when it calls `lu62_register_tp`. The wild-carded name must, however, be configured on the local LU; i.e, the `tp_name` parameter must match exactly with a TP `TP_NAME` configuration parameter. * and ? matches are allowed, where a * is used to match any sub-string (including NULL), and ? matches any single character.

Be sure to configure the wild-carded TP names correctly. When an FMH5 Attach is received, the SunLink SNA PU2.1 9.1 server searches through its local TPs in the order in which they are configured, and uses the first match it finds to locate a potential acceptor or listener. Thus, if a TP name matches both a specific name and a wild-card, the first configured will be used. Ideally, a local TP should match one configuration entry.

See Also

`lu62_accept`, `lu62_listen`

Section 5.4, “Accepting Conversations,” provides an example of the use of this verb with `lu62_accept`.

Section 5.5, “Transaction Dispatch Using lu62_listen,” provides an example of the use of this verb with `lu62_listen` and wild-carded TP names.

10.7 *lu62_unregister_tp

`lu62_unregister_tp` is used to unregister a local TP name (including a wild-carded TP name) from the SunLink SNA PU2.1 9.1 server, thus specifying that the calling program will no longer accept incoming conversations for that TP. The TP must already be registered on the local LU. Previously issued and still pending `lu62_accept` and `lu62_listen` verbs are unaffected.

Synopsis

```
int lu62_unregister_tp(lu62_unregister_tp_t * rgp);
```

Request Structure

```
typedef struct {
    bit32       port_id;                             /* s */
    bit32       tp_id;                               /* s */
    bit32       return_code;                         /* r*/
} lu62_send_ps_data_t;
```

10.7.1 lu62_send_ps_data_t *Request Structure Members*

The following subsections describe the lu62_send_ps_data_t request structure members:

port_id

(supplied) Specifies the port_id of the LU connection to use. The port_id is returned by lu62_open.

tp_id

(supplied) Specifies the id of the registered TP.

return_code

(returned) Specifies the result of verb execution, one of the following:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
  - — LU62_PARAMETER_CHECK
  - — LU62_PORT_ID_UNKNOWN
  - — LU62_PARAMETER_ERROR
  - — LU62_TP_UNKNOWN

State Changes

Not applicable.

Usage Notes

1. If a local TP is unregistered while an LU62_accept verb is outstanding, that lu62_accept could still return a conversation for the unregistered TP. In this case, the returned tp_id will be unknown and cannot be related with a tp_ids returned by a previous lu62_register_tp. Issue lu62_get_tp_properties on the newly accepted conversation and examine the returned tp_name.

See Also

lu62_accept, lu62_listen, lu62_register_tp

*lu62_send_ps_data

This extension to the *IBM SNA Transaction Programmer's Reference Manual* is provided to send sync-point flows on SYNC_LEVEL_SYNCPT conversations (see Appendix F). lu62_send_ps_data is used to send PS headers to the remote program. The data consists of a completely formatted PS header, including the two-byte long LL field, specifying an invalid length of one byte.

Synopsis

```
int lu62_send_ps_data(lu62_send_ps_data_t *rqp);
```

Request Structure

```
typedef struct {
    bit32       conv_id;                               /* s */
    bit8        *data;                                 /* s */
    int         length;                                /* s */
    lu62_forget_eforget;                               /* s */
    lu62_flush_eflush;                                 /* s */
    bit32       return_code;                           /* r */
    bit32       request_to_send_received;              /* r */
} lu62_send_ps_data_t;
```

10.7.2 lu62_send_ps_data_t *Request Structure Members*

The following subsections describe the lu62_send_ps_data_t request structure members:

conv_id

(supplied) Specifies the id of the conversation to use. conv_id is returned by lu62_(mc_)allocate or lu62_accept.

data

(supplied) The address of the user buffer containing the completely formatted PS Header, including the two-byte long LL field specifying an invalid length of one byte.

`length`

(supplied) Specifies the length of the data to be sent.

`forget`

(supplied) Set by the sync-point agent to indicate that it has “committed” the transaction. When the next normal flow is received on the session, the SunLink SNA PU2.1 9.1 server notifies a listening sync-point manager of the “implied forget.” See `lu62_listen`.

`flush`

(supplied) Specifies whether the PS header is to be sent to the remote program immediately or buffered in the local LU's send buffer. `flush` is set to one of the following:

- `FLUSH_NO`
- `FLUSH_YES`

`return_code`

(returned) Specifies the result of verb execution, one of the following:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
- `LU62_CONV_ID_UNKNOWN`
- `LU62_NULL_DATA`
- `LU62_BAD_LENGTH`
- `LU62_PROGRAM_STATE_CHECK`
  - — Conversation is not is Send state.
  - — `LU62_VERB_IN_PROGRESS`
- `LU62_ALLOCATION_ERROR`
- `LU62_DEALLOCATE_ABEND_PROG`
- `LU62_DEALLOCATE_ABEND_TIMER`
- `LU62_DEALLOCATE_ABEND_SVC`
- `LU62_PROG_ERROR_PURGING`
- `LU62_SVC_ERROR_PURGING`
- `LU62_RESOURCE_FAILURE_NO_RETRY`
- `LU62_RESOURCE_FAILURE_RETRY`

`request_to_send_received`

(returned) Indicates whether or not a request-to-send indication is received from the remote program as follows:

- TRUE, indicates that a request-to-send indication was received
- FALSE, indicates that a request-to-send indication was not received

State Changes

No state change occurs.

Usage Notes

1. The local LU buffers the data to be sent to the remote LU until it accumulates a sufficient amount of data for transmission (from one or more `lu62_send_(ps_)` data verbs), or until the local program issues a call that causes the LU to flush its send buffer. The amount of data sufficient for transmission depends on the characteristics of the session allocated for the conversation, and varies from one session to another.

2. When `request_to_send_received` is TRUE, the remote program requests that the local program "give up the turn", that is, that it enter Receive state, thereby placing the remote program in Send state. The local program enters Receive state by issuing `lu62_prep_to_receive` or `lu62_receive_and_wait`. The remote program issues `lu62_receive_immediate` or `lu62_receive_and_wait` to receive the resulting Send indication (`what_received` = `WR_SEND`).

See Also

Appendix F, "LU6.2 Sync-Point," discusses the use of this verb.

10.8 `lu62_wait`

`lu62_wait` is used to wait for posting to occur on any basic or mapped conversation. A list of conversations for which posting is expected is supplied. Posting of a conversation occurs when posting is active for the conversation and the LU has information that the program can receive, that is, data, status, or a confirmation request. When `lu62_wait` returns, the program should issue `lu62_receive_immediate` or `lu62_mc_receive_immediate` to receive the information.

Synopsis

```
int lu62_wait(lu62_wait_t *rqp);
```

Request Structure

```
typedef struct {
    bit32       port_id;                                /* s */
    int         conv_count;                             /* s */
    bit32       *conv_list;                             /* s */
    bit32       conv_id;                                /* r */
    bit32       return_code;                            /* r */
} lu62_wait_t;
```

10.8.1 `lu62_wait_t` *Request Structure Members*

The following subsections describe the `lu62_wait_t` request structure members:

`port_id`

(supplied) Specifies the port_id of the LU connection to use. The `port_id` is returned by `lu62_open`.

`conv_count`

(supplied) Specifies the number of conversations in the following `conv_list`.

`conv_list`

(supplied) Points to an array of `conv_ids`, of length `conv_count`, specifying the conversations for which posting is expected.

`conv_id`

(returned) Specifies the `conv_id` of the first conversation in the `conv_list` to be posted.

return_code

(returned) Specifies the result of verb execution.

If a mapped conversation is posted, return_code can be one of the following:

- LU62_OK
  - LU62_OK_DATA
  - LU62_OK_NOT_DATA
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - LU62_CONV_ID_UNKNOWN
- LU62_PROGRAM_STATE_CHECK
  - LU62_VERB_IN_PROGRESS
  - LU62_POSTING_NOT_ACTIVE
- LU62_ALLOCATION_ERROR
- LU62_DEALLOCATE_ABEND
- LU62_DEALLOCATE_NORMAL
- LU62_FMH_DATA_NOT_SUPPORTED
- LU62_PROG_ERROR_NO_TRUNC
- LU62_PROG_ERROR_PURGING
- LU62_RESOURCE_FAILURE_NO_RETRY
- LU62_RESOURCE_FAILURE_RETRY

If a basic conversation is posted, return_code can be one of the following:

- LU62_OK
  - LU62_OK_DATA
  - LU62_OK_NOT_DATA
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - LU62_CONV_ID_UNKNOWN
- LU62_PROGRAM_STATE_CHECK
  - LU62_VERB_IN_PROGRESS
- LU62_ALLOCATION_ERROR
- LU62_DEALLOCATE_ABEND_PROG
- LU62_DEALLOCATE_ABEND_TIMER
- LU62_DEALLOCATE_ABEND_SVC

- `LU62_DEALLOCATE_NORMAL`
- `LU62_PROG_ERROR_NO_TRUNC`
- `LU62_PROG_ERROR_PURGING`
- `LU62_PROG_ERROR_TRUNC`
- `LU62_SVC_ERROR_NO_TRUNC`
- `LU62_SVC_ERROR_PURGING`
- `LU62_SVC_ERROR_TRUNC`
- `LU62_RESOURCE_FAILURE_NO_RETRY`
- `LU62_RESOURCE_FAILURE_RETRY`

State Changes

No state change occurs.

Usage Notes

1. This verb is used in conjunction with `lu62_(mc_)post_on_receipt` to provide non-blocking receive processing.

2. The `conv_list` may contain any combination of basic and mapped conversations. Posting for each conversation may be active or inactive. The verb waits for posting to occur only on those conversations for which posting is active.

See Also

`lu62_(mc_)post_on_receipt`, `lu62_(mc_)receive_immediate`, `lu62_(mc_)test`

Control Operator Verbs 11

Control operator (COPR) verbs are used to control and monitor the operation of the local LU. SunLink P2P LU6.2 9.1 COPR verbs are divided into the following categories:

- Change number of sessions (CNOS) verbs
- CNOS notification verbs
- Session control verbs
- Display verbs

Detailed man pages for the verbs follow by category.

11.1 Control Operator Privileges

To perform COPR functions, programs require special privileges and must be defined as TPs in the SunLink P2P LU6.2 9.1 configuration. The `TP PRIVILEGE` parameter is used to assign privileges to TPs. A control operator program registers its TP name with the SunLink SNA PU2.1 9.1 server when it opens a connection to the server using `lu62_open`.

11.2 CNOS Verbs

Change-number-of-sessions verbs allow a local program to change the (LU, mode) session limit, which controls the number of LU-LU sessions per mode that are available for conversations and, for parallel-sessions, to establish the contention-winner polarities for the mode. The CNOS verbs apply to both single- and parallel-session modes. Table 11-1 summarizes the CNOS verbs.

Table 11-1 SunLink LU6.2 CNOS Verbs

Verb	**Function**
`lu62_change_session_limit`	Changes (LU,mode) session limit for parallel sessions
`lu62_initialize_session_limit`	Initialize (LU,mode) session limits
`lu62_reset_session_limit`	Resets (LU,mode) session limit to 0

11.2.1 CNOS Privilege

To issue CNOS verbs, your program must be configured with TP PRIVILEGE = CNOS.

11.2.2 Single-Sessions and SNA Service Manager (SNASVCMG)

Single-session connections, and SNASVCMG, may be initialized or reset. The requested session limits are applied only at the local LU.

11.2.3 Parallel-Sessions

For parallel-session modes, a CNOS negotiation is initiated with the partner LU. Both LUs are involved in processing the changes, and parameter negotiation may occur.

You may use the CNOS notification verbs to receive notifications of all CNOS events that occur on the local LU, whether initiated locally (by your program or another control operator program) or remotely.

11.2.4 lu62_change_session_limit

lu62_change_session_limit changes the session limit and the contention-winner polarities for parallel-session connections.

Synopsis

```
int lu62_change_session_limit(lu62_change_session_limit_t *rqp);
```

Request Structure

```
typedef struct {
    bit32       port_id;                            /* s */
    char        lu_name[LU62_LU_NAME_LEN+1];        /* s */
    char        mode_name[LU62_MODE_NAME_LEN+1];    /* s */
    int         lu_mode_session_limit;              /* s */
    int         min_conwinners_source;              /* s */
    int         min_conwinners_target;              /* s */
    lu62_responsible_lu_e responsible_lu;           /* s */
    bit32       return_code;                        /* r */
} lu62_change_session_limit_t;
```

11.2.5 lu62_change_session_limit_t *Request Structure Members*

The following subsections describe the lu62_change_session_limit_t request structure members:

port_id

(supplied) Specifies the port_id of the LU connection to use. The port_id is returned by lu62_open.

lu_name

(supplied) Specifies the locally known name of the partner LU to which the change applies. lu_name is supplied as an ASCII (null-terminated) string. It corresponds to the PTNR_LU NAME parameter in the configuration.

mode_name

(supplied) Specifies the name of the mode to be updated. mode_name is supplied as an ASCII (null-terminated) string and is translated to EBCDIC by the SunLink SNA PU2.1 9.1 server. It corresponds to the MODE NAME parameter in the configuration.

lu_mode_session_limit

(supplied) Specifies the maximum number of sessions to be allowed between the local (source) LU and the partner LU, for the specified mode_name. lu_mode_session_limit must be greater than 0, and greater than or equal to the sum of min_conwinners_source + min_conwinners_target.

min_conwinners_source

(supplied) Specifies the minimum number of sessions for which the local LU is the contention winner. min_conwinners_source must be greater than or equal to 0.

min_conwinners_target

(supplied) Specifies the minimum number of sessions for which the partner LU is the contention winner. min_conwinners_target must be greater than or equal to 0.

responsible_lu

(supplied) Specifies which LU is responsible for deactivating sessions as a result of a decrease in the session limit and contention-winners; see Usage Note 1 for more discussion. responsible_lu can be one of the following:

- SL_SOURCE
- SL_TARGET

return_code

(returned) Specifies the result of verb execution from one of the following:

- LU62_OK
  — LU62_OK_AS_SPECIFIED
  — LU62_OK_AS_NEGOTIATED
- LU62_OPERATION_INCOMPLETE

- LU62_PARAMETER_CHECK
  - LU62_PROGRAM_NOT_PRIVILEGED
  - LU62_PORT_ID_UNKNOWN
  - LU62_LU_NAME_REQD
  - LU62_MODE_NAME_REQD
  - LU62_BAD_LU_NAME
  - LU62_BAD_MODE_NAME
  - LU62_BAD_RESPONSIBLE_LU
  - LU62_BAD_SESSION_LIMIT
  - LU62_BAD_MIN_CONWINNERS
- LU62_PROGRAM_STATE_CHECK
  - LU62_VERB_IN_PROGRESS
- LU62_PARAMETER_ERROR
  - LU62_UNKNOWN_PARTNER_LU
  - LU62_UNKNOWN_MODE
- LU62_ALLOCATION_ERROR
- LU62_COMMAND_RACE_REJECT
- LU62_MODE_SESSION_LIMIT_ZERO
- LU62_SESSION_LIMIT_EXCEEDED
- LU62_REQUEST_EXCEEDS_MAX_ALLOWED
- LU62_RESOURCE_FAILURE_NO_RETRY
- LU62_UNRECOGNIZED_MODE_NAME

Usage Notes

1. As a result of this verb, sessions may be activated, deactivated, or both activated and deactivated to conform to the new session limits. The `responsible_lu` deactivates its own conwinner sessions until the new session limit is exceeded or it reaches its `min_conwinners`. Either LU may automatically activate sessions to acquire conwinner sessions. SunLink LU6.2 LUs will acquire conwinner sessions to satisfy outstanding requests and to reach the minimum of its `min_conwinners` and its configured auto-activation limit, LU AUTOACT_LMT.

2. Sessions with active conversations are not deactivated until the conversation is deallocated.

See Also

CNOS notification verbs.

11.2.6 `lu62_init_session_limit`

`lu62_init_session_limit` initializes the session limit for single- and parallel-session connections, and the contention-winner polarities for parallel-session connections.

Synopsis

```
int lu62_init_session_limit(lu62_init_session_limit_t *rqp);
```

Request Structure

```
typedef struct {
    bit32       port_id;                            /* s */
    char        lu_name[LU62_LU_NAME_LEN+1];        /* s */
    char        mode_name[LU62_MODE_NAME_LEN+1];    /* s */
    int         lu_mode_session_limit;              /* s */
    int         min_conwinners_source;              /* s */
    int         min_conwinners_target;              /* s */
    bit32       return_code;                        /* r */
} lu62_init_session_limit_t;
```

11.2.7 `lu62_init_session_limit_t` *Request Structure Members*

The following subsections describe the `lu62_init_session_limit_t` request structure members:

`port_id`

(supplied) Specifies the `port_id` of the LU connection to use. The `port_id` is returned by `lu62_open`.

lu_name

(supplied) Specifies the locally known name of the partner LU to which the initialization applies. lu_name is supplied as an ASCII (null-terminated) string. It corresponds to the PTNR_LU NAME parameter in the configuration.

mode_name

(supplied) Specifies the name of the mode to be initialized. mode_name is supplied as an ASCII (null-terminated) string and is translated to EBCDIC by the SunLink SNA PU2.1 9.1 server. It corresponds to the MODE NAME parameter in the configuration.

lu_mode_session_limit

(supplied) Specifies the maximum number of sessions to be allowed between the local (source) LU and the partner LU, for the specified mode_name. lu_mode_session_limit must be greater than 0, and greater than or equal to the sum of min_conwinners_source + min_conwinners_target. For single-session modes, lu_mode_session_limit must be 1.

min_conwinners_source

(supplied) Specifies the minimum number of sessions for which the local LU is the contention winner. min_conwinners_source must be greater than or equal to 0.

min_conwinners_target

(supplied) Specifies the minimum number of sessions for which the partner LU is the contention winner. min_conwinners_target must be greater than or equal to 0.

return_code

(returned) Specifies the result of verb execution. For single- and parallel-sessions, return_code may be one of the following:

- LU62_OK
  - LU62_OK_AS_SPECIFIED
  - LU62_OK_AS_NEGOTIATED
- LU62_OPERATION_INCOMPLETE

- LU62_PARAMETER_CHECK
  - LU62_PROGRAM_NOT_PRIVILEGED
  - LU62_PORT_ID_UNKNOWN
  - LU62_LU_NAME_REQD
  - LU62_MODE_NAME_REQD
  - LU62_BAD_LU_NAME
  - LU62_BAD_MODE_NAME
  - LU62_BAD_SESSION_LIMIT
  - LU62_BAD_MIN_CONWINNERS
- LU62_PROGRAM_STATE_CHECK
  - LU62_VERB_IN_PROGRESS
- LU62_PARAMETER_ERROR
  - LU62_UNKNOWN_PARTNER_LU
  - LU62_UNKNOWN_MODE
- LU62_COMMAND_RACE_REJECT
- LU62_MODE_SESSION_LIMIT_NOT_ZERO
- LU62_SESSION_LIMIT_EXCEEDED
- LU62_REQUEST_EXCEEDS_MAX_ALLOWED

For parallel-sessions, in which a CNOS conversation occurs with the partner LU, `return_code` may additionally be one of the following:

- LU62_ALLOCATION_ERROR
- LU62_MODE_SESSION_LIMIT_CLOSED
- LU62_RESOURCE_FAILURE_NO_RETRY
- LU62_UNRECOGNIZED_MODE_NAME

Usage Notes

1. Single-session and SNASVCMG limits are initialized in the local LU only. Thus the initialization must be performed at both LUs for a session to be activated. Further, the contention-winner polarity initialized for each LU must be consistent, i.e., one LU must be configured to be the conwinner, the other to be the conloser.

2. SNASVCMG modes must be initialized to:
   - `lu_mode_session_limit = 2`
   - `min_conwinners_source = 1`
   - `min_conwinners_target = 1`
3. To establish the session limits for a parallel-session connection between a local and a target LU, the session limit and contention-winner polarities for the SNASVCMG connection between the two LUs must first be initialized. SNASVCMG is initialized locally so that it can be allocated to the CNOS conversations used to establish the initial limits for other parallel sessions.
4. As a result of this verb, sessions may be activated by either LU. LUs activate sessions to satisfy queued conversation allocation requests. LUs also auto-activate conwinner sessions to acquire a set of available conwinner sessions for future conversations. Sessions may also be activated manually by local control operators issuing `lu62_activate_session` requests. SunLink LU6.2 LUs will acquire conwinner sessions to satisfy outstanding requests and to reach the minimum of its `min_conwinners` and its configured auto-activation limit, LU `CW_AUTOACT_LMT`.

See Also

CNOS Notification verbs.

11.2.8 `lu62_reset_session_limit`

`lu62_reset_session_limit` resets the session limit for single- and parallel session connections, and the contention-winner polarities for parallel-session connections. The verb may be issued for a specified mode, or for all modes (except SNASVCMG) between the local and partner LUs. All active sessions for the specified (LU, mode) are deactivated as a result of this verb. No sessions carrying active conversations are deactivated, however, until the conversation is deallocated. As an option, currently pending allocation requests can also be satisfied before the (LU, mode) is closed. An `lu62_init_session_limit` verb is required before the (LU, mode) can be re-used.

Synopsis

```
int lu62_reset_session_limit(lu62_reset_session_limit_t *rqp);
```

Request Structure

```
typedef struct }
    bit32  port_id;                                          /* s */
    char   lu_name[LU62_LU_NAME_LEN+1];                      /* s */
    char   mode_name[LU62_MODE_NAME_LEN+1];                  /* s */
   lu62_responsible_lu_e responsible_lu;                     /* s */
    int    drain_source;                                     /* s */
    int    drain_target;                                     /* s */
    int    force;                                            /* s */
    bit32  return_code;                                      /* r */
} lu62_reset_session_limit_t;
```

11.2.9 lu62_reset_session_limit_t *Request Structure Members*

The following subsections describe the lu62_reset_session_limit_t request structure members:

port_id

(supplied) Specifies the port_id of the LU connection to use. The port_id is returned by lu62_open.

lu_name

(supplied) Specifies the locally known name of the partner LU to which the reset applies. lu_name is supplied as an ASCII (null-terminated) string. It corresponds to the PTNR_LU NAME parameter in the configuration.

mode_name

(supplied) Specifies the name of the mode to be reset, or LU62_ALL_MODES, to indicate all modes between the local and partner LU except SNASVCMG. mode_name is supplied as an ASCII (null-terminated) string and is translated to EBCDIC by the SunLink SNA PU2.1 9.1 server. If not, `LU62_ALL_MODES` and `mode_name` correspond to the `MODE NAME` parameter in the configuration.

`responsible_lu`

(supplied) Specifies which LU is responsible for deactivating sessions as a result of the reset from one of the following:

- `SL_SOURCE`
- `SL_TARGET`

The `drain_source` and `drain_target` parameters determine when the `responsible_lu` can deactivate the sessions. If an LU is to drain its allocation requests, the `responsible_lu` does not deactivate a session until the current conversation is deallocated and no allocation request is queued for the session.

`drain_source`

(supplied) Specifies whether the source LU can drain its allocation requests. `drain_source` can be one of the following:

- FALSE: All queued allocation requests, and all subsequent allocation requests are refused by the local LU with a `return_code`, `LU62_ALLOCATION_FAILURE_NO_RETRY`, indicating that the session limit is zero
- TRUE: All allocation requests received while the source LU is draining are satisfied. Draining ends when no allocation requests are queued; all subsequent allocation requests are refused with a `return_code` of `LU62_ALLOCATION_FAILURE_NO_RETRY,` indicating that the session limit is zero

This parameter is not applicable to `SNASVCMG`.

`drain_source`

(supplied) Specifies whether the target LU can drain its allocation requests. `drain_target` can be one of the following:

- FALSE: All queued allocation requests, and all subsequent allocation requests are refused by the target LU with a `return_code`, `LU62_ALLOCATION_FAILURE_NO_RETRY`, indicating that the session limit is zero

- TRUE: All allocation requests received while the target LU is draining are satisfied. Draining ends when no allocation requests are queued; all subsequent allocation requests are refused with a `return_code` of `LU62_ALLOCATION_FAILURE_NO_RETRY`, indicating that the session limit is zero

This parameter is not applicable to `SNASVCMG`.

`force`

(supplied) For parallel-session connections, this parameter requests that the source LU reset the session limit even when certain error conditions occur that prevent the CNOS negotiation. `force` can be one of the following:

- FALSE
- TRUE

This parameter is not applicable to single-session connections or SNASVCMG.

`return_code`

(returned) Specifies the result of verb execution. For single- and parallel-sessions, regardless of whether `force` is set or not, `return_code` may be one of the following:

- `LU62_OK`
  - `LU62_OK_AS_SPECIFIED`
  - `LU62_OK_AS_NEGOTIATED`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  - `LU62_PROGRAM_NOT_PRIVILEGED`
  - `LU62_PORT_ID_UNKNOWN`
  - `LU62_LU_NAME_REQD`
  - `LU62_MODE_NAME_REQD`
  - `LU62_BAD_LU_NAME`
  - `LU62_BAD_MODE_NAME`
  - `LU62_BAD_SESSION_LIMIT`
  - `LU62_PLU_SESSION_LIMIT_NOT_ZERO`
    SNASVCMG is specified and one or more session (LU, mode) session limits for the partner LU are not 0.

  - LU62_DRAIN_SOURCE_NO_REQD
    Session limit is already 0, and drain_source is set.
- LU62_PROGRAM_STATE_CHECK
  - LU62_VERB_IN_PROGRESS
- LU62_PARAMETER_ERROR
  - LU62_UNKNOWN_PARTNER_LU
  - LU62_UNKNOWN_MODE
- LU62_COMMAND_RACE_REJECT

For parallel-sessions, in which a CNOS conversation occurs with the partner LU, and force is NOT set, return_code may additionally be one of the following:

- LU62_ALLOCATION_ERROR
- LU62_MODE_SESSION_LIMIT_CLOSED
- LU62_RESOURCE_FAILURE_NO_RETRY
- LU62_UNRECOGNIZED_MODE_NAME

If force is set and the CNOS conversation fails for one of the above reasons, return_code is:

- LU62_OK_FORCED

Usage Notes

1. Single-session limits are reset at the local LU. The source LU deactivates the session, if it is active, in accordance with the drain parameters.

2. SNASVCMG modes are reset at the local LU, provided that no other mode between the source and target LU has non-zero session limits. These modes could, however, be in the processing of draining. The local LU, therefore, does not deactivate its conwinner SNASVCMG session, if active, until all other sessions between the two LUs are deactivated.

3. This verb can be issued when the session limit is already zero to discontinue draining at the source or target LU.

See Also

CNOS Notification verbs.

11.3 CNOS Notification Verbs

Your program can request to receive notifications of CNOS events occurring at the local LU. Table 11-2 below, summarizes the CNOS notification verbs.

Table 11-2 SunLink LU6.2 CNOS Notification Verbs

Verb	Function
*`lu62_request_notification`	Requests notifications of CNOS events
*`lu62_stop_notification`	Stops notifications of CNOS events
*`lu62_poll_notification`	Tests for pending CNOS event
*`lu62_receive_notification`	Receives CNOS event

11.3.1 Receiving CNOS Notifications

A separate socket connection is opened to the SunLink SNA PU2.1 9.1 server using the `lu62_request_notification` verb. This verb is issued for each LU for which you want to receive notifications. The `lu62_receive_notification` is used to read the notification from the socket. `lu62_receive_notification` will, however, block until a notification is available to be read. To prevent blocking, your program can poll the connection for pending notifications using the `lu62_poll_notification` verb, or it can use the Unix `select` call to receive CNOS notifications asynchronously. When you want to stop receiving notifications from a particular LU, use the `lu62_stop_notification` call.

Eventual Returns

CNOS notification verbs do not operate in the same way as other COPR verbs, or any other SunLink LU6.2 API verb. `lu62_request_notification` and `lu62_stop_notification` cause messages to be sent to the SunLink SNA PU2.1 9.1 server and then return immediately to the caller with a `return_code` of `LU62_OPERATION_INCOMPLETE`. The server generates responses to these verbs that you must receive using `lu62_receive_notification.` Unlike the other API verbs you do not have the option of setting a blocking processing mode to cause the API to wait for the verb to complete.

See Section 5.11, “Using the Select Call to Receive CNOS Notifications,” for an example of how the `select` call may be used to multiplex program control between COPR verb returns, CNOS events, and terminal input.

11.4 lu62_receive_notification

`lu62_receive_notification` is used to receive notifications from the SunLink SNA PU2.1 9.1 server.

Synopsis

```
int lu62_receive_notification(lu62_notification_header_t *hp,
                                          bit8 *data);
```

Parameters

`hp`

(supplied) The notification header, see below.

`data`

(supplied) Specifies the address of a data buffer to receive any data associated with the notification. The format of the data depends on `hp->op_code`.

Return Value

`LU62_OK`

Indicates that a notification was received successfully from the SunLink SNA PU2.1 9.1 server.

`LU62_ERROR`

Indicates that the API has rejected the request. *lu62_errno* is set to indicate the reason for failure which can be one of the following:

- `LU62_PARAMETER_ERROR`
- `LU62_PORT_ID_UNKNOWN`

- `LU62_SERV_DCNX`
  The connection to the server has failed.

Notification Header

```
typedef struct {
    bit32          port_id;                              /* s */
    lu62_op_code_e op_code;                              /* r */
    bit32          return_code;                          /* r */
} lu62_notification_header_t;
```

11.4.1 `lu62_notification_header_t` *Request Structure Members*

The following subsections describe the `lu62_notification_header_t` request structure members:

`port_id`

(supplied) Specifies the port id of the LU connection to use. The `port_id` is returned by `lu62_request_notification`.

`op_code`

(returned) Specifies the type of notification received as one of the following:

- `LU62_REQUEST_NOTIFICATION_REPLY`
  The notification is the server response to an `lu62_request_notification` verb. There is no associated data.
- `LU62_STOP_NOTIFICATION_REPLY`
  The notification is the server response to an `lu62_stop_notification` verb. There is no associated data.
- `LU62_CNOS_NOTIFICATION`
  Unsolicited CNOS notification. The data buffer contains the CNOS notification, see below.

`return_code`

(returned) For verb replies, `return_code` specifies the result of verb execution in the SunLink SNA PU2.1 9.1 server. See the appropriate verb for specific values. For unsolicited notifications, `return_code` is set to `LU62_OK`.

CNOS Notification

```
typedef struct {
    lu62_cnos_type_ecnos_type;
    int          local_invocation;
    int          lu_mode_session_limit;
    int          min_conwinners_source;
    int          min_conwinners_target;
    lu62_responsible_lu_e  responsible_lu;
    int          drain_source;
    int          drain_target;
    int          force;
    char         lu_name[LU62_LU_NAME_LEN+1];
    char         mode_name[LU62_MODE_NAME_LEN+1];
} lu62_cnos_notification_t;
```

11.4.2 `lu62_cnos_notification_t` Request Structure

The following subsections describe the `lu62_cnos_notification_t` request structure members:

`cnos_type`

(returned) Specifies the type of CNOS request received by the local LU from the following:

- `LU62_RESET_SESSION_LIMIT`
- `LU62_INIT_SESSION_LIMIT`
- `LU62_CHANGE_SESSION_LIMIT`

`lu_name`

(returned) Specifies the name of the affected partner LU. `lu_name` is an ASCII (null-terminated) string. It corresponds to the `PTNR_LU NAME` parameter in the configuration.

`mode_name`

(returned) Specifies the name of the affected mode. `mode_name` is an ASCII (null-terminated) string. It corresponds to the `MODE NAME` parameter in the configuration.

`local_invocation`

(returned) Specifies whether the corresponding CNOS verb was issued by the local LU or, for parallel-session connections, the local LU. `local_invocation` can be one of the following:

- TRUE: The verb was issued by a COPR program attached to the local LU (including potentially the current program).
- FALSE: The verb was issued by the remote LU.

`lu_mode_session_limit`

(returned) Specifies the maximum number of sessions to be allowed between the local (source) LU and the partner LU, for the specified `mode_name`. For `cnos_type = LU62_RESET_SESSION_LIMIT`, `lu_mode_session_limit` should be zero.

`min_conwinners_source`

(returned) Specifies the minimum number of sessions for which the local LU is the contention winner.

`min_conwinners_target`

(returned) Specifies the minimum number of sessions for which the partner LU is the contention winner.

`responsible_lu`

(returned) For `cnos_types` of `LU62_RESET_SESSION_LIMIT` and `LU62_CHANGE_SESSION_LIMIT`, this value specifies which LU is responsible for deactivating sessions as a result of a decrease in the session limit and contention-winners. `responsible_lu` can be one of the following:

- `SL_SOURCE`
- `SL_TARGET`

This following structure members are only applicable when `cnos_type` = `LU62_RESET_SESSION_LIMIT`.

`drain_source`

(returned) Specifies whether the source LU can drain its allocation requests. `drain_source` can be one of the following:

- FALSE
- TRUE

This parameter is not applicable to SNASVCMG.

`force`

(returned) For parallel-session connections, this parameter requests that the source LU reset the session limit even when certain error conditions occur that prevent the CNOS negotiation. `force` can be one of the following:

- FALSE
- TRUE

This parameter is not applicable to single-session connections or SNASVCMG.

See Also

Section 11.2, “CNOS Verbs.”

`lu62_request_notification, lu62_stop_notification`

11.4.3 lu62_request_notification

lu62_request_notification requests that the SunLink SNA PU2.1 9.1 server send notifications of all CNOS events relating to a local LU.

If parameters are valid, lu62_request_notification opens a socket connection to the specified server and sends the request on to the server. The verb then returns to the caller with a return_code of LU62_OPERATION_INCOMPLETE. The caller must issue lu62_receive_notification to receive the eventual return, which indicates whether the server accepted the request or not.

Synopsis

```
int lu62_request_notification(lu62_request_notification_t *rqp);
```

Request Structure

```
typedef struct {
    char     host[MAXHOSTNAMELEN+1];             /* s */
    char     lu_name[LU62_LU_NAME_LEN+1];        /* s */
    bit32    return_code;                        /* r */
    bit32    port_id;                            /* r */
    int      port_desc;                          /* r */
} lu62_request_notification_t;
```

11.4.4 lu62_request_notification_t *Request Structure Members*

The following subsections describe the lu62_request_notification_t request structure members:

host

(supplied/optional) Specifies the TCP/IP hostname of the SunLink SNA PU2.1 9.1 server host. Hostnames are configured in the Unix network configuration file /etc/hosts, or maintained by NIS. Server is supplied as an ASCII (null-terminated) string. If host is not supplied, localhost is assumed.

`lu_name`

(supplied) Specifies the name of the logical unit in the SunLink SNA PU2.1 9.1 server with which this connection is to be associated. This name corresponds to the `LU_NAME` parameter of the LU definition. `lu_name` is supplied as an ASCII (null-terminated) string.

`return_code`

(returned) Provides an immediate response. The `return_code` variable can have one of the following values:

- `LU62_OPERATION_INCOMPLETE`
  Indicates that a request has been sent to the SunLink SNA PU2.1 9.1 server. `lu62_receive_notification` must be issued to receive the server's reply.
- `LU62_PARAMETER_CHECK`
  - `LU62_HOST_UNKNOWN`
  - `LU62_SERVER_UNKNOWN`
  - `LU62_LU_NAME_REQD`
  - `LU62_BAD_LU_NAME`

`port_id`

(returned) Specifies the port identifier assigned to the connection. All subsequent CNOS notification verbs issued to the selected LU designate this `port_id`.

`port_desc`

(returned) Specifies the file descriptor associated with the socket connection. The `port_desc` is made available for those users who want to use `select` to multiplex CNOS events with events from other devices (including other connections to the SunLink SNA PU2.1 9.1 server).

Server Response

The SunLink SNA PU2.1 9.1 server responds to the request with an eventual return, which is received using `lu62_receive_notification`.

The lu62_notification_header_t returned values are:

op_code

LU62_REQUEST_NOTIFICATION_REPLY

return_code

One of the following:

- LU62_OK
- LU62_PARAMETER_ERROR
  - LU62_UNKNOWN_LU

There is no data associated with an LU62_REQUEST_NOTIFICATION_REPLY.

See Also

lu62_receive_notification, lu62_stop_notification

11.4.5 lu62_poll_notification

lu62_poll_notification is used to test whether a notification from the SunLink SNA PU2.1 9.1 server is pending or not.

Synopsis

```
#include <sys/time.h>
int lu62_poll_notification(lu62_notification_header_t *hp,
                           struct timeval *timeout);
```

Parameters

hp

(supplied) The notification header, see lu62_receive_notification.

`timeout`

(supplied) Specifies how long to wait for a notification. If timeout is NULL, the verb waits indefinitely. If timeout is zeroed, the verb test whether a notification is pending and returns immediately with the result.

Return Value

`LU62_OK`

Indicates that a notification is pending. The program should issue `lu62_receive_notification` to receive the notification.

`LU62_ERROR`

It can be one of the following:

- `LU62_WAIT_TIMEOUT`

No notification is pending.

- `LU62_PARAMETER_ERROR`
  - `LU62_PORT_ID_UNKNOWN`
- `LU62_SERV_DCNX`

The connection to the server has failed.

See Also

`lu62_receive_notification`, `lu62_request_notification`

11.4.6 `lu62_stop_notification`

`lu62_stop_notification` is used to request that the SunLink SNA PU2.1 9.1 server stop sending CNOS notifications to the program.

If parameters are valid, `lu62_stop_notification` sends the request to the SunLink SNA PU2.1 9.1 server and sends the request on to the server. The verb then returns to the caller with a `return_code` of `LU62_OPERATION_INCOMPLETE`. The caller must issue `lu62_receive_notification` to receive the eventual return.

Synopsis

```
int lu62_stop_notification(lu62_stop_notification_t *rqp);
```

Request Structure

```
typedef struct {
    bit32           port_id;                        /* s */
    bit32           return_code;                    /* r */
} lu62_stop_notification_t
```

11.4.7 `lu62_stop_notification_t` *Request Structure Members*

The following subsections describe the `lu62_stop_notification_t` request structure members:

`port_id`

(supplied) Specifies the `port_id` of the LU connection to close. The `port_id` is returned by `lu62_request_notification`.

`return_code`

(returned) Provides an immediate response. The `return_code` variable can have one of the following values:

- `LU62_OPERATION_INCOMPLETE`

Indicates that a request was sent to the SunLink SNA PU2.1 9.1 server. `lu62_receive_notification` must be issued to receive the server's reply.

- `LU62_PARAMETER_CHECK`
  — `LU62_PORT_ID_UNKNOWN`

Server Response

The SunLink SNA PU2.1 9.1 server responds to the request with an eventual return, which is received using `lu62_receive_notification`.

The `lu62_notification_header_t` returned values are:

`op_code`

`LU62_STOP_NOTIFICATION_REPLY`

`return_code`

`LU62_OK`.

There is no data associated with an `LU62_STOP_NOTIFICATION_REPLY`.

See Also

`lu62_receive_notification`, `lu62_request_notification`

11.5 Session Control Verbs

Session control verbs allow a program to activate and deactivate LU-LU sessions. Table 11-4 summarizes the Session Control verbs.

Table 11-3 SunLink LU6.2 Session Control Verbs

Verb	**Function**
`lu62_activate_session`	Activates a session with the specified mode
`lu62_deactivate_session`	Deactivate LU-LU session

CNOS Privilege

In order to issue CNOS verbs, your program must be configured with TP `PRIVILEGE = SESSION_CONTROL`.

11.5.1 `lu62_activate_session`

`lu62_activate_session` activates a session with the specified mode name to the remote LU.

Synopsis

```
int lu62_activate_session(lu62_activate_session_t *rqp);
```

Request Structure

```
typedef struct {
    bit32       port_id;                            /* s */
    char        lu_name[LU62_LU_NAME_LEN+1];        /* s */
    char        mode_name[LU62_MODE_NAME_LEN+1];    /* s */
    bit32       return_code;                        /* r */
} lu62_activate_session_t;
```

11.5.2 `lu62_activate_session_t` *Request Structure Members*

The following subsections describe the `lu62_activate_session_t` request structure members:

`port_id`

(supplied) Specifies the `port_id` of the LU connection to use. The `port_id` is returned by `lu62_open`.

`lu_name`

(supplied) Specifies the locally known name of the partner LU to which the session is to be activated. `lu_name` is supplied as an ASCII (null-terminated) string. It corresponds to the `PTNR_LU NAME` parameter in the configuration.

`mode_name`

(supplied) Specifies the name of the mode for the session. `mode_name` is supplied as an ASCII (null-terminated) string and is translated to EBCDIC by the SunLink SNA PU2.1 9.1 server. It corresponds to the `mode_name` parameter in the configuration.

`return_code`

(returned) Specifies the result of verb execution which can be one of the following:

- LU62_OK
  - LU62_OK_AS_SPECIFIED
    Single session conwinner activated.
  - LU62_OK_AS_NEGOTIATED
    Single session conloser activated.
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - LU62_PROGRAM_NOT_PRIVILEGED
  - LU62_PORT_ID_UNKNOWN
  - LU62_LU_NAME_REQD
  - LU62_MODE_NAME_REQD
  - LU62_BAD_LU_NAME
  - LU62_BAD_MODE_NAME
- LU62_PROGRAM_STATE_CHECK
  - LU62_VERB_IN_PROGRESS
- LU62_PARAMETER_ERROR
  - LU62_UNKNOWN_PARTNER_LU
  - LU62_UNKNOWN_MODE
- LU62_ACTIVATION_FAILURE_NO_RETRY
  - LU62_ACTIVATION_FAILURE_RETRY
  - LU62_MODE_SESSION_LIMIT_EXCEEDED
  - LU62_UNRECOGNIZED_MODE_NAME

Usage Notes

1. This verb can be used to activate a single-session as a contention winner for the local or remote LU. The contention-winner for the session is established by `lu62_init_session_limit`. If the local LU is the conwinner, the session is activated. If the local LU is the conloser, the local LU bids for the session.

2. This verb can be used to activate one or both parallel-sessions for the SNASVCMG mode. The local LU is the conwinner for one session and the conloser for the other.

3. This verb can be used to activate a parallel-session as a contention winner for the local or remote LU. A conwinner or conloser session is activated by the local LU depending on the current session limit and contention-polarities established by CNOS. The local LU activates a conwinner session if:

```
active_conwinners < lu_mode_session_limit -
min_conwinners_target
```

Otherwise, the local LU bids for a session.

See Also

CNOS verbs

11.5.3 lu62_deactivate_session

`lu62_deactivate_session` deactivates the specified session.

Synopsis

```
int lu62_deactivate_session(lu62_deactivate_session_t *rqp);
```

Request Structure

```
typedef struct {
    bit32       port_id;                             /* s */
    char        session_id[LU62_SESSION_ID_LEN+1];   /* s */
    lu62_ds_type_type;                                  /* s */
    int         sense_code_supplied;                 /* s */
    bit32       sense_code;                          /* s */
    lu62_ds_return_control_e return_control;         /* s */
    bit32       return_code;                         /* r */
} lu62_deactivate_session_t;
```

11.5.4 lu62_deactivate_session_t *Request Structure Members*

The following subsections describe the lu62_deactivate_session_t request structure members:

port_id

(supplied) Specifies the port_id of the LU connection to use. The port_id is returned by lu62_open.

session_id

(supplied) Specifies the identifier of the session to be deactivated. Session identifiers are returned by lu62_display_mode.

type

(supplied) Specifies the type of deactivation:

- DS_CLEANUP
  The session is to be deactivated immediately regardless of whether a conversation is allocated or not.
- DS_NORMAL
  The session is to be deactivated normally, after the current, if any, conversation is deallocated.

sense_code_supplied

(supplied) Specifies whether a sense_code is provided for the deactivation. This parameter is ignored if type = DS_NORMAL.

- FALSE
- TRUE

sense_code

(supplied/conditional). Specifies the sense_code to be used in the deactivation. Refer to the *IBM SNA Formats* manual for valid sense codes. No error checking is performed.

return_control

(supplied) Specifies when control should be returned to the program to be one of the following:

- DS_IMMEDIATE
  Return control as soon as session deactivation has been initiated by the local LU.

Note – LU62_OPERATION_INCOMPLETE will still be returned when the processing mode is non-blocking.

- DS_DELAYED
  Return control when session deactivation is complete.

return_code

(returned) Specifies the result of verb execution, which can be one of the following:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - LU62_PROGRAM_NOT_PRIVILEGED
  - LU62_PORT_ID_UNKNOWN
  - LU62_SESSION_ID_REQD
- LU62_PROGRAM_STATE_CHECK
  - LU62_VERB_IN_PROGRESS

Usage Notes

1. Each session is assigned a unique session ID when it is activated. Session IDs exist for the duration of the session and are deleted when the session is deactivated. Use lu62_display_mode to retrieve the session IDs for active sessions.

2. If session_id specifies an unknown session, the session is assumed to have been deactivated, and return_code of LU62_OK is returned to the program.

See Also

lu62_display_mode

11.6 Display Verbs

Display verbs are used to examine the local LU's operating parameters. Table 11-4 lists the display verbs.

Table 11-4 SunLink LU6.2 Display Verbs

Verb	Function
`lu62_display_local_lu`	Returns information about the local LU
`lu62_display_mode`	Returns information about the mode
`lu62_display_remote_lu`	Returns information about the remote LU
`lu62_display_tp`	Returns information about the TP

CNOS Privilege

To issue CNOS verbs, your program must be configured with TP PRIVILEGE = DISPLAY.

Variable Length Data

The display verbs return fixed length parameters and variable length parameters. An example of a variable length parameter is the list of TP names returned by `lu62_display_local_lu`. To receive such parameters, the user must provide a data buffer. The verb returns a count of the number of entries in the list as a fixed length parameter, and then the list entries themselves are written into the user's buffer. If the user's buffer is of insufficient length to receive the data, a `return_code`, `LU62_BUFFER_TOO_SMALL`, is provided and no data is returned.

11.6.1 `lu62_display_local_lu`

`lu62_display_local_lu` returns the current values of the local LU's operating parameters.

Synopsis

```
int lu62_display_local_lu(lu62_display_local_lu_t *rqp);
```

Request Structure

Code Example 11-1

```
typedef struct {
    bit32     port_id;                                 /* s */
    bit8      *buffer;                                 /* s */
    int       length;                                /* s/r */
    bit32     return_code;                             /* r */
    char      nq_lu_name[LU62_NQ_LU_NAME_LEN+1];     /* r */
    char      lu_name[LU62_LU_NAME_LEN+1];           /* r */
    int       lu_session_limit;                      /* r */
    int       lu_session_count;                      /* r */
    int       bind_rsp_queue_capability;             /* r */
    int       security_count;                        /* r */
    int       map_name_count;                        /* r */
    int       remote_lu_name_count;                  /* r */
    int       tp_name_count;                         /* r */
lu62_display_local_lu_t;
}
```

11.6.2 `lu62_display_local_lu_t` *Request Structure Members*

The following subsections describe the `lu62_display_local_lu_t` request structure members:

`port_id`

(supplied) Specifies the `port_id` of the LU connection to use. The `port_id` is returned by `lu62_open`.

`buffer`

(supplied) Specifies the address of a user buffer to receive the variable length parameters returned by this call, see below.

`length`

(supplied / returned) On input, length specifies the length of the supplied buffer. On output, length, contains the length of the variable length parameters contained in the buffer.

`nq_lu_name`

(returned) Specifies the network qualified name of the local LU. `nq_lu_name` is returned as an ASCII (null-terminated) string. It corresponds to the `LU NQ_LU_NAME` parameter in the configuration.

`lu_name`

(returned) Specifies the locally known name of the local LU. `lu_name` is returned as an ASCII (null-terminated) string. It corresponds to the LU NAME parameter in the configuration.

`lu_session_limit`

(returned) Specifies the maximum number of LU-LU sessions supported by the local LU for all modes.

`lu_session_count`

(returned) Specifies the current number of active LU-LU sessions supported by the local LU for all modes.

`bind_rsp_queue_capability`

(returned) Specifies whether the local LU allows the remote LU to queue session activation (BIND) requests if a session cannot be activated immediately.

`security_count`

(ignored) Reserved for future use.

`map_name_count`

(ignored) Reserved for future use.

`remote_lu_name_count`

(returned) Returns the number of entries in the variable length list of `remote_lu_names`, see below.

`tp_name_count`

(returned) Returns the number of entries in the variable length list of `tp_names`, see below.

`return_code`

(returned) Specifies the result of verb execution, one of:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  - `LU62_PROGRAM_NOT_PRIVILEGED`
  - `LU62_PORT_ID_UNKNOWN`
  - `LU62_NQ_LU_NAME_REQD`
  - `LU62_BUFFER_TOO_SMALL`
- `LU62_PROGRAM_STATE_CHECK`
  - `LU62_VERB_IN_PROGRESS`
- `LU62_PARAMETER_ERROR`
  - `LU62_UNKNOWN_LU.`

Variable Length Parameters

Variable length parameters are written into the buffer in the following format:

```
    char remote_lu_names [remote_lu_name_count-1]
[LU62_NQ_LU_NAME_LEN+1];
    char tp_names [tp_name_count-1][LU62_TP_NAME_LEN+1];
```

where:
`remote_lu_names`

The list of the network qualified remote LU names defined at the local LU. Remote LU names are returned as ASCII (null-terminated) strings. The name corresponds to the `NQ_LU_NAME` of the partner LU parameter in the configuration.

`tp_names`

The list of TP names defined at the local LU. TP names are returned as ASCII (null-terminated) strings. The name corresponds to the `TP_NAME` parameter in the configuration.

Usage Notes

See Also

Refer to the *SunLink SNA PU2.1 9.1 Server Configuration and Administration Manual* for more information regarding the configuration parameters.

11.6.3 `lu62_display_mode`

`lu62_display_mode` returns the current values of the (LU, mode) operating parameters.

Synopsis

```
int lu62_display_mode(lu62_display_mode_t *rqp);
```

Request Structure

```
typedef struct {
    bit32    port_id;                              /* s */
    char     nq_lu_name[LU62_NQ_LU_NAME_LEN+1];    /* s */
    char     mode_name[LU62_MODE_NAME_LEN+1];      /* s */
    bit8     *buffer;                              /* s */
    int      length;                               /* s/r */
    bit32    return_code;                          /* r */
    char     lu_name[LU62_LU_NAME_LEN+1];          /* r */
    int      send_max_ru_size_lb;                    /* r */
    int      send_max_ru_size_ub;                    /* r */
    int      recv_max_ru_size_lb;                    /* r */
    int      recv_max_ru_size_ub;                    /* r */
    lu62_single_session_reinit_e single_session_reinit; /* r */
    lu62_session_level_crypto_e session_level_crypto;/* r */
    int      conwinner_autoactivate_limit;           /* r */
    int      local_max_session_limit;                /* r */
    int      lu_mode_session_limit;                  /* r */
    int      min_conwinners;                         /* r */
    int      min_conlosers;                          /* r */
    int      termination_count;                      /* r */
    int      drain_local_lu;                         /* r */
    int      drain_remote_lu;                        /* r */
    int      lu_mode_session_count;                  /* r */
    int      conwinners_session_count;               /* r */
    int      conlosers_session_count;                /* r */
    int      conv_group_count;                       /* r */
    int      preferred_received_ru_size;             /* r */
    int      preferred_send_ru_size;                 /* r */
    char     sess_deact_tp_name[LU62_TP_NAME_LEN+1]  /* r */
    char     unique_session_name                         /* r */
               [LU62_UNIQUE_SESSION_NAME_LEN+1];
} lu62_display_mode_t;
```

11.6.4 `lu62_display_mode_t` *Request Structure Members*

The following subsections describe the `lu62_display_mode_t` request structure members:

`port_id`

(supplied) Specifies the `port_id` of the LU connection to use. The `port_id` is returned by `lu62_open`.

`nq_lu_name`

(supplied) Specifies the network qualified name of the partner LU. `nq_lu_name` is supplied as an ASCII (null-terminated) string. It corresponds to the `PTNR_LU NQ_LU_NAME` parameter in the configuration.

`mode_name`

(supplied) Specifies the name of the mode. `mode_name` is supplied as an ASCII (null-terminated) string. It corresponds to the `MODE NAME` parameter in the configuration.

`buffer`

(supplied) Specifies the address of a user buffer to receive the variable length parameters returned by this call, see below.

`length`

(Supplied / returned.) On input, `length` specifies the length of the supplied buffer. On output, `length` contains the length of the variable length parameters contained in the buffer.

`lu_name`

(returned) Specifies the locally known name of the partner LU. `lu_name` is returned as an ASCII (null-terminated) string. It corresponds to the `PTNR_LU NAME` parameter in the configuration.

`send_max_ru_size_lb`

(returned) Specifies the lower bound of the maximum send RU size.

`send_max_ru_size_ub`

(returned) Specifies the upper bound of the maximum send RU size.

`recv_max_ru_size_lb`

(returned) Specifies the lower bound of the maximum receive RU size.

`recv_max_ru_size_ub`

(returned) Specifies the upper bound of the maximum receive RU size.

`single_session_reinit`

(ignored) Reserved for future use.

`session_level_crypto`

(ignored) Reserved for future use.

`conwinner_autoactivate_limit`

(returned) Specifies the local LU's automatic activation limit on the number of conwinner sessions.

`local_max_session_limit`

(returned) Specifies the maximum number of sessions for all modes supported by the local LU.

`lu_mode_session_limit`

(returned) Specifies the maximum number of sessions to be allowed between the local (source) LU and the partner LU, for the specified `mode_name`.

`min_conwinners`

(returned) Specifies the minimum number of sessions with the given `mode_name` for which the local LU is the contention winner.

`min_conlosers`

(returned) Specifies the minimum number of sessions with the given `mode_name` for which the partner LU is the contention winner.

`termination_count`

(returned) Specifies the number of sessions that the local LU is responsible for deactivating as a result of CNOS processing.

`drain_local_lu`

(returned) Specifies whether or not the local LU is allowed to drain its allocation requests following a CNOS reset session limit command.

`drain_remote_lu`

(returned) Specifies whether or not the remote LU is allowed to drain its allocation requests following a CNOS reset session limit command.

`lu_mode_session_count`

(returned) Specifies the current (LU, mode) session count. The session identifiers assigned to the active sessions are returned as variable length parameters, see below.

`conwinners_session_count`

(returned) Specifies the number of active sessions for which the local LU is the contention-winner.

`conwinners_session_count`

(returned) Specifies the number of active sessions for which the local LU is the loser.

`conv_grp_count`

(ignored) Reserved for future use.

`preferred_received_ru_size`

(returned) Specifies the preferred receive RU size.

`preferred_send_ru_size`

(returned) Specifies the preferred send RU size.

`sess_deact_tp_name`

(ignored) Reserved for future use.

`unique_session_name`

(Returned) Specifies the `unique_session_name`, if specified.

`return_code`

(returned) Specifies the result of verb execution from one of the following:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  - `LU62_PROGRAM_NOT_PRIVILEGED`
  - `LU62_PORT_ID_UNKNOWN`
  - `LU62_NQ_LU_NAME_REQD`
  - `LU62_MODE_NAME_REQD`
  - `LU62_BUFFER_TOO_SMALL`
- `LU62_PROGRAM_STATE_CHECK`
  - `LU62_VERB_IN_PROGRESS`
- `LU62_PARAMETER_ERROR`
  - `LU62_UNKNOWN_PARTNER_LU`
  - `LU62_UNKNOWN_MODE`

Variable Length Parameters

Variable length parameters are written into the buffer in the following format:

```
char session_ids [lu_mode_session_count-
1][LU62_SESSION_ID_LEN+1];
```

where:
`session_ids`

The session identifiers assigned to the active sessions. A session identifier is returned as a (null-terminated) ASCII-hex string.

See Also

Refer to the *SunLink SNA PU2.1 9.1 Server Configuration and Administration Manual* for more information regarding the configuration parameters.

11.7 lu62_display_remote_lu

`lu62_display_remote_lu` returns the current values of the parameters that control the operation of the local LU in conjunction with a remote LU.

Synopsis

```
int lu62_display_remote_lu(lu62_display_remote_lu_t *rqp);
```

Request Structure

```
typedef struct {
    bit32       port_id;                            /* s */
    char        nq_lu_name[LU62_NQ_LU_NAME_LEN+1];  /* s */
    bit8        *buffer;                            /* s */
    int         length;                            /* s/r */
    bit32       return_code;                        /* r */
    char        lu_name[LU62_LU_NAME_LEN+1];        /* r */
    char        ui_lu_name[LU62_LU_NAME_LEN+1];     /* r */
    lu62_initiate_type_e initiate_type;             /* r */
    int         parallel_session_support;           /* r */
    int         cnos_support;                       /* r */
    lu62_security_accept_esecurity_accept_local_lu; /* r */
    lu62_security_accept_esecurity_accept_remote_lu /* r */
    int         mode_name_count;                    /* r */
} lu62_display_remote_lu_t;
```

11.7.1 `lu62_display_remote_lu_t` *Request Structure Member*

The following subsections describe the `lu62_display_remote_lu_t` request structure members:

`port_id`

(supplied) Specifies the `port_id` of the LU connection to use. The `port_id` is returned by `lu62_open`.

`nq_lu_name`

(supplied) Specifies the network qualified name of the partner LU. `nq_lu_name` is supplied as an ASCII (null-terminated) string. It corresponds to the `PTNR_LU NQ_LU_NAME` parameter in the configuration.

`buffer`

(supplied) Specifies the address of a user buffer to receive the variable length parameters returned by this call, see below.

`length`

(supplied / returned) On input, `length` specifies the length of the supplied buffer. On output, `length` contains the length of the variable length parameters contained in the buffer.

`lu_name`

(returned) Specifies the locally known name of the partner LU. `lu_name` is returned as an ASCII (null-terminated) string. It corresponds to the `PTNR_LU NAME` parameter in the configuration.

`ui_lu_name`

(returned) Specifies the uninterpreted name of the partner LU. `ui_lu_name` is returned as an ASCII (null-terminated) string. It corresponds to the `PTNR_LU UI_LU_NAME` parameter in the configuration.

`initiate_type`

(returned) Specifies the session-initiation type for the remote LU from one of the following:

- INITIATE_ONLY
- INITIATE_OR_QUEUE

parallel_session_support

(returned) Specifies whether or not the remote LU supports parallel sessions with the local LU.

cnos_support

(returned) Specifies whether or not the remote LU supports CNOS.

security_accept_local_lu

(returned) Specifies the level of access security information the local LU will accept on allocation requests from the remote LU as one of the following:

- SA_NONE
- SA_CONVERSATION
- SA_ALREADY_VERIFIED

security_accept_remote_lu

(returned) Specifies the level of access security information the remote LU will accept on allocation requests from the local LU as one of the following:

- SA_NONE
- SA_CONVERSATION
- SA_ALREADY_VERIFIED

mode_name_count

(returned) Returns the number of entries in the variable length list of mode_names, see below.

return_code

(returned) Specifies the result of verb execution of one of the following:

- LU62_OK
- LU62_OPERATION_INCOMPLETE
- LU62_PARAMETER_CHECK
  - LU62_PROGRAM_NOT_PRIVILEGED

  - LU62_PORT_ID_UNKNOWN
  - LU62_NQ_LU_NAME_REQD
  - LU62_BUFFER_TOO_SMALL
- LU62_PROGRAM_STATE_CHECK
  - LU62_VERB_IN_PROGRESS
- LU62_PARAMETER_ERROR
  - LU62_UNKNOWN_PARTNER_LU

Variable Length Parameters

Variable length parameters are written into the buffer in the following format:

```
char mode_names [mode_name_count1][LU62_MODE_LEN+1];
```

where:

`mode_names`

The list of mode names defined at the local LU for sessions with the remote LU. Mode names are returned as ASCII (null-terminated) strings. The name corresponds to the `MODE NAME` parameter in the configuration.

See Also

Refer to the *SunLink SNA PU2.1 9.1 Server Configuration and Administration Manual* for more information regarding the configuration parameters.

11.7.2 lu62_display_tp

`lu62_display_tp` returns the current values of the parameters that control the operation of the local LU in conjunction with a transaction program.

Synopsis

```
int lu62_display_tp(lu62_display_tp_t *rqp);
```

Request Structure

```
typedef struct {
    bit32       port_id;                          /* s */
    char        nq_lu_name[LU62_NQ_LU_NAME_LEN+1];  /* s */
    char        tp_name[LU62_TP_NAME_LEN+1];      /* s */
    bit8        *buffer;                          /* s */
    int         length;                           /* s/r */
    bit32       return_code;                      /* r */
    lu62_tp_status_estatus;                       /* r */
    int         basic_support;                    /* r */
    int         mapped_support;                   /* r */
    int         sync_level_none;                  /* r */
    int         sync_level_confirm;               /* r */
    int         sync_level_syncpt;                /* r */
    lu62_security_required_e security_required;   /* r */
    int         security_access_count;            /* r */
    lu62_pip_e  pip;                              /* r */
    int         pip_count;                        /* r */
    int         data_mapping;                     /* r */
    lu62_fmh_data_e fmh_data;                     /* r */
    int         cnos_privilege;                   /* r */
    int         session_control_privilege;        /* r */
    int         define_privilege;                 /* r */
    int         display_privilege;                /* r */
    int         allocate_svc_tp_privilege;        /* r */
    int         instance_limit;                   /* r */
    int         instance_count;                   /* r */
} lu62_display_tp_t;
```

11.7.3 `lu62_display_tp_t` *Request Structure Members*

The following subsections describe the `lu62_display_tp_t` request structure members:

`port_id`

(supplied) Specifies the `port_id` of the LU connection to use. The `port_id` is returned by `lu62_open`.

nq_lu_name

(supplied / optional) By default, lu62_display_tp applies to the local LU as implied by port_id. For backward compatibility, this verb will allow the network qualified name of a local LU to be supplied. nq_lu_name is supplied as an ASCII (null-terminated) string. It corresponds to the LU NQ_LU_NAME parameter in the configuration.

tp_name

(supplied) Specifies the name of the TP. tp_name is supplied as an ASCII (null-terminated) string. It corresponds to the TP NAME parameter in the configuration.

buffer

(supplied) Specifies the address of a user buffer to receive the variable length parameters returned by this call, see below.

length

(supplied / returned) On input, length specifies the length of the supplied buffer. On output, length contains the length of the variable length parameters contained in the buffer.

status

(returned) Specifies the status for starting execution of the transaction program from one of the following:

- TP_ENABLED
- TP_TEMP_DISABLED
- TP_PERM_DISABLED

basic_support

(returned) Specifies whether or not the TP supports basic conversations.

mapped_support

(returned) Specifies whether or not the TP supports mapped conversations.

`sync_level_none`

(returned) Specifies whether or not the TP supports conversations with no synchronization level.

`sync_level_confirm`

(returned) Specifies whether or not the TP supports conversations with a synchronization level of CONFIRM.

`sync_level_syncpt`

(returned) Specifies whether or not the TP supports conversations with a synchronization level of SYNCPT.

`security_required`

(returned) Specifies the type of security verification that is required on incoming allocation requests to the TP as one of the following:

- `SE_NONE`
- `SE_CONVERSATION`
- `SA_ACCESS`

`security_access_count`

(ignored) Reserved for future use.

`pip`

(returned) Specifies whether PIP data is required by the TP and, if so, whether the LU or the TP verify the number of PIP subfields. `pip` may be one of the following:

- `PIP_NO`
- `PIP_NO_LU_VERIFICATION`
- `PIP_YES`

`pip_count`

(returned) If pip = `PIP_YES`, specifies the number of PIP subfields required on incoming allocation requests.

`data_mapping`

(ignored) Reserved for future use.

`fmh_data`

(returned) Specifies whether or not FMH data support is provided to the TP for one of the following:

- `FMH_NO`
- `FMH_YES`

`cnos_privilege`

(returned) Specifies whether or not the TP has privilege to issue CNOS verbs.

`session_control_privilege`

(returned) Specifies whether or not the TP has privilege to issue session control verbs.

`define_privilege`

(ignored) Reserved for future use.

`display_privilege`

(returned) Specifies whether or not the TP has privilege to issue display verbs.

`allocate_svc_tp_privilege`

(ignored) Reserved for future use.

`instance_limit`

(returned) Specifies the number of instances of the TP that can be active concurrently.

`instance_count`

(returned) Specifies the current number of active TP instances.

`return_code`

(returned) Specifies the result of verb execution as one of the following:

- `LU62_OK`
- `LU62_OPERATION_INCOMPLETE`
- `LU62_PARAMETER_CHECK`
  - `LU62_PROGRAM_NOT_PRIVILEGED`
  - `LU62_PORT_ID_UNKNOWN`
  - `LU62_NQ_LU_NAME_REQD`
  - `LU62_TP_NAME_REQD`
  - `LU62_BUFFER_TOO_SMALL`
- `LU62_PROGRAM_STATE_CHECK`
  - `LU62_VERB_IN_PROGRESS`
- `LU62_PARAMETER_ERROR`
  - `LU62_UNKNOWN_PARTNER_LU`
  - `LU62_UNKNOWN_TP`

Variable Length Parameters

Reserved for future use.

See Also

Refer to the *SunLink SNA PU2.1 9.1 Server Configuration and Administration Manual* for more information regarding the configuration parameters.

SunLink LU6.2 Utilities 12

This chapter documents the various utilities supplied with the SunLink LU6.2 API. These utilities are listed in Table 12-1. Detailed man pages follow.

Table 12-1 SunLink LU6.2 Utilities

Verb Function	
Tracing	
`*lu62_trace`	Outputs a program trace
`*lu62_set_trace_flag`	Sets program trace flags
`*lu62_get_trace_flag`	Gets program trace flags
`*lu62_dump_buffer`	Hex dumps a buffer
Character Conversion	
`*conv_ascii_to_ebcdic`	Converts a character from ASCII to EBCDIC (640)
`*conv_ebcdic_to_ascii`	Converts a character from EBCDIC (640) to ASCII
`*b_asc_to_ebc`	Converts a buffer from ASCII to EBCDIC (640)
`*b_ebc_to_asc`	Converts a buffer from EBCDIC (640) to ASCII
`*str_asc_to_ebc`	Converts a string from ASCII to EBCDIC (640)

Table 12-1 SunLink LU6.2 Utilities *(Continued)*

Verb Function	
*str_ebc_to_asc	Converts a string from EBCDIC(640) to ASCII
*strn_asc_to_ebc	Converts a max length string from ASCII to EBCDIC(640)
*strn_ebc_to_asc	Converts a max length string from EBCDIC(640) to ASCII

12.1 *lu62_trace

lu62_trace is used to output trace information to a file. Traces are only output if the result of a bitwise AND between the specified trace *type* and the external variable lu62_trace_flag is non-zero. lu62_trace_flag is an external *bit32* variable (unsigned), which may be accessed by the routines lu62_set_trace_flag and lu62_get_trace_flag. Each bit in the lu62_trace_flag word represents a different trace type.

Trace output is written to a file sunlu62l_$$ in the process working directory (where ," indicates the current process id). When 1000 traces are accumulated, the file is saved as sunlu62l_$$.1, and truncated.

Traces are used extensively by the API. Unused trace types are available for use by user programs.

```
,” trace flag access ,"
extern bit32 lu62_trace_flag;
void lu62_set_trace_flag(bit32 flag);
bit32 lu62_get_trace_flag();

/* trace statement */
void lu62_trace(unsigned type,
                char     *caller,
                char     *statement,
                int      length,
                char     *buffer,
                int      format);
```

Synopsis

Parameters

`type`

(supplied) Specifies the required trace type. Traces are only output if the result of a bitwise AND between `type` and the external variable `lu62_trace_flag` is non-zero.

`caller`

(supplied/optional) The name of the calling routine, supplied as a null-terminated string.

`statement`

(supplied/optional) A trace header, supplied as a null-terminated string.

`length`

(supplied/optional) The length of the trace buffer.

`buffer`

(supplied/optional) This is the trace buffer.

`format`

(supplied/optional) Specifies how the trace buffer is to be formatted in the trace output as one of the following:

- `STRING:` The buffer contains a null-terminated string for output. The buffer length is ignored.
- `ASCII_DATA:` The buffer is dumped as hex data. Buffer contents are interpreted as ASCII characters and printable data characters are output.
- `EBCDIC_DATA:`The buffer is dumped as hex data. Buffer contents are interpreted as EBCDIC characters and printable data characters are output.
- `NO_FMT_DATA:` The buffer is dumped as hex data.

Example

The use of these parameters is shown in the example below. The following trace statement is issued by `tpi_send_msg` (an internal routine), to trace the transmission of buffers to the SunLink SNA PU2.1 9.1 server:

```
    lu62_trace(LU62_API_BUFS, “tpi_send_msg”, sym_conv_id(cp),
                  length, send_buffer, ASCII_DATA);
```

If the result of (`lu62_trace_flag & LU62_API_BUFS`) is non-zero, an example of the resulting trace is as follows:

```
<-- timestamp -><- caller -><  statement >

05/25/95 15:31:27 tpi_send_msg: LUA: 190240
000000000002e720 0000000000000001 0000000000000020     .......
...............
0000000000000000 0000000000000000 0000006700000000
...................g....
4c55410000000000 00434f5052000000 00000000
LUA......COPR.......

<----------------- hex dump of buffer --------------->< --
interpreted data -->
```

12.2 *`lu62_dump_buffer`

`lu62_dump_buffer` is used by `lu62_trace` to trace buffers; it is also a useful debugging routine.

Synopsis

```
void lu62_dump_buffer(FILE *file,
                      int   length,
                      char *buffer,
                      int   format);
```

Parameters

`file`

(supplied) Specifies the required output file.

`length`

(supplied) The length of the buffer.

`buffer`

(supplied) The buffer.

`format`

(supplied) Specifies how the buffer is to be formatted in the output using one of the following:

- `ASCII_DATA`: The buffer is dumped as hex data. Buffer contents are interpreted as ASCII characters and printable data characters are output.
- `EBCDIC_DATA`: The buffer is dumped as hex data. Buffer contents are interpreted as EBCDIC characters and printable data characters are output.
- `NO_FMT_DATA`: The buffer is dumped as hex data.

12.3 Character Conversion Routines

Character conversion routines convert between ASCII and EBCDIC (00640) character sets.

The source code for the conversion routines `lu62_convert.c` is supplied with the distribution. You are free to modify the source as required. Ensure that your modified code is linked with your program before the SunLink LU6.2 library, or rebuild the library to include your file.

Synopsis

```
,” single character conversion ,"
unsigned conv_ascii_to_ebcdic(unsigned asc);
unsigned conv_ebcdic_to_ascii(unsigned ebc);

/* buffer conversion - convert all characters including ‘’ */
unsigned char *b_asc_to_ebc(unsigned char *asc_str,
                            unsigned char *ebc_str,
                            int len);

unsigned char *b_ebc_to_asc(unsigned char *ebc_str,
                            unsigned char *asc_str,
                            int len);

/* string conversion - return converted string */
unsigned char *str_asc_to_ebc(unsigned char *asc_str,
                              unsigned char *ebc_str);

unsigned char *str_ebc_to_asc(unsigned char *ebc_str,
                              unsigned char *asc_str);

/* string length conversion - return converted string */
unsigned char *strn_asc_to_ebc(unsigned char *asc_str,
                               unsigned char *ebc_str,
                               int len);

unsigned char *strn_ebc_to_asc(unsigned char *ebc_str,
                               unsigned char *asc_str,
                               int len);
```

SunLinkLU6.2 Return Codes A

This appendix documents the return codes that are passed to the program at the completion of a verb execution. The following return codes are described:

- Conversation return codes
- Control Operator return codes
- Product specific return codes

Most verbs support a `return_code` parameter. The return code can be used to determine the result of verb execution results and any state changes that may have occurred on the specified conversation. On some verbs, the return code is not the only source of verb-execution information. In particular, on the `lu62_receive` verbs, the `what_received` parameter should also be checked.

Some of the return codes indicate the results of the local processing of a verb. These return codes are returned on the verb that invoked the local processing. Other return codes indicate results of processing invoked at the remote end of the conversation. Depending on the verb, these return codes can be returned on the verb that invoked the remote processing or on a subsequent verb. Still other return codes report events that originate at the remote end of the conversation. In all cases, only one code is returned at a time.

Some of the return codes associated with the allocation of a conversation have the suffix RETRY or NO_RETRY in their names.

- RETRY means that the condition indicated by the return code may not be permanent, and the program can try to allocate the conversation again. Whether or not the retry attempt succeeds depends on the duration of the condition. In general, the program should limit the number of times it attempts to retry without success.
- NO_RETRY means that the condition is probably permanent. In general, the program should not attempt to allocate the conversation again until the condition is corrected.

In the descriptions that follow, the SunLink P2P LU6.2 9.1 verb names are used rather than the *IBM SNA Transaction Programmer's Reference Manual for LU Type 6.2* names.

A.1 Implementing Return Codes and Subcodes

Return codes consist of a primary return code and, in some cases, a qualifying sub-code. SunLink LU6.2 implements the return code as a 32-bit unsigned integer (bit32). The top 16 bits contain the primary code, the lower 16 bits contain the subcode. To check for a particular primary code, you must mask off the lower 16 bits, as follows:

```
if ((return_code & 0xFFFF0000) ,“ LU62_ALLOCATION_ERROR)
```

The SunLink LU6.2 include file, `sunlu62.h`, located in Appendix C, contains `#defines` for all return codes. The return codes include the architected return codes specified in the *IBM SNA Transaction Programmer's Reference Manual for LU Type 6.2*, and SunLink LU6.2 product-specific return codes and subcodes.

A.2 Conversation Return Codes and Subcodes

The return codes shown below are listed alphabetically, and each description includes the following:

- The meaning of the return code
- The origin of the condition indicated by the return code
- When the return code is reported to the program
- The state of the conversation when control is returned to the program

The individual verb descriptions in Chapter 8, Chapter 9, and Chapter 10 list the return code values that are valid for each verb.

A.2.1 LU62_ALLOCATION_ERROR

The local program issued an `lu62_(mc_)allocate` verb and allocation of the specified conversation could not be completed. When this return code is returned to the program, the conversation is in Deallocate state. The following subcodes identify the specific error.

A.2.1.1 LU62_ALLOCATE_FAILURE_NO_RETRY

The conversation cannot be allocated on a session because of a condition that is not temporary. When this return code is returned to the program, the conversation is in Reset state. For example, the session to be used for the conversation cannot be activated because the current session limit for the specified LU-name and mode-name pair is 0, or because of a system definition error or a session activation protocol error. This return code is also returned when the session is deactivated because of a session protocol error before the conversation can be allocated. The program should not retry the allocation request until the condition is corrected. This return code is returned on the `lu62_(mc_)allocate` verb when the program specifies (by means of the `return_control` parameter) that the local LU is to allocate a session before returning control to the program; otherwise, it is returned on a subsequent verb.

A.2.1.2 LU62_ALLOCATE_FAILURE_RETRY

The conversation cannot be allocated on a session because of a condition that may be temporary. When this return code is returned to the program, the conversation is in Reset state. For example, the session to be used for the conversation cannot be activated because of a temporary lack of resources at the local LU or remote LU. This return code is also returned if the session is deactivated because of a session outage before the conversation can be allocated. The program can retry the allocation request. This return code is returned on the `lu62_(mc_)allocate` verb when the program specifies (by means of the `return_control` parameter) that the local LU is to allocate a session before returning control to the program. Otherwise, it is returned on a subsequent verb.

A.2.1.3 LU62_CONVERSATION_TYPE_MISMATCH

The remote LU rejected the allocation request because the local program issued an `lu62_mc_allocate` or `lu62_allocate` verb and the remote program does not support the respective mapped or basic conversation protocol boundary, or the local program issued an `lu62_mc_allocate` verb and the remote LU does not support mapped conversations. This return code is returned on a verb subsequent to the `lu62_(mc_)allocate.`

A.2.2 LU62_PIP_NOT_SPECIFIED_CORRECTLY

The remote LU rejected the allocation request because the remote program requires one or more program initialization parameter (PIP) variable. SunLink LU6.2 does not support PIP data. This return code is returned on a verb subsequent to the `lu62_(mc_)allocate`.

A.2.2.1 LU62_SECURITY_NOT_VALID

The remote LU rejected the allocation request because the access security information (with the `security` parameters) is invalid. This return code is returned on a verb subsequent to the `lu62_(mc_)allocate`.

A.2.3 LU62_SYNC_LEVEL_NOT_SUPPORTED_PGM

The remote LU rejected the allocation request because the local program specified a synchronization level (with the `sync_level` parameter) that the remote program does not support. This return code is returned on a verb subsequent to the `lu62_(mc_)allocate`.

A.2.4 LU62_TPN_NOT_RECOGNIZED

The remote LU rejected the allocation request because the local program specified a remote program name (with the `remote_tp_name` parameter) that the remote LU does not recognize. This return code is returned on a verb subsequent to the `lu62_(mc_)allocate`.

A.2.5 LU62_TP_NOT_AVAILABLE_NO_RETRY

The remote LU rejected the allocation request because the local program specified a remote program that the remote LU recognizes but cannot start. The condition is not temporary, and the program should not retry the allocation request. This return code is returned on a verb subsequent to the `lu62_(mc_)allocate`.

A.2.6 LU62_TP_NOT_AVAILABLE_RETRY

The remote LU rejected the allocation request because the local program specified a remote program that the remote LU recognizes but currently cannot start. The condition may be temporary and the program can retry the allocation request. This return code is returned on a verb subsequent to the `lu62_(mc_)allocate`.

A.2.7 LU62_DEALLOCATE_ABEND

The remote program issued an `lu62_mc_deallocate` verb with type set to DA_ABEND, or the remote LU has done so because of a remote program ABEND condition. If the conversation for the remote program was in Receive state when the verb was issued, information sent by the local program and not yet received by the remote program is purged. This return code is reported to the local program on a verb the program issues in Send or Receive state. The conversation is in Deallocate state.

A.2.8 LU62_DEALLOCATE_ABEND_PROG

The remote program issued an `lu62_deallocate` verb with type set to DA_ABEND_PROG, or the remote LU has done so because of a remote program ABEND condition. If the conversation for the remote program was in Receive state when the verb was issued, information sent by the local program and not yet received by the remote program is purged. This return code is reported to the local program on a verb the program issues in Send or Receive state. The conversation is in Deallocate state.

A.2.9 LU62_DEALLOCATE_ABEND_SVC

The remote program issued an `lu62_deallocate` verb with type set to `DA_ABEND_SVC`, or the remote LU has done so because of a remote program ABEND condition. If the conversation for the remote program was in Receive state when the verb was issued, information sent by the local program and not yet received by the remote program is purged. This return code is reported to the local program on a verb the program issues in Send or Receive state. The conversation is in Deallocate state.

This return code is typically generated by the remote LU, while `LU62_DEALLOCATE_ABEND_PROG` is generated by the remote transaction program. Thus, system errors may be distinguished from program errors.

A.2.10 LU62_DEALLOCATE_ABEND_TIMER

The remote program issued an `lu62_deallocate` verb with type set to `DA_ABEND_TIMER`. If the conversation for the remote program was in Receive state when the verb was issued, information sent by the local program and not yet received by the remote program is purged. This return code is reported to the local program on a verb the program issues in Send or Receive state. The conversation is in Deallocate state.

A.2.11 LU62_DEALLOCATE_NORMAL

The remote program issued an `lu62_mc_deallocate` or `lu62_deallocate` verb specifying the type parameter as `DA_SYNC_LEVEL` or `DA_FLUSH`. If type is `DA_SYNC_LEVEL`, the conversation sync-level is `SYNC_LEVEL_NONE`. This return code is reported to the local program on a verb the program issues for a conversation in Receive state. The conversation is in Deallocate state.

A.2.12 LU62_FMH_DATA_NOT_SUPPORTED

The remote program issued an `lu62_mc_send_data` verb specifying that the data record contains FM headers (by means of the `fmh_data` parameter), and that either the remote LU or remote program does not support FM header data. This return code is reported on a subsequent verb. All information sent by the local program on the `lu62_mc_send_data` verb and subsequent verbs issued prior to the reporting of the `LU62_FMH_DATA_NOT_SUPPORTED` return code is purged. The conversation is in Send state.

A.2.13 LU62_OK

The verb issued by the local program executed successfully (that is, the function defined for the verb, up to the point at which control is returned to the program, was performed as specified). The state of the conversation is as defined for the verb.

`lu62_(mc_)test` (when the test type is `TEST_POSTED`) and `lu62_wait` verbs return a subcode to provide additional information:

`LU62_OK_DATA`—Indicates that data is available for the program to receive.

`LU62_OK_NOT_DATA`—Indicates that information other than data is available for the program to receive.

A.2.14 LU62_PARAMETER_ERROR

The local program issued a verb specifying a parameter containing an invalid argument. The source of the argument is considered to be outside the program definition, such as an LU name supplied by a system administrator and used as an argument to `lu62_(mc_)allocate`. This return code is returned on the verb specifying the invalid argument. The state of the conversation remains unchanged.

A.2.15 LU62_POSTING_NOT_ACTIVE

This return code indicates that the local program issued `lu62_(mc_)test` (when the test type is `TEST_POSTED`) and posting is not active for the specified conversation; or the program issued `lu62_wait` and posting is not active for any of the specified conversations.

A.2.16 LU62_PROGRAM_ERROR_NO_TRUNC

One of the following has occurred:

- The remote program issued an `lu62_mc_send_error` verb and the conversation for the remote program was in Send state. No truncation occurs at the mapped conversation protocol boundary. This return code is reported to the local program on an `lu62_mc_receive` verb the program type issues before receiving any data records or after receiving one or more data records.

- The remote program issued an `lu62_send_error` verb with type set to `PROG`, the conversation for the remote program was in Send state, and the verb did not truncate a logical record. No truncation occurs at the basic conversation protocol boundary when the program issues `lu62_send_error` before sending any logical records or after sending a complete logical record. This return code is reported to the local program on an `lu62_receive` verb the program issues before receiving any logical records or after receiving one or more complete logical records.

The conversation remains in Receive state.

A.2.17 LU62_PROGRAM_ERROR_PURGING

The remote program issued an `lu62_mc_send_error` verb or it issued an `lu62_send_error` verb with type set to PROG, and the conversation for the remote program was in Receive or Confirm state. The verb may have caused information to be purged. Purging occurs when a program issues `lu62_send_error` for a conversation in Receive before receiving all the information sent by its partner program (all of the information sent before reporting the `LU62_PROGRAM_ERROR_PURGING` return code to the partner program). The purging can occur at the local LU, remote LU, or both. No purging occurs when a program issues the verb for a conversation in Confirm state, or in Receive state after receiving all the information sent by its partner program.

This return code is normally reported to the local program on a verb the program issues after sending some information to the remote program. However, the return code can be reported on a verb the program issues before sending any information, depending on the verb and when it is issued. The conversation remains in Receive state.

A.2.18 LU62_PROGRAM_ERROR_TRUNC

The remote program issued an `lu62_send_error` verb with type set to `PROG`, the conversation for the remote program was in Send state, and the verb truncated a logical record. Truncation occurs at the basic conversation protocol boundary when a program begins sending a logical record and then issues `lu62_send_error` before sending the complete logical record. This return code is reported to the local program on an `lu62_receive` verb the program issues after receiving the truncated logical record. The conversation remains in Receive state.

A.2.19 LU62_PROGRAM_PARAMETER_CHECK

The local program issued a verb in which a programming error has been found in one or more of the parameters. The source of the argument is considered to be inside the program definition (under the control of the local program). This return code may be caused by the program specifying an incorrect parameter value, or a parameter value that is inconsistent with the other parameters or conversation characteristics such as conversation type or sync-level. The program should not examine any other returned variables associated with the verb as nothing is placed in the variables. The state of the conversation remains unchanged.

A.2.20 LU62_PROGRAM_STATE_CHECK

The local program issued a verb for a conversation in a state that was not valid for the verb. The program should not examine any other returned variables associated with the verb as nothing is placed in the variables. The state of the conversation remains unchanged.

A.2.21 LU62_RESOURCE_FAILURE_NO_RETRY

This return code indicates that a failure occurred that caused the conversation to be prematurely terminated. For example, the session being used for the conversation was deactivated because of a session protocol error, or the conversation was deallocated because of a protocol error between the mapped conversation components of the LUs. The condition is not temporary, and the program should not retry until the condition is corrected. This return code can be reported to the local program on a verb it issues for a conversation in any state other than Reset or Deallocate. The conversation is in Deallocate state.

A.2.22 LU62_RESOURCE_FAILURE_RETRY

This return code indicates that a failure occurred that caused the conversation to be prematurely terminated. For example, the session being used for the conversation was deactivated because of a session outage such as a line failure or a modem failure. The condition may be temporary, and the program can retry the transaction. This return code can be reported to the local program on a verb it issues for a conversation in any state other than Reset or Deallocate. The conversation is in Deallocate state.

A.2.22.1 LU62_SVC_ERROR_NO_TRUNC

The remote program issued an `lu62_send_error` verb with type set to `PROG_SVC`. Otherwise, these return codes, as they apply to the basic conversation protocol boundary, have the same meaning as their `LU62_PROGRAM_ERROR` equivalents. The conversation is in Receive state.

A.2.22.2 LU62_SVC_ERROR_PURGING

Refer to Section A.2.22.1, "LU62_SVC_ERROR_NO_TRUNC."

A.2.22.3 LU62_SVC_ERROR_TRUNC

Refer to Section A.2.22.1, "LU62_SVC_ERROR_NO_TRUNC."

A.2.23 LU62_UNSUCCESSFUL

The verb issued by the local program did not execute successfully. This return code is returned on the unsuccessful verb. The state of the conversation remains unchanged.

A.3 Control Operator Return Codes and Subcodes

The return codes reported by the Control Operator verbs are described below. Each description includes the meaning of the return code and the origin of the condition indicated by the return code. The individual verb descriptions in Chapter 11, "Control Operator Verbs," list the return code values that are valid for each verb.

A.3.1 LU62_ACTIVATION_FAILURE_NO_RETRY

The `lu62_activate_session` verb failed to activate the session because of a condition that is not temporary. For example, the session cannot be activated because the current session limit for the specified `lu_name and mode_name` pair is 0, or because of a system definition error or a session-activation protocol error. The control operator should not retry the request until the condition is corrected.

A.3.2 LU62_ACTIVATION_FAILURE_RETRY

The `lu62_activate_session` verb failed to activate the session because of a temporary condition. For example, the session cannot be activated because of a temporary lack of resources at the source LU or target LU. The control operator may retry the request later.

A.3.3 LU62_ALLOCATION_ERROR

This return code indicates that a CNOS verb failed because the allocation of the CNOS conversation with the target LU cannot be completed; the subcode indicates its reason.

A.3.3.1 LU62_ALLOCATE_FAILURE_NO_RETRY

The CNOS conversation cannot be allocated on a session because of a condition that is not temporary. For example, a session activation protocol error occurs when trying to activate a session for the CNOS conversation, or the session is deactivated because of a session protocol error before the conversation can be allocated. The control operator should not retry the request until the condition is corrected.

A.3.3.2 LU62_ALLOCATE_FAILURE_RETRY

The CNOS conversation cannot be allocated on a session because of a condition that may be temporary. For example, the session to be used for the conversation cannot be activated because of a temporary lack of resources at the local LU or remote LU. The program can retry the request later.

A.3.3.3 LU62_TP_NOT_AVAILABLE_RETRY

Indicates that the target LU is currently unable to start the CNOS service transaction program (hex 06F1). The condition is temporary, and the control operator may retry the request later.

The source and target LUs' CNOS parameters are not changed by the unsuccessful CNOS verb.

A.3.4 LU62_COMMAND_RACE_REJECT

This return code indicates that the CNOS verb failed because the source or target LU is currently processing another CNOS transaction for the same `mode_name`. The other transaction is processed to completion. The source and target LUs CNOS parameters are not changed by the unsuccessful CNOS verb.

A.3.5 LU62_MODE_SESSION_LIMIT_CLOSED

This return code indicates that the CNOS verb failed because the target LU currently will not allow the (LU, mode) session limit for the specified `mode_name` to be raised above 0. This condition is not necessarily permanent, and the control operator may retry the request later.

A.3.6 LU62_MODE_SESSION_LIMIT_EXCEEDED

This return code indicates that the `lu62_activate_session` verb could not activate a session with the specified `mode_name` to the target LU for one of the following reasons:

- For a single-session connection to the target LU, either the (LU, mode) session limit is currently 0, or an LU-LU session is already active (with the specified or different `mode_name`).
- For a parallel connection to the target LU, the number of currently active sessions with the specified `mode_name` equals the (LU, mode) session limit.

A.3.7 LU62_MODE_SESSION_LIMIT_NOT_ZERO

This return code indicates that the program issued an `lu62_initialize_session_limit` verb for an (LU, mode) session limit that is already initialized (already greater than 0). The source and target LUs' CNOS parameters are unchanged.

A.3.8 LU62_MODE_SESSION_LIMIT_ZERO

This return code indicates that the program issued an `lu62_change_session_limit` verb for an (LU, mode) session limit that has not been initialized (i.e., is 0). The source and target LUs' CNOS parameters are unchanged.

A.3.9 LU62_SESSION_LIMIT_EXCEEDED

This return code indicates that the CNOS verb did not execute successfully because the new (LU, mode) session limit would cause the sum of all (LU, mode) session limits for the source LU to exceed the defined limit (`LU MAX_SESS_LMT`).

A.3.10 LU62_OK

The verb executed successfully. The following subcodes may be returned by CNOS verbs:

`LU62_OK_AS_SPECIFIED`—Indicates that the source and target LUs established the requested CNOS parameters.

`LU62_OK_AS_NEGOTIATED`—Indicates that one or more of the CNOS parameters have been negotiated. The program can receive the negotiated values by issuing `lu62_display_mode`.

`LU62_OK_FORCED`—Indicates that the source LU forced the resetting of its (LU, mode) session limit even though an error condition arose that prevented the CNOS conversation from completing. The target LUs CNOS parameters may not be changed, depending on the error condition and when it occurred.

LU62_PARAMETER_ERROR—The program issued a verb specifying a parameter containing an invalid argument. The source of the argument is considered to be outside the program definition, such as an LU name supplied by a system administrator and used as an argument. If an LU62_PARAMETER_ERROR occurs, the requested action is not performed.

A.3.11 LU62_REQUEST_EXCEEDS_MAX_ALLOWED

This return code indicates that the CNOS verb did not execute successfully because the new (LU, mode) session limit would exceed the defined maximum session limit for the (LU, mode) (MODE LCL_MAX_SESS_LMT).

A.3.12 LU62_RESOURCE_FAILURE_NO_RETRY

This return code indicates that a failure occurred, which caused the CNOS conversation to be prematurely terminated. For example, the session being used for the conversation was deactivated because of a session protocol error. The condition is not temporary, and the program should not retry the request until the condition is corrected. The CNOS parameters remain unchanged at the source LU. The target LU's parameters may have been changed, depending on when the failure occurred.

A.3.13 LU62_UNRECOGNIZED_MODE_NAME

This return code indicates that the CNOS verb did not execute successfully because the target LU did not recognize the specified mode_name. The source and target LUs' CNOS parameters are unchanged.

A.4 Product-Specific Return Codes and Subcodes

In addition to the architected return codes described above, additional SunLink LU6.2 product-specific return codes and subcodes may be received by the local program. The return codes are described below, together with selected subcodes. Refer to sunlu62.h for additional subcodes.

A.4.1 LU62_API_ERR

An internal API error has occurred while processing the verb request or response. The conversation should be aborted. Contact your local Sun dealer or call Sun Technical Support.

A.4.2 LU62_OPERATION_INCOMPLETE

A non-blocking operation was started on a conversation that was initiated or set by `lu62_set_processing_mode` to operate in `PM_NON_BLOCKING` mode. The `lu62_wait_for_server` verb is issued by the local program when it is prepared to process the return. The conversation state is unchanged until the verb return is received. No other verb may be issued on the conversation until that time.

A.4.3 LU62_SERVER_ERROR

The SunLink SNA PU2.1 9.1 server has encountered a fatal error while processing the verb. The subcode, and server console messages indicate the reason for the error. The conversation should be aborted. Contact your local Sun dealer or call Sun Technical Support.

A.4.4 LU62_SERVER_RESOURCE_FAILURE

The server failed to allocate a buffer for the verb execution.

A.4.5 LU62_TPI_ERROR

An error has occurred on the API to server interface. The majority of these errors affect the establishment and maintenance of the socket connection to the SunLink SNA PU2.1 9.1 server, and are indicated by the following subcodes:

`LU62_HOST_UNKNOWN`—The host parameter to the `lu62_open` call specifies an unknown host. The specified host is not configured in `/etc/hosts` or is not known to NIS. Contact your system administrator.

`LU62_SERV_DCNX`—The socket connection to the SunLink SNA PU2.1 9.1 server has been broken. Since a socket connection may carry multiple conversation channels, this condition is reported independently for each conversation.

LU62_SERVER_UNKNOWN—The TCP server sunlu62_serv is not configured in /etc/services or is not known to NIS. Contact your system administrator.

A.4.6 LU62_WAIT_TIMEOUT

An lu62_wait_for_server verb has returned because the specified timeout has expired and no verb was completed. The program can issue another lu62_wait_for_server verb or may abort the conversation using lu62_abort. No other verb may be issued.

Conversation State Table B

The verbs that a program may issue on a conversation depend on the state of the conversation. As the program issues verbs the state of the conversation can change. The state changes as a result of the function of the verb, the result of a verb issued by the remote program, or a result of network errors.

This appendix presents the Conversation State Table (Table 2) which defines the verbs that are valid in each conversation state, and the state changes (or transitions) that can occur when a program issues a verb. In the descriptions that follow, the SunLink LU6.2 verb names are used rather than the *IBM SNA Transaction Programmer's Reference Manual* names.

B.1 Conversation States

The state of a conversation is defined in terms of the local program's view of the its end of the conversation. A conversation may be in one of eight states, as described in Table B-1. Note that the state numbers correspond to the numbers defined in the *IBM SNA Transaction Programmer's Reference Manual* names. State numbers are missing because the sync-point states are not maintained.

Table B-1 Conversation States

#	State	Description
1	Reset	No conversation exists. The program can issue `lu62_(mc_)allocate` to initiate an outgoing conversation, or `lu62_accept` to receive an incoming conversation.
2	Send	The program is able to send data on the conversation or request confirmation.
5	Receive	The program is able to receive information from the remote program.
6	Confirm	A confirmation request was received on the conversation; that is, the remote program issued an `lu62_confirm` call and is waiting for the local program to issue `lu62_confirmed`. After responding with `lu62_confirmed`, the local program's end of the conversation enters Receive state.
7	Confirm-Send	A confirmation request and send control were received on this conversation; that is, the remote program issued `lu62_(mc_)prep_to_receive` (PR_SYNC_LEVEL) and the sync level for this conversation is SYNC_LEVEL_CONFIRM; or the remote program issued `lu62_(mc_)prep_to_receive` (PR_CONFIRM). After responding with `lu62_confirmed`, the local program's end of the conversation enters Send state.
8	Confirm-Deallocate	A confirmation request and deallocation notification were received on this conversation; that is, the remote program issued `lu62_(mc_)deallocate` (DA_SYNC_LEVEL) and the sync level for the conversation is SYNC_LEVEL_CONFIRM; or the remote program issued `lu62_(mc_)deallocate` (DA_CONFIRM). After the local program responds with `lu62_confirmed`, the conversation is deallocated.
12	Deallocate	The conversation enters this state when the remote program, the remote LU or the local LU has abnormally terminated the conversation for some reason. The local program must issue `lu62_(mc_)deallocate` (DA_LOCAL).

B.2 State Table Abbreviations

Abbreviations are used in the state table to indicate the different permutations of calls and characteristics. There are two categories of abbreviations:

- `return_code` abbreviations are enclosed by brackets [ ]
- `what_received` abbreviations are enclosed by braces { }

B.2.1 Return Code Values [ ]

The abbreviations shown in Table B-2 are used for `return_codes`.

Table B-2 Return Codes Values

Abbr	Meaning
ae	For an lu62_(mc_)allocate call, ae means one of the following: • LU62_ALLOCATION_FAILURE_NO_RETRY • LU62_ALLOCATION_FAILURE_RETRY For basic and mapped conversations, ae means one of the following: • LU62_CONVERSATION_TYPE_MISMATCH • LU62_PIP_NOT_SPECIFIED_CORRECTLY • LU62_SECURITY_NOT_VALID • LU62_SYNC_LEVEL_NOT_SUPPORTED_PGM • LU62_TPN_NOT_RECOGNIZED • LU62_TP_NOT_AVAILABLE_NO_RETRY • LU62_TP_NOT_AVAILABLE_RETRY For mapped conversations, ae can additionally mean: • LU62_FMH_DATA_NOT_SUPPORTED
da	For basic conversations, da means one of the following: • LU62_DEALLOCATE_ABEND_PROG • LU62_DEALLOCATE_ABEND_SVC • LU62_DEALLOCATE_ABEND_TIME For mapped conversations, da means: • LU62_DEALLOCATE_ABEND
dn	• LU62_DEALLOCATE_NORMAL
en	means one of the following: • LU62_PROG_ERROR_NO_TRUNC • LU62_SVC_ERROR_NO_TRUNC
ep	means one of the following: • LU62_PROG_ERROR_PURGING • LU62_SVC_ERROR_PURGING

Table B-2 Return Codes Values *(Continued)*

Abbr	Meaning
et	means one of the following: • LU62_PROG_ERROR_NO_TRUNC • LU62_SVC_ERROR_NO_TRUNC
ok	• LU62_OK
pc	• LU62_PARAMETER_CHECK
pe	• LU62_PARAMETER_ERROR
rf	means one of the following: • LU62_RESOURCE_FAILURE_NO_RETRY • LU62_RESOURCE_FAILURE_RETRY
sc	• LU62_PROGRAM_STATE_CHECK
un	• LU62_UNSUCCESSFUL

B.2.2 `what_received` *Values*

The abbreviations shown in Table B-3 are used for `what_received` values returned by `lu62_(mc_)receive_immediate` and `lu62_(mc_)receive_and_wait` verbs. See the definition of type `lu62_what_received_e` in `sunlu62.h`, Appendix F.

Table B-3 `what_received` Values

Abbr	Meaning
da	On a basic conversation, da means one of the following: • WR_DATA • WR_DATA_COMPLETE • WR_DATA_INCOMPLETE • WR_DATA_TRUNCATED • WR_LL_TRUNCATED On a mapped conversation, da means one of the following: • WR_DATA

Table B-3 `what_received` Values *(Continued)*

Abbr	Meaning
	• `WR_DATA_COMPLETE`
	• `WR_DATA_INCOMPLETE`
	• `WR_DATA_TRUNCATED`
	• `WR_LL_TRUNCATED`
	• `WR_FMH_DATA_COMPLETE`
	• `WR_FMH_DATA_INCOMPLETE`
	• `WR_FMH_DATA_TRUNCATED`
`cd`	• `WR_CONFIRM_DEALLOCATE`
`co`	• `WR_CONFIRM`
`cs`	• `WR_CONFIRM_SEND`
`se`	• `WR_SEND`

B.3 Table Symbols

The symbols shown in Table B-4 are used in the state table to indicate the condition that results when a call is issued from a certain state.

Table B-4 State Table Symbols

Symbol	Meaning
/	Cannot occur. This input is not allowed or will never return the indicated return codes for this state.
–	Remain in the current state.
1-12	Number of the next state. Note that the state numbers correspond to the numbers defined in the *IBM SNA Transaction Programmer's Reference Manual*. State numbers are missing because sync-point states are not maintained.
	It is valid to make this call from this state. See the table entries immediately below to determine the state transition resulting from the call.

B.4 Using the State Table

Each LU6.2 verb is represented in the table by a group of input rows. The possible conversation states are shown across the top of the table. The states correspond to the columns of the matrix. The intersection of input (row) and state (column) represents the validity of a verb in that particular state and, for valid calls, what state transition, if any, occurs.

The first row of each call input grouping (delineated by horizontal lines) contains the name of the call and a symbol in each state column showing whether the call is valid for that state. A call is valid for a given state only if that state's column contains the symbol Δ (delta). If the column contains "pc" (`LU62_PARAMETER_CHECK` or `sc` (`LU62_PROGRAM_STATE_CHECK`), the call is invalid for that state and the indicated `return_code` is returned. No state transitions occur for invalid verbs.

Example

For example, look at the group of input rows for the `lu62_(mc_)deallocate(Confirm)` call. The first row in this group shows that this call is only valid when the conversation is in Send state. For all other states, the call is invalid (`pc` or `sc` is indicated).

Beneath the input row containing `lu62_(mc_)deallocate` (Confirm), there are four rows showing the possible return codes returned by this call. Since the call is only valid in Send state, only this state's column contain transition values on these four rows. These transition values provide the following information:

- The conversation goes from Send state to Reset state when a return code of `LU62_OK` ("`ok`") is returned.
- The conversation goes from Send state to Deallocate state when a return code abbreviated as "`ok`," "`da`," or "`rf`" is returned.
- The conversation goes from Send state to Receive state when a return code abbreviated as "`ep`" is returned.
- There is no state transition when a return code of `LU62_PARAMETER_CHECK` ("`pc`") is returned.

Table B-5 shows the conversion state table.

Table B-5 Conversion State Table

Inputs	**States**	**Reset 1**	**Send 2**	**Receive 3**	**Confirm 4**	**Confirm Send 5**	**Confirm Dealloc 6**	**Dealloc 7**
`lu62_accept`		Δ	/	/	/	/	/	/
	`[ok]`	5						
	`[pc]`	-						
`lu62_(mc_)allocate`		Δ	sc	sc	sc	sc	sc	sc
	`[ok]`	2						
	`[ae]`	12						
	`[pc,pe,un]`	-						
`lu62_abort`		pc	Δ	Δ	Δ	Δ	Δ	Δ
	`[ok,pc]`		-	-	-	-	-	-
`lu62_(mc_)confirm`		pc	Δ	sc	sc	c	sc	sc
	`[ok,pc]`		-					
	`[ae,da,rf]`		12					
	`[ep]`		5					
`lu62_(mc_)confirmed`		pc	sc	sc	Δ	Δ	Δ	sc
	`[ok]`				5	2	12	
	`[pc]`				-	-	-	
`lu62_(mc_)deallocate(Abend)`		pc	Δ	Δ	Δ	Δ	Δ	sc
	`[ok]`		1	1	1	1	1	1
	`[pc]`		-	-	-	-	-	
`lu62_(mc_)deallocate(Confirm)`		pc	Δ	sc	sc	sc	sc	sc
	`[ok]`		1					
	`[ae,da,rf]`		12					
	`[ep]`		5					
	`[pc]`		-					
`lu62_(mc_)deallocate(Flush)`		pc	Δ	sc	sc	sc	sc	sc
	`[ok]`		1					

Table B-5 Conversion State Table *(Continued)*

Inputs **States**	**Reset 1**	**Send 2**	**Receive 3**	**Confirm 4**	**Confirm Send 5**	**Confirm Dealloc 6**	**Dealloc 7**
[pc]		-					
lu62_(mc_)deallocate(Local)	pc	sc	sc	sc	sc	sc	Δ
[ok]							1
[pc]							-
lu62_(mc_)flush	pc	Δ	sc	sc	sc	sc	sc
[ok,pc]		-					
lu62_(mc_)get_attributes	pc	Δ	Δ	Δ	Δ	Δ	Δ
[ok,pc]		-	-	-	-	-	-
lu62_get_tp_properties	pc	Δ	Δ	Δ	Δ	Δ	Δ
[ok,pc]		-	-	-	-	-	-
lu62_get_type	pc	Δ	Δ	Δ	Δ	Δ	Δ
[ok,pc]		-	-	-	-	-	-
lu62_(mc_)post_on_receipt	pc	sc	Δ	sc	sc	sc	sc
[ok,pc]			-				
lu62_(mc_)prep_to_receive(Confirm)	pc	Δ	sc	sc	sc	sc	sc
[ok,ep]		5					
[ae,da,rf]		12					
[pc]		-					
lu62_(mc_)prep_to_receive(Flush)	pc	Δ	sc	sc	sc	sc	sc
[ok]		5					
[pc]		-					
lu62_(mc_)receive_and_wait	pc	Δ	Δ	sc	sc	sc	sc
[ok]{da}		5	-				
[ok]{se}		-	2				
[ok]{co}		6	6				
[ok]{cs}		7	7				
[ok]{cd}		8	8				
[ea,da,dn,rf]		12	12				

Table B-5 Conversion State Table *(Continued)*

Inputs	States	Reset	Send	Receive	Confirm	Confirm Send	Confirm Dealloc	Dealloc
		1	2	3	4	5	6	7
	[en,ep]		5	-				
	[et]		/	-				
	[pc]		-	-				
lu62_(mc_)receive_immediate		pc	sc	Δ	sc	sc	sc	sc
	[ok]{da}			-				
	[ok]{se}			2				
	[ok]{co}			6				
	[ok]{cs}			7				
	[ok]{cd}			8				
	[ea,da,dn,rf]			12				
	[en,ep,et,pc,un]			-				
lu62_(mc_)request_to_send		pc	Δ	Δ	sc	sc	sc	sc
	[ok,pc]		-	-				
lu62_(mc_)send_data		pc	Δ					sc
	[ok]		-					
	[ae,da,rf]		12					
	[ep]		5					
	[pc]		-					
lu62_(mc_)send_error		pc	Δ	Δ	Δ	Δ	Δ	sc
	[ok]		-	2	2	2	2	
	[ae,da]		12	/	/	/	/	
	[dn]		/	12	/	/	/	
	[ep]		5	/	/	/	/	
	[rf]		12	12	12	12	12	
	[pc]		-	-	-	-	-	
lu62_(mc_)test(Posted)		pc	sc	Δ	sc	sc	sc	sc
	[ok,en,ep,et,pc,pn,un]			-				
	[ae,da,dn,rf]			12				

Table B-5 Conversion State Table *(Continued)*

Inputs	States	Reset 1	Send 2	Receive 3	Confirm 4	Confirm Send 5	Confirm Dealloc 6	Dealloc 7
lu62_(mc_)test(RTS_received)		pc	Δ	Δ	sc	sc	sc	sc
[ok][*cs]			-	-				
lu62_wait		pc	sc	Δ	sc	sc	sc	sc
[ok,en,ep,et,pc,pn,un]				-				
[ae,da,dn,rf]				12				

LU 6.2 Include Files

C

This appendix contains the SunLink P2P LU6.2 9.1 API include files:

- `sun_general.h`
- `sunlu62.h`

Code Example C-1 *(1 of 39)*

```
/*
 * COPYRIGHT (c) 1997 BY Sun Microsystems, Inc.
 */

#ifndef _sun_general_h
#define _sun_general_h

/***************************************************************************
*
* Module Name:  sun_general.h
*
* Function:     Gathering place for widely use constants and types.
*
* Usage:
*   #include "sun_general.h"
*
* Creation Date: 03/21/91
*
* Change Log:
*
```

Code Example C-1 (2 of 39)

```
*****************************************************************************/

#ifndef TRUE
#define TRUE             1
#endif

#ifndef FALSE
#define FALSE            0
#endif

#ifndef NULL
#define NULL            0
#endif

#ifndef SUN_TYPES
#define SUN_TYPES
typedef unsigned long  addr_int;                    /* integer for ptr -> int -> ptr */

#if defined(ALPHA) /* 64 bit */
typedef unsigned char  bit8;
typedef unsigned short bit16;
typedef unsigned int   bit32;
typedef unsigned int   mask;
#else /* 32 bit */
typedef unsigned char  bit8;
typedef unsigned short bit16;
typedef unsigned long  bit32;
typedef unsigned short mask;
#endif /* !ALPHA */
#endif /* SUN_TYPES */

#if defined(MSWINDOWS) || defined(WNT)

#define CFG

#include <stdio.h>
#include <string.h>
#ifndef MSWINDOWS
#include "tydefs.h"
#endif
#endif  /* MSWINDOWS or WNT */

#if defined(ALPHA) /* 64 bit */
```

Code Example C-1 (3 of 39)

```
typedef unsigned char    BIT8;
typedef unsigned short    BIT16;
typedef unsigned int    BIT32;
#else /* 32 bit */
typedef unsigned char    BIT8;
typedef unsigned short    BIT16;
typedef unsigned long    BIT32;
#endif /* !ALPHA */

typedef char    CHAR;
typedef short    SHORT;

#ifndef INT
#define INT    int
#endif

#ifndef LONG
#define LONG    long
#endif

typedef unsigned int    UINT;

#ifndef ULONG
typedef unsigned long    ULONG;
#endif

typedef int    RESULT;


#if !defined (MSWINDOWS)
#define FAR
#define _far
#define __far
#define NEAR
#define _near
#define __near
#endif /* not MSWINDOWS and not WNT */


/* #ifdef MSWINDOWS */

typedef ULONG FAR*    FPULONG;
```

Code Example C-1 (4 of 39)

```
typedef char NEAR*    NPCHAR;
typedef char FAR*    FPCHAR;

typedef CHAR NEAR*    PSTR;
typedef CHAR NEAR*    NPSTR;
typedef CHAR FAR*    FPSTR;
#ifdef MSWINDOWS
typedef const CHAR FAR*    FPCSTR;
#endif

typedef BIT8 FAR*    PBIT8;
typedef BIT8 NEAR*    NPBIT8;
typedef BIT8 FAR*    FPBIT8;

typedef BIT16 FAR*    PBIT16;
typedef BIT16 NEAR*    NBIT16;
typedef BIT16 FAR*    FPBIT16;

typedef BIT32 FAR*    PBIT32;
typedef BIT32 NEAR*    NPBIT32;
typedef BIT32 FAR*    FPBIT32;

typedef INT NEAR*    PINT;
typedef INT NEAR*    NPINT;
typedef INT FAR*    FPINT;

typedef void FAR*    FPVOID;

/* #endif */ /* MSWINDOWS */

#if 0
#ifdef WNT

typedef ULONG *    FPULONG;

typedef char *    NPCHAR;
typedef char *    FPCHAR;

typedef CHAR *    PSTR;
typedef CHAR *    NPSTR;
typedef CHAR *    FPSTR;
typedef const CHAR *    FPCSTR;
```

Code Example C-1 (5 of 39)

```
typedef BIT8 *    PBIT8;
typedef BIT8 *    NPBIT8;
typedef BIT8 *    FPBIT8;

typedef BIT16 *    PBIT16;
typedef BIT16 *    NBIT16;
typedef BIT16 *    FPBIT16;

typedef BIT32 *    PBIT32;
typedef BIT32 *    NPBIT32;
typedef BIT32 *    FPBIT32;

typedef INT *    PINT;
typedef INT *    NPINT;
typedef INT *    FPINT;

typedef void *    FPVOID;

#endif /* WNT */
#endif

#if defined(SVR4) || defined(WNT)

#if !defined(KERNEL) && !defined(_KERNEL)
#define bcopy(f,t,n)    memcpy(t,f,n)
#define bzero(s,n)      memset(s,0,n)
#define bcmp(s,d,n)     memcmp(s,d,n)
#endif /* not KERNEL */

#define index(s,r)      strchr(s,r)
#define rindex(s,r)     strrchr(s,r)

#endif /* SVR4 or WNT */

#ifdef MSWINDOWS
#define bcopy(a,b,c)             _fmemcpy((FPCHAR ) b,(FPCHAR ) a,c)
#define bzero(s,n)               _fmemset((FPCHAR ) s,'',n)
#define bcmp(s,d,n)              _fmemcmp((FPCHAR ) s,(FPCHAR ) d,n)

#endif

#ifdef WNT
```

Code Example C-1 (6 of 39)

```
#ifdef EVENT_VIEWER
extern brx_log_event(BIT32 type, ...);
#define printf sun_log_event
#else
extern void brx_printf(char *, ...);
#define printf sun_printf

#endif /* WNT and not EVENT_VIEWER */

#endif /* WNT */

#ifdef _SEQUENT_
#define gettimeofday(tvp,x) get_process_stats(tvp, -1, NULL, NULL)
typedef unsigned char    u_char;
typedef unsigned short    u_short;
typedef unsigned long    u_long;
#endif /* _SEQUENT_ */

#endif /* _sun_general_h */


/*
 * COPYRIGHT (c) 1997 Sun Microsystems, Inc.
 */

#ifndef BRXLU62_H
#define BRXLU62_H


/*****************************************************************************
*
* Module Name:  sunlu62.h
*
* Function:     This include file contains the SUNLU62 application interface
*      definitions.
*
* Usage:
*   #include "sunlu62.h"
*
* External data definitions:
*   lu62_errno;       sunlu62 library errors
*   lu62_trace_flag;             sunlu62 trace options
*
```

Code Example C-1 (7 of 39)

```
* External routine definitions:
*       all SUNLU62 API routines
*
* Creation Date: 1/17/92
*
* Change Log:
*
*****************************************************************************/

#ifdef WNT
#include <stdlib.h>
#include <winsock.h>

#else /* !WNT */
#include <sys/param.h>                          /* system parameters */
#endif

#include <sys/types.h>

#ifdef SVR4
#include <netdb.h>                              /* network db parameters */
#endif

#ifdef SCO
#include <sys/socket.h>                         /* MAXHOSTNAMELEN */
#endif

#ifndef MAXHOSTNAMELEN
#define MAXHOSTNAMELEN 128
#endif

#include "sun_general.h"


/*****************************************************************************
 *
 * LU62 Constants
 *
 *****************************************************************************
 */
#define VO_LU62_TP_NAME_LEN                 8
#define LU62_TP_NAME_LEN                  64
```

Code Example C-1 (8 of 39)

```
#define LU62_LU_NAME_LEN                     8
#define LU62_NQ_LU_NAME_LEN                  17
#define LU62_MODE_NAME_LEN                   8
#define LU62_MAP_NAME_LEN                    8
#define VO_LU62_MAX_USER_LEN                  8
#define VO_LU62_MAX_PASSWD_LEN                 8
#define VO_LU62_MAX_PROFILE_LEN                10
#define LU62_MAX_USER_ID_LE                  64
#define LU62_MAX_PASSWD_LEN                  64
#define LU62_MAX_PROFILE_LEN                 64
#define LU62_MAX_DATA_LEN3                      32767
#define LU62_MAX_CONV_CORR_LEN               8
#define LU62_MAX_LOG_DATA_LEN                512
#define LU62_MAX_RESOURCE_LIST_ENTRIES       256
#define LU62_SESSION_ID_LEN                  16  /* ASCII/HEX for COPR verbs */
#define LU62_MAX_SESS_ID_LEN                 8   /* binary for attributes */
#define LU62_MIN_LUW_LEN                     10
#define LU62_MAX_LUW_LEN                     26
#define LU62_LUW_INSTANCE_LEN                16

#define SNASVCMG"SNASVCM4"
#define CPSVCMG"CPSVCM4"
#define LU62_ALL_MODES                        "*"


/****************************************************************************
 *
 * LU62 Message Field typedefs
 *
 ****************************************************************************
 */

/* Conversation Type */
typedef enum lu62_conv_type {
    CONVERSATION_BASIC = 0,
    CONVERSATION_MAPPED
} lu62_conv_type_e;

/* Conversation States */
typedef enum lu62_conv_state {

    CONV_RESET = 0,
```

Code Example C-1 (9 of 39)

```
    CONV_SEND,
    CONV_DEFER_RECEIVE,
    CONV_DEFER_DEALLOCATE,
    CONV_RECEIVE,

    CONV_CONFIRM,
    CONV_CONFIRM_SEND,
    CONV_CONFIRM_DEALLOCATE,
    CONV_SYNCPT,
    CONV_SYNCPT_SEND,
    CONV_SYNCPT_DEALLOCATE,
    CONV_DEALLOCATE,
    CONV_BACKOUT_REQD,
    CONV_CANT_HAPPEN

} lu62_conv_state_e;

/* Deallocate Type */
typedef enum {
    DA_SYNC_LEVEL = 0,
    DA_FLUSH,
    DA_CONFIRM,
    DA_ABEND,/* MC only */
    DA_ABEND_PROG,/* Basic only */
    DA_ABEND_SVC,/* Basic only */
    DA_ABEND_TIMER,/* Basic only */
    DA_LOCAL,
    /* Sync Level syncpoint additions */
    DA_UNBIND
} lu62_deallocate_type_e;

/* Encrypt Type */
typedef enum {
    ENCRYPT_NO = 0,
    ENCRYPT_YES = 1
} lu62_encrypt_e;

/* Fill type */
typedef enum {
    FILL_LL = 0,
    FILL_BUFFER = 1
lu62_fill_e;
```

Code Example C-1 (10 of 39)

```
/* FMH Data Type */
typedef enum {
    FMH_NO = 0,
    FMH_YES = 1,
    /* Sync Level syncpoint additions */
    FMH_ELN,
    FMH_CS
} lu62_fmh_data_e;

/* forget indicator type */
typedef enum {
    FORGET_NO = 0,
    FORGET_YES = 1
} lu62_forget_e;

/* Locks type */
typedef enum {
    LOCKS_SHORT = 0,
    LOCKS_LONG = 1
} lu62_locks_e;

/* Log Data Type */
typedef enum {
    FLUSH_NO = 0,
    FLUSH_YES = 1
} lu62_flush_e;

/* PIP presence in allocate or attach */
typedef enum {
    PIP_NOT_PRESENT = 0,
    PIP_PRESENT = 1
} lu62_pip_presence_e;

/* Post Control */
typedef enum {
    PC_ALWAYS_POST = 0,
    PC_USER_POST = 1
} lu62_post_control_e;

/* Prepare to Receive Type */
typedef enum {
    PR_SYNC_LEVEL = 0,
```

Code Example C-1 (11 of 39)

```
    PR_FLUSH,
    PR_CONFIRM
} lu62_prep_to_rcv_type_e;

/* Processing Mode */
typedef enum {
    PM_BLOCKING = 0,
    PM_NON_BLOCKING = 1
} lu62_processing_mode_e;

/* Program Type */
typedef enum {
    PROG = 0,
    PROG_SVC = 1
} lu62_prog_type_e;

/* Allocation Return Control */
typedef enum {
    RC_WHEN_SESSION_ALLOCATED = 0,
    RC_IMMEDIATE,
    RC_WHEN_CONWINNER_ALLOCATED,
    RC_WHEN_CONV_GROUP_ALLOCATED
} lu62_return_control_e;

/* Conversation Security */
typedef enum {
    SECURITY_NONE = 0,
    SECURITY_SAME,
    SECURITY_PROGRAM
lu62_security_e;

/* Conversation Sync Level */
typedef enum {
    SYNC_LEVEL_NONE = 0,
    SYNC_LEVEL_CONFIRM,
    SYNC_LEVEL_SYNCPT
} lu62_sync_level_e;

/* Test Type */
typedef enum {
    TEST_POSTED = 0,
    TEST_REQUEST_TO_SEND_RECEIVED = 1
```

Code Example C-1 (12 of 39)

```
} lu62_test_type_e;

/* What Received */
typedef enum {
    WR_DATA = 0,/* Basic only */
    WR_LL_TRUNCATED,/* Basic only */
    WR_DATA_COMPLETE,
    WR_DATA_INCOMPLETE,
    WR_DATA_TRUNCATED,/* MC only */
    WR_FMH_DATA_COMPLETE,/* MC only */
    WR_FMH_DATA_INCOMPLETE,/* MC only */
    WR_FMH_DATA_TRUNCATED,/* MC only */
    WR_SEND,
    WR_CONFIRM,
    WR_CONFIRM_SEND,
    WR_CONFIRM_DEALLOCATE,
    WR_TAKE_SYNCPT,
    WR_TAKE_SYNCPT_SEND,
    WR_TAKE_SYNCPT_DEALLOCATE,

    /* Sync Level syncpoint additions */
    WR_PS_DATA_COMPLETE,/* Basic SYNC_LEVEL_SYNCPT only */
    WR_PS_DATA_INCOMPLETE,/* Basic SYNC_LEVEL_SYNCPT only */
} lu62_what_received_e;

typedef bit8 lu62_pip_t[32];


/* Sync Level syncpoint additions */

/* lu62_listen Response Type */
typedef enum {
    LISTEN_ATTACH = 0,
    LISTEN_FORGET = 1
} lu62_response_type_e;


/*****************************************************************************
 *
 * LU62 COPR Message Field typedefs
 *
 *****************************************************************************
```

Code Example C-1 (13 of 39)

```
 */

/* Deactivate Session Type */
typedef enum {
    DS_CLEANUP = 0,
    DS_NORMAL
} lu62_ds_type_e;

/* Deactivate Session Return_Control */
typedef enum {
    DS_IMMEDIATE = 0,
    DS_DELAYED
} lu62_ds_return_control_e;

/* Change Session Limit Responsible */
typedef enum {
    SL_SOURCE = 0,
    SL_TARGET
} lu62_responsible_lu_e;

/* Single Session Reinitiation responsibility */
typedef enum {
    SR_OPERATOR = 0,
    SR_PLU,
    SR_SLU,
    SR_PLU_OR_SLU
} lu62_single_session_reinit_e;

/* Session Level Cryptography */
typedef enum {
    SC_NOT_SUPPORTED = 0,
    SC_MANDATORY,
    SC_SELECTIVE
} lu62_session_level_crypto_e;

/* Session Initiation Type */
typedef enum {
    SI_INITIATE_ONLY = 0,
    SI_INITIATE_OR_QUEUE
} lu62_initiate_type_e;

/* Security Acceptance Level */
```

Code Example C-1 (14 of 39)

```
typedef enum {
    SA_NONE = 0,
    SA_CONVERSATION,
    SA_ALREADY_VERIFIED
} lu62_security_accept_e;

/* Security Required Level */
typedef enum {
    SE_NONE = 0,
    SE_CONVERSATION,
    SE_ACCESS
} lu62_security_required_e;

/* TP Status */
typedef enum {
    TP_ENABLED = 0,
    TP_TEMP_DISABLED,
    TP_PERM_DISABLED
} lu62_tp_status_e;

/* PIP */
typedef enum {
    PIP_NO = 0,
    PIP_NO_LU_VERIFICATION,
    PIP_YES
 } lu62_pip_e;


/**************************************************************************
 *
 * Open and Close Requests
 *
 **************************************************************************
 */
typedef struct {
    char host[MAXHOSTNAMELEN+1];/* s */
    char lu_name[LU62_LU_NAME_LEN+1];/* so */
    char tp_name[LU62_TP_NAME_LEN+1];/* so */
    lu62_processing_mode_e processing_mode;/* s */

    bit32 return_code;  /* r */
    bit32 port_id; /* r */
```

Code Example C-1 (15 of 39)

```
    int port_desc; /* r */
} lu62_open_req_t;


typedef struct
    bit32  port_id;   /* s */
    bit32 return_code;  /* r */
lu62_close_req_t;


/*****************************************************************************
 *
 * Sun LU6.2 Verbs
 *
 *****************************************************************************
 */

typedef struct
    bit32 port_id;/* s */
    bit32 own_tp_instance;/* s/r */
    lu62_processing_mode_e processing_mode;/* s */

    bit32 conv_id; /* r */
    bit32 p_id;    /* r */
    lu62_pip_presence_e pip_presence;    /* r */
    bit32 return_code;   /* r */
} lu62_accept_t;


typedef struct {
    bit32 port_id;   /* s */
    char tp_name[LU62_TP_NAME_LEN+1];   /* s */

    bit32  tp_id;  /* r */
    bit32 return_code;  /* r */
} lu62_register_tp_t;


typedef struct {
    bit32 conv_id;   /* s */
    bit32 return_code;  /* r */
} lu62_abort_t;
```

Code Example C-1 (16 of 39)

```
typedef struct {
    bit32 port_id;    /* s */
    lu62_processing_mode_e processing_mode; /* s */

    /* conv_id is only valid while LU62_OPERATION_INCOMPLETE */
    bit32 conv_id; /* r */

    /* Accept parameters */
    lu62_pip_presence_e pip_presence; /* r */

    /* Conversation Type - see lu62_get_type */
    lu62_conv_type_e type;  /* r */

    /* Conversation Attributes - see lu62_get_attributes */
    char partner_lu_name[LU62_LU_NAME_LEN+1];/* r */
    char   mode_name[LU62_MODE_NAME_LEN+1];/* r */
    bit8   partner_qlu_name[LU62_NQ_LU_NAME_LEN+1];/* r */
    int    partner_qlu_name_len;/* r */
    lu62_sync_level_e sync_level; /* r */
    int    conv_corr_len; /* r */
    bit8   conv_corr[LU62_MAX_CONV_CORR_LEN]; /* r */
    bit32   conv_grp_id;  /* ???? *//* r */

    /* TP Properties - see lu62_get_tp_properties */
    bit32  tp_id;            /* r */
    bit32  own_tp_instance;/* r */
    char   tp_name[LU62_TP_NAME_LEN+1];/* r */
    char local_len_name[LU62_LU_NAME+1];   /* r */
    bit8                qlu_name[LU62_NQ_LU_NAME_LEN+1];        /* r */
    int                 qlu_name_len;                           /* r */
    char   user_id[LU62_MAX_USER_ID_LEN+1];/* r */
    int    user_id_len;   /* r */
    char   profile[LU62_MAX_PROFILE_LEN+1];/* r */
    int    profile_len;   /* r */

    bit32  return_code;   /* r */

    /* Sync Level syncpoint additions */
    lu62_response_type_e response_type; /* r */
    int    sess_id_len;   /* r */
    bit8   sess_id[LU62_MAX_SESS_ID_LEN];/* r */
    int    luw_len;       /* r */
```

Code Example C-1 (17 of 39)

```
    bit8    luw[LU62_MAX_LUW_LEN];/* r */
} lu62_listen_t;


/*****************************************************************************
 *
 * Standard LU6.2 Verbs
 *
 *****************************************************************************
 */
typedef struct {
    bit32 port_id;/* s */
    bit32 tp_id;          /* s */
    char lu_name[LU62_LU_NAME_LEN+1];  /* s */
    char mode_name[LU62_MODE_NAME_LEN+1]; /* s */
    char   remote_tp_name[LU62_TP_NAME_LEN+1];
                              /* s */
    bit32 conv_grp_id;   /* s */
    lu62_processing_mode_e processing_mode; /* s */
    lu62_conv_type_e type;/* Basic only */ /* s */
    lu62_flush_e flush;  /* s */
    lu62_return_control_e return_control; /* s */
    lu62_sync_level_e sync_level; /* s */
    lu62_pip_presence_e pip_presence; /* s */
    lu62_security_e security; /* s */
    char user_id[LU62_MAX_USER_ID_LEN+1]; /* s */
    char passwd[LU62_MAX_PASSWD_LEN+1]; /* s */
    char profile[LU62_MAX_PROFILE_LEN+1];  /* s */

    bit32 conv_id;    /* r */
    bit32 return_code;/* r */
} lu62_allocate_t;


typedef struct {
    bit32 conv_id; /* s */
    bit32 return_code; /* r */
    bit32 request_to_send_received; /* r */
lu62_confirm_t;


typedef struct {
```

Code Example C-1 (18 of 39)

```
    bit32 conv_id;  /* s */
    bit32 return_code;/* r */
lu62_confirmed_t;


typedef struct
    bit32 conv_id;/* s */
    lu62_deallocate_type_e type;/* s */
    char *log_data;/* Basic only *//* s */
    bit32 return_code;/* r */
} lu62_deallocate_t;


typedef struct
    bit32 conv_id;/* s */
    bit32 return_code;/* r */
lu62_flush_t;


typedef struct
    bit32conv_id;     /* s */

    bit32 return_code;/* r */
    char partner_lu_name[LU62_LU_NAME_LEN+1]; /* r */
    char mode_name[LU62_MODE_NAME_LEN+1];/* r */
    bit8 partner_qlu_name[LU62_NQ_LU_NAME_LEN+1];/* r */
    int    partner_qlu_name_len;
    lu62_sync_level_esync_level; /* r */
    lu62_conv_state_e   conv_state;
    int    conv_corr_len; /* r */
    bit8 conv_corr[LU62_MAX_CONV_CORR_LEN]; /* r */
    bit32 conv_grp_id;  /* ???? *//* r */

    /* Sync Level syncpoint additions */
    int    sess_id_len;   /* r */
    bit8 sess_id[LU62_MAX_SESS_ID_LEN];/* r */
    int    luw_len;       /* r */
    bit8 luw[LU62_MAX_LUW_LEN];/* r */
} lu62_get_attributes_t;


typedef struct {
```

Code Example C-1 (19 of 39)

```
    bit32 conv_id;  /* s */
    int    length;           /* s */
    lu62_fill_e fill;  /* Basic only *//* s */
    bit32 return_code;/* r */
} lu62_post_on_receipt_t;


typedef struct {
    bit32 conv_id;  /* s */
    bit32 return_code; /* r */
} lu62_prep_for_syncpt_t;


typedef struct
    bit32 conv_id;    /* s */
    lu62_prep_to_rcv_type_e type;/* s */
    lu62_locks_elocks; /* s */
    bit32 return_code;/* r */
} lu62_prep_to_receive_t;


typedef struct {
    bit32 conv_id;    /* s */
    lu62_fill_e fill;/* Basic only *//* s */

    int    length;           /* s/r */

    bit32 return_code;/* r */
    bit32 request_to_send_received;/* r */
    bit8 *data;           /* r */
    lu62_what_received_e  what_received;/* r */
    char map_name[LU62_MAP_NAME_LEN+1];/* r */
                  /* MC only */
} lu62_receive_t;


typedef struct {
    bit32 conv_id;    /* s */
    bit32 return_code;/* r */
} lu62_request_to_send_t;
```

Code Example C-1 (20 of 39)

```
typedef struct {
    bit32  conv_id;   /* s */
    bit8 *data;           /* s */
    int    length;            /* s */

    charmap_name[LU62_MAP_NAME_LEN+1];/* s */
                  /* MC only */
    lu62_fmh_data_e  fmh_data;/* MC only *//* s */
    lu62_encrypt_e encrypt;/* s */
    lu62_flush_e flush; /* s */

    bit32 return_code;/* r */
    bit32 request_to_send_received;/* r */
} lu62_send_data_t;


type def struct
    bit 32conv_id;    /* s */
    bit8 *data;           /* s */
    int    length;            /* s */

    lu62_forget_e forget;/* s */
    lu62_flush_e flush; /* s */

    bit32 return_code;/* r */
    bit32 request_to_send_received;/* r */
} lu62_send_ps_data_t;


typedef struct {
    bit32 conv_id;    /* s */
    lu62_prog_type_e type;/* Basic only *//* s */
    char *log_data;/* Basic only *//* s */
    int    error_direction;/* CPIC only *//* s */

    bit32 return_code;/* r */
    bit32 request_to_send_received;/* r */
} lu62_send_error_t;


typedef struct {
    bit32 conv_id;    /* s */
```

Code Example C-1 (21 of 39)

```
    lu62_test_type_etest;/* s */
    bit32 return_code;/* r */
} lu62_test_t;

typedef struct {
    bit32 conv_id;     /* s */
    lu62_conv_type_etype;/* r */
    bit32 return_code;/* r */
} lu62_get_type_t;


typedef struct {
    bit32 conv_id;     /* s */

    bit32 return_code;/* r */
    bit32 own_tp_instance;/* r */
    char tp_name[LU62_TP_NAME_LEN+1];/* r */
    bit8 qlu_name[LU62_NQ_LU_NAME_LEN+1];/* r */
    int     qlu_name_len;  /* r */
    char user_id[LU62_MAX_USER_ID_LEN+1];/* r */
    int     user_id_len;   /* r */
    char profile[LU62_MAX_PROFILE_LEN+1];/* r */
    int     profile_len;   /* r */
} lu62_get_tp_properties_t;


typedef struct {
    bit32 port_id;     /* s */
    int     conv_count;    /* s */
    bit32 *conv_list; /* s */

    bit32 conv_id;     /* r */
    bit32 return_code;/* r */
} lu62_wait_t;


/*****************************************************************************
 *
 * LU6.2 COPR Verbs
 *
 *****************************************************************************
 */
```

Code Example C-1 (22 of 39)

```
typedef struct {
    bit32 port_id;    /* s */
    char                lu_name[LU62_LU_NAME_LEN+1];            /* s */
    char                mode_name[LU62_MODE_NAME_LEN+1];        /* s */
    bit32 return_code;/* r */
} lu62_activate_session_t;


typedef struct {
    bit32 port_id;    /* s */
    bit32 conv_group_id;/* s */
    lu62_ds_type_e type;/* s */
    int sense_code_supplied;/* s */
    bit32 sense_code; /* s */
    lu62_ds_return_control_e  return_control;/* s */

    bit32 return_code;/* r */
} lu62_deactivate_conv_group_t;


typedef struct {
    bit32 port_id;    /* s */
    char session_id[LU62_SESSION_ID_LEN+1];/* s */
    lu62_ds_type_e type;/* s */
    int sense_code_supplied;/* s */
    bit32 sense_code; /* s */
    lu62_ds_return_control_e return_control;/* s */

    bit32 return_code;/* r */
} lu62_deactivate_session_t;


typedef struct {
    bit32port_id;     /* s */
    char lu_name[LU62_LU_NAME_LEN+1];            /* s */
    char mode_name[LU62_MODE_NAME_LEN+1];        /* s */
    int    lu_mode_session_limit;/* s */
    int    min_conwinners_source;/* s */
    int    min_conwinners_target;/* s */
    lu62_responsible_lu_e responsible_lu;/* s */

    bit32 return_code;/* r */
```

Code Example C-1 (23 of 39)

```
} lu62_change_session_limit_t;

typedef struct {
    bit 32port_id;    /* s */
    char lu_name[LU62_LU_NAME_LEN+1];            /* s */
    char mode_name[LU62_MODE_NAME_LEN+1];        /* s */
    int    lu_mode_session_limit;/* s */
    int    min_conwinners_source;/* s */
    int    min_conwinners_target;/* s */

    bit32 return_code;/* r */
} lu62_init_session_limit_t;

typedef struct {
    bit32 port_id;    /* s */
    char lu_name[LU62_LU_NAME_LEN+1];            /* s */
    char mode_name[LU62_MODE_NAME_LEN+1];        /* s */
    lu62_responsible_lu_e responsible_lu;/* s */
    int    drain_source;  /* s */
    int    drain_target;  /* s */
    int    force;             /* s */

    bit32 return_code;/* r */
} lu62_reset_session_limit_t;


typedef struct {
    bit32 port_id;    /* s */
    bit8 *buffer;     /* s */
    int    length;            /* s/r */

    bit32 return_code;/* r */
    char nq_lu_name[LU62_NQ_LU_NAME_LEN+1];/* r */
    char lu_name[LU62_LU_NAME_LEN+1];/* r */
    int    lu_session_limit;/* r */
    int    lu_session_count;/* r */
    int    bind_rsp_queue_capability;/* r */
    int    security_count;/* r */
    int    map_name_count;/* r */
    int    remote_lu_name_count;/* r */
    int    tp_name_count; /* r */
lu62_display_local_lu_t;
```

Code Example C-1 (24 of 39)

```
typedef struct
    bit32port_id;      /* s */
    char nq_lu_name[LU62_NQ_LU_NAME_LEN+1];/* s */
    char mode_name[LU62_MODE_NAME_LEN+1];/* s */
    bit8*buffer;       /* s */
    int    length;            /* s/r */

    bit32return_code; /* r */
    char lu_name[LU62_LU_NAME_LEN+1];/* r */
    int    send_max_ru_size_lb;/* r */
    int    send_max_ru_size_ub;/* r */
    int    recv_max_ru_size_lb;/* r */
    int    recv_max_ru_size_ub;/* r */
    lu62_single_session_reinit_e  single_session_reinit;/* r */
    lu62_session_level_crypto_e  session_level_crypto;/* r */
    int    conwinner_autoactivate_limit;/* r */
    int    local_max_session_limit;/* r */
    int    lu_mode_session_limit;/* r */
    int    min_conwinners;/* r */
    int    min_conlosers; /* r */
    int    termination_count;/* r */
    int    drain_local_lu;/* r */
    int    drain_remote_lu;/* r */
    int    lu_mode_session_count;/* r */
    int    conwinners_session_count;/* r */
    int    conlosers_session_count;/* r */
    int    conv_group_count;/* r */
    int    preferred_received_ru_size;/* r */
    int    preferred_send_ru_size;/* r */
    char sess_deact_tp_name[LU62_TP_NAME_LEN+1];/* r */
} lu62_display_mode_t;


typedef struct {
    bit32 port_id;     /* s */
    char nq_lu_name[LU62_NQ_LU_NAME_LEN+1];/* s */
    bit8 *buffer;      /* s */
    int    length;            /* s/r */

    bit32return_code; /* r */
    char lu_name[LU62_LU_NAME_LEN+1];/* r */
```

Code Example C-1 (25 of 39)

```
    char ui_lu_name[LU62_LU_NAME_LEN+1];/* r */
    lu62_initiate_type_e initiate_type;/* r */
    int    parallel_session_support;/* r */
    int    cnos_support;  /* r */
    lu62_security_accept_e security_accept_local_lu;/* r */
    lu62_security_accept_e security_accept_remote_lu;/* r */
    int    mode_name_count;/* r */
} lu62_display_remote_lu_t;


typedef struct {
    bit32 port_id;     /* s */
    char nq_lu_name[LU62_NQ_LU_NAME_LEN+1];/* so */
    char tp_name[LU62_TP_NAME_LEN+1];/* s */
    bit8 *buffer;      /* s */
    int    length;            /* s/r */

    bit32 return_code;/* r */
    lu62_tp_status_e status;/* r */
    int    basic_support; /* r */
    int    mapped_support;/* r */
    int    sync_level_none; /* r */
    int    sync_level_confirm; /* r */
    int    sync_level_syncpt; /* r */
    lu62_security_required_e  security_required; /* r */
    int    security_access_count;/* r */
    lu62_pip_e pip;       /* r */
    int    pip_count;     /* r */
    int    data_mapping;  /* r */
    lu62_fmh_data_e fmh_data;/* r */
    int    cnos_privilege;/* r */
    int    session_control_privilege;/* r */
    int    define_privilege;/* r */
    int    display_privilege;/* r */
    int    allocate_svc_tp_privilege;/* r */
    int    instance_limit;/* r */
    int    instance_count;/* r */
} lu62_display_tp_t;


/**************************************************************************
 *
```

Code Example C-1 (26 of 39)

```
 * COPR Notifications
 *
 ******************************************************************************
 */
typedef struct {
    char                host[MAXHOSTNAMELEN+1];                 /* s */
    char                lu_name[LU62_LU_NAME_LEN+1];            /* s */

    bit32 return_code;  /* r */
    bit32               port_id;                                /* r */
    int                 port_desc;                              /* r */
} lu62_request_notification_t;


typedef struct {
    bit32 port_id;    /* s */
    bit32 return_code;  /* r */
} lu62_stop_notification_t;


typedef enum {
    LU62_REQUEST_NOTIFICATION_REPLY = 1,
    LU62_STOP_NOTIFICATION_REPLY,
    LU62_CNOS_NOTIFICATION
} lu62_op_code_e;


typedef enum {
    LU62_RESET_SESSION_LIMIT = 1,
    LU62_INIT_SESSION_LIMIT,
    LU62_CHANGE_SESSION_LIMIT
} lu62_cnos_type_e;


typedef struct {
    bit32               port_id;                                /* s */
    lu62_op_code_e  op_code;/* r */
    bit32  return_code;/* r */
} lu62_notification_header_t;


typedef struct {
```

Code Example C-1 (27 of 39)

```
    lu62_cnos_type_e cnos_type;
    int    local_invocation;
    int                     lu_mode_session_limit;
    int                     min_conwinners_source;
    int                     min_conwinners_target;
    lu62_responsible_lu_e responsible_lu;
    int                     drain_source;
    int                     drain_target;
    int                     force;
    char                    lu_name[LU62_LU_NAME_LEN+1];
    char                    mode_name[LU62_MODE_NAME_LEN+1];
} lu62_cnos_notification_t;


/***********************************************************************
 *
 * API Verb Return Codes and Error Handling
 *
 * The majority of the LU62 verbs return a 32 bit return_code in the
 * corresponding field of the user's request.
 *
 * In addition, all LU62 verbs return an integer value.  A zero (LU62_OK)
 * return indicates success.  The negative LU62_ERROR return value indicates
 * that an error occurred.  In this case, lu62_errno is set to indicate the
 * reason for failure.
 *
 * In the case of an error detected by the API, the return_code and
 * lu62_errno will have the same value.
 *
 *---------------------------------------------------------------------
 * API return values
 *---------------------------------------------------------------------
 */
#define LU62_ERROR          -1


/*---------------------------------------------------------------------
 * Verb Return Codes - also set in lu62_errno on LU62_ERROR
 *---------------------------------------------------------------------
 */

/*
```

Code Example C-1 (28 of 39)

```
 * OK Returns
 */
#define LU62_OK 0x00000000
#define LU62_OK_DATA 0x00000001
#define LU62_OK_NOT_DATA 0x00000002
#define LU62_OK_ALL_AGREED 0x00000003
#define LU62_OK_VOTED_READ_ONLY 0x00000004
#define LU62_OK_LUW_OUTCOME_PENDING 0x00000005
#define LU62_OK_LUW_OUTCOME_MIXED 0x00000006

/* Additional non-blocking OK returns */
#define LU62_OPERATION_INCOMPLETE 0x00000010
#define LU62_READ_INCOMPLETE 0x00000011
#define LU62_WAIT_TIMEOUT  0x00000012

/*
 * API Errors
 */
#define LU62_API_ERR 0x00010000
#define LU62_INTERNAL_ERR 0x00010001
#define LU62_SUN_ALLOC_FAILURE 0x00010002
#define LU62_TPI_CREATE_FAILURE 0x00010003

/*
 * API/Server Errors
 */
#define LU62_TPI_ERROR 0x00020000
#define LU62_SERVER_UNKNOWN 0x00020001
#define LU62_HOST_UNKNOWN 0x00020002
#define LU62_SOCKET 0x00020003
#define LU62_CONNECT 0x00020004
#define LU62_SELECT 0x00020005
#define LU62_SERV_DCNX 0x00020006
#define LU62_OPEN_FAIL 0x00020007
#define LU62_SERVER_WRITE 0x00020008
#define LU62_SERVER_READ 0x00020009
#define LU62_UNEXPECTED_RSP 0x0002000A
#define LU62_UNKNOWN_RSP 0x0002000B
#define LU62_UNKNOWN_NOTIFICATION 0x0002000C
#define LU62_CONV_ID_MISMATCH 0x0002000D
#define LU62_TP_DISCONNECTED 0x0002000E
```

Code Example C-1 (29 of 39)

```
/*
 * Server Errors
 */
#define LU62_SERVER_ERROR 0x00030000
#define LU62_SERVER_RESOURCE_FAILURE 0x00030001

/*
 * Allocation Errors
 */
#define LU62_ALLOCATION_ERROR 0x00040000
#define LU62_ALLOCATION_FAILURE_NO_RETRY 0x00040001
#define LU62_ALLOCATION_FAILURE_RETRY 0x00040002
#define LU62_CONVERSATION_TYPE_MISMATCH 0x00040003
#define LU62_PIP_NOT_ALLOWED 0x00040004
#define LU62_PIP_NOT_SPECIFIED_CORRECTLY 0x00040005
#define LU62_SECURITY_NOT_VALID 0x00040006
#define LU62_SYNC_LEVEL_NOT_SUPPORTED_BY_LU 0x00040007
#define LU62_SYNC_LEVEL_NOT_SUPPORTED_BY_PGM 0x00040008
#define LU62_TPN_NOT_RECOGNIZED 0x00040009
#define LU62_TP_NOT_AVAILABLE_NO_RETRY 0x0004000A
#define LU62_TP_NOT_AVAILABLE_RETRY 0x0004000B

/*
 * Backed Out Errors
 */
#define LU62_BACKED_OUT 0x00050000
#define LU62_BACKED_OUT_ALL_AGREED 0x00050001
#define LU62_BACKED_OUT_LUW_OUTCOME_PENDING 0x00050002
#define LU62_BACKED_OUT_LUW_OUTCOME_MIXED 0x00050003

/*
 * Deallocation Errors
 */
#define LU62_DEALLOCATE_ABEND 0x00060000
#define LU62_DEALLOCATE_ABEND_BO 0x00060001

#define LU62_DEALLOCATE_ABEND_PROG 0x00070000
#define LU62_DEALLOCATE_ABEND_PROG_BO 0x00070001

#define LU62_DEALLOCATE_ABEND_SVC 0x00080000
#define LU62_DEALLOCATE_ABEND_SVC_BO 0x00080001
```

Code Example C-1 (30 of 39)

```
#define LU62_DEALLOCATE_ABEND_TIMER 0x00090000
#define LU62_DEALLOCATE_ABEND_TIMER_BO 0x00090001

#define LU62_DEALLOCATE_NORMAL 0x000A0000
#define LU62_DEALLOCATE_NORMAL_BO 0x000A0001

#define LU62_ENCRYPTION_NOT_SUPPORTED 0x000B0000

#define LU62_FMH_DATA_NOT_SUPPORTED 0x000C0000

#define LU62_MAP_EXECUTION_FAILURE 0x000D0000

#define LU62_MAP_NOT_FOUND 0x000E0000

#define LU62_MAPPING_NOT_SUPPORTED 0x000F0000

/*
 * Parameter Errors
 */
#define LU62_PARAMETER_ERROR 0x00100000
#define LU62_UNKNOWN_TP 0x00100001
#define LU62_UNKNOWN_LU 0x00100002
#define LU62_UNKNOWN_PARTNER_LU 0x00100003
#define LU62_UNKNOWN_MODE 0x00100004
#define LU62_NO_SECURITY_INFO 0x00100005

/*
 * Posting Not Active
 */
#define LU62_POSTING_NOT_ACTIVE 0x00110000

/*
 * Program Errors
 */
#define LU62_PROG_ERROR_NO_TRUNC 0x00120000

#define LU62_PROG_ERROR_TRUNC 0x00130000

#define LU62_PROG_ERROR_PURGING 0x00140000

/*
 * Program Parameter Check
```

Code Example C-1 (31 of 39)

```
 */
#define LU62_PARAMETER_CHECK 0x00150000

/* returned by API */
#define LU62_PORT_ID_UNKNOWN 0x00150001
#define LU62_CONV_ID_UNKNOWN 0x00150002
#define LU62_NULL_REQUEST 0x00150003
#define LU62_NULL_DATA 0x00150004
#define LU62_BUFFER_TOO_SMALL 0x00150005
#define LU62_TP_UNKNOWN 0x00150006
#define LU62_TP_NAME_REQD 0x00150007
#define LU62_TP_ID_REQD 0x00150008
#define LU62_TP_NOT_STARTED 0x00150009
#define LU62_LU_NAME_REQD 0x0015000A
#define LU62_MODE_NAME_REQD 0x0015000B
#define LU62_REMOTE_TP_NAME_REQD 0x0015000C
#define LU62_TP_ALREADY_REGISTERED 0x0015000D

#define LU62_BAD_TP_NAME 0x00150010
#define LU62_BAD_REMOTE_TP_NAME 0x00150011
#define LU62_BAD_LU_NAME 0x00150012
#define LU62_BAD_MODE_NAME 0x00150013
#define LU62_BAD_MAP_NAME 0x00150014
#define LU62_BAD_CONV_SUPPORT 0x00150015
#define LU62_BAD_CONV_TYPE 0x00150016
#define LU62_BAD_DEALLOCATE_TYPE 0x00150017
#define LU62_BAD_ENCRYPT_TYPE 0x00150018
#define LU62_BAD_FILL_TYPE 0x00150019
#define LU62_BAD_FLUSH_TYPE 0x0015001A
#define LU62_BAD_FMH_DATA_TYPE 0x0015001B
#define LU62_BAD_LENGTH 0x0015001C
#define LU62_BAD_LOCKS_TYPE 0x0015001D
#define LU62_BAD_LOG_DATA 0x0015001E
#define LU62_BAD_PIP_PRESENCE 0x0015001F
#define LU62_BAD_POST_CONTROL 0x00150020
#define LU62_BAD_PREP_TO_RCV_TYPE 0x00150021
#define LU62_BAD_PROCESSING_MODE 0x00150022
#define LU62_BAD_PROG_TYPE 0x00150023
#define LU62_BAD_RESOURCE_COUNT 0x00150024
#define LU62_BAD_RETURN_CONTROL 0x00150025
#define LU62_BAD_SECURITY 0x00150026
#define LU62_BAD_SECURITY_PROGRAM 0x00150027
```

Code Example C-1 (32 of 39)

```
#define LU62_BAD_USERID 0x00150028
#define LU62_BAD_PASSWD 0x00150029
#define LU62_BAD_PROFILE 0x0015002A
#define LU62_BAD_SYNC_LEVEL  0x0015002B
#define LU62_BAD_TEST_TYPE 0x0015002C
#define LU62_BAD_FORGET_TYPE 0x0015002D

#define LU62_PARAM_NOT_SUPPORTED 0x001500F0
#define LU62_LOG_DATA_NOT_SUPPORTED 0x001500F1
#define LU62_SYNCPT_NOT_SUPPORTED 0x001500F2

/* CPIC returns via lu62_errno */
#define LU62_SYM_DEST_UNKNOWN 0x00154001
#define LU62_SYM_DEST_ERROR 0x00154002
#define LU62_BAD_CONVERSATION_ID 0x00154003
#define LU62_BAD_ERROR_DIRECTION 0x00154004
#define LU62_BAD_RECEIVE_TYPE 0x00154005
#define LU62_BAD_SEND_TYPE 0x00154006

/* returned by LU62 Server */
#define LU62_RESOURCE_UNKNOWN 0x00158001
#define LU62_INCOMPATIBLE_VERB 0x00158002
#define LU62_BASIC_CONV_SUPPORT 0x00158003
#define LU62_SVC_MODES_INVALID 0x00158004
#define LU62_INCOMPATIBLE_SYNC_LEVEL 0x00158005
#define LU62_INVALID_LL_FIELD 0x00158006

/*
 * Program State Check
 */
#define LU62_PROGRAM_STATE_CHECK 0x00160000
#define LU62_NO_RSP_EXPECTED 0x00160001
#define LU62_VERB_IN_PROGRESS 0x00160002
#define LU62_NO_VERB_IN_PROGRESS 0x00160003
#define LU62_NO_TP_REGISTERED 0x00160004
#define LU62_SEND_INCOMPLETE 0x00160005
#define LU62_PIP_PENDING 0x00160006

/*
 * Resource Failure
 */
#define LU62_RESOURCE_FAILURE_NO_RETRY 0x00170000
```

Code Example C-1 (33 of 39)

```
#define LU62_RESOURCE_FAIL_NO_RETRY_BO 0x00170001

#define LU62_RESOURCE_FAILURE_RETRY 0x00180000
#define LU62_RESOURCE_FAIL_RETRY_BO 0x00180001

/*
 * SVC Errors
 */
#define LU62_SVC_ERROR_NO_TRUNC 0x00190000

#define LU62_SVC_ERROR_TRUNC 0x001A0000

#define LU62_SVC_ERROR_PURGING 0x001B0000

/*
 * Unsuccessful
 */
#define LU62_UNSUCCESSFUL 0x001C0000

#define LU62_SYSTEM_EVENT 0x00200000

/*
 * Additional COPR return codes
 */
#define LU62_OK_AS_SPECIFIED 0x00001001
#define LU62_OK_AS_NEGOTIATED 0x00001002
#define LU62_OK_FORCED 0x00001003

/* Program Parameter Checks */
#define LU62_BAD_DS_TYPE 0x00151001
#define LU62_BAD_RESPONSIBLE_LU 0x00151002
#define LU62_BAD_SESSION_ID 0x00151003
#define LU62_BAD_SESSION_LIMIT 0x00151004
#define LU62_BAD_MIN_CONWINNERS 0x00151005
#define LU62_BAD_NQ_LU_NAME 0x00151006

#define LU62_PROGRAM_NOT_PRIVILEGED 0x00151010
#define LU62_SESSION_ID_REQD 0x00151011
#define LU62_PLU_SESSION_LIMIT_NOT_ZERO 0x00151012
#define LU62_DRAIN_SOURCE_NO_REQD 0x00151013
#define LU62_NQ_LU_NAME_REQD 0x00151014
```

Code Example C-1 (34 of 39)

```
/* COPR codes */
#define LU62_ACTIVATION_FAILURE_NO_RETRY 0x10010000

#define LU62_ACTIVATION_FAILURE_RETRY 0x10020000
#define LU62_COMMAND_RACE_REJECT 0x10030000
#define LU62_MODE_SESSION_LIMIT_CLOSED 0x10040000
#define LU62_MODE_SESSION_LIMIT_EXCEEDED 0x10050000
#define LU62_MODE_SESSION_LIMIT_NOT_ZERO 0x10060000
#define LU62_MODE_SESSION_LIMIT_ZERO 0x10070000
#define LU62_SESSION_LIMIT_EXCEEDED 0x10080000
#define LU62_REQUEST_EXCEEDS_MAX_ALLOWED 0x10090000
#define LU62_UNRECOGNIZED_MODE_NAME 0x100A0000


/****************************************************************************
 *
 * The lu62_trace Facility
 *
 * Traces are output to a trace file, sunlu62l_$$ in the pwd.
 * When 1000 traces have accumulated, the trace file is saved as
 * sunlu62l_$$.1, and truncated.
 *
 * Traces are output if a bit set in the trace type matches the corresponding
 * bit in the external lu62_trace_flag.  Trace output is controlled by
 * the format parameter.
 *
 * The lu62_trace interface is as follows:
 *
 * lu62_trace(type, caller, statement, length, buffer, format)
 * unsigned type;- trace type selection, see below
 * char     *caller;    - calling routine
 * char     *statement; - header
 * int      length;- length of following buffer (n/a to STRING)
 * char     *buffer;- buffer
 * int      format;- output format, see below
 *
 * The lu62_trace_flag may be set by
 *
 * lu62_set_trace_flag(flag)
 * bit32    flag;
 *
 * To read lu62_trace_flag, use
```

Code Example C-1 (35 of 39)

```
 *
 * bit32 lu62_get_trace_flag()
 *
 *---------------------------------------------------------------------
 * Trace Type Codes
 *
 * Trace types are enabled by setting a bit in the lu62_trace_flag.
 * The top 8 bits are reserved for current (and future) API trace types.
 * The next 8 bits are reserved for Sun supplied Transaction Programs.
 * The bottom 16 bits are available to users.
 *---------------------------------------------------------------------
 */
#define LU62_API_BUFS 0x80000000/* trace buffers to/from API*/
#define LU62_API_ERROR 0x40000000/* trace API detected errors*/
#define LU62_API_INFO 0x20000000/* API informational trace*/
#define LU62_API_CALLS 0x10000000/* trace lu62 verb calls*/
#define LU62_CPIC_CALLS 0x08000000/* trace CPIC verb calls*/
#define LU62_CPIC_ERROR 0x04000000/* trace CPIC verb calls*/
#define LU62_USR_TRACE 0x00000001 /* user defined                 */


/*---------------------------------------------------------------------
 * Trace Output Format Codes
 *---------------------------------------------------------------------
 */
#define STRING1/* supplied buffer contains string    */
#define NO_FMT_DATA0/* hex dump of supplied buffer        */
#define ASCII_DATA2/* interpet supplied buffer as ASCII  */
#define EBCDIC_DATA3/* interpet supplied buffer as EBCDIC */


/***************************************************************************
 *
 * API Global Data Structures
 *
 ***************************************************************************
 */
extern bit32lu62_errno;/* LU62 library errors   */
extern bit32lu62_trace_flag;/* trace options     */
extern char*sunlu62_serv;/* SUNLU62 tcp service name*/
```

Code Example C-1 (36 of 39)

```
/*************************************************************************
 *
 * API Entry Points
 *
 *************************************************************************
 */

/* Trace facility */
extern bit32lu62_get_trace_flag();
extern voidlu62_set_trace_flag();
extern voidlu62_trace();
extern voidlu62_dump_buffer();

/*
 * Display verb return values.
 * The following routines are used by lu62_trace to display return values.
 *
 * Interface is as follows unless shown otherwise:
 *   tpi_dis_verb_return(rqp, rc)
 *   lu62_verb_t *rqp;
 *   int rc;
 *
 */
extern char *tpi_display_return();/* (bit32 retcode, int rc) */
extern char *tpi_dis_wait_return(); /* (bit32 conv_id, int rc) */
extern char *tpi_dis_notification();
/* (lu62_notification_header_t *nhp, bit8 *buf, bit32 rc) */

extern char *tpi_dis_abort_return();
extern char *tpi_dis_accept_return();
extern char *tpi_dis_allocate_return();
extern char *tpi_dis_confirm_return();
extern char *tpi_dis_confirmed_return();
extern char *tpi_dis_deallocate_return();
extern char *tpi_dis_display_local_lu_return();
extern char *tpi_dis_display_mode_return();
extern char *tpi_dis_display_remote_lu_return();
extern char *tpi_dis_display_tp_return();
extern char *tpi_dis_flush_return();
extern char *tpi_dis_get_attributes_return();
extern char *tpi_dis_listen_return();
extern char *tpi_dis_open_return();
```

Code Example C-1 (37 of 39)

```
extern char *tpi_dis_post_on_receipt_return();
extern char *tpi_dis_prep_for_syncpt_return();
extern char *tpi_dis_prep_to_receive_return();
extern char *tpi_dis_receive_return();
extern char *tpi_dis_register_tp_return();

extern char *tpi_dis_request_notification_return();
extern char *tpi_dis_request_to_send_return();
extern char *tpi_dis_send_data_return();
extern char *tpi_dis_send_ps_data_return();
extern char *tpi_dis_send_error_return();
extern char *tpi_dis_test_return();
extern char *tpi_dis_get_type_return();
extern char *tpi_dis_get_tp_properties_return();

/* Character conversion */
extern unsigned int      conv_ascii_to_ebcdic();
extern unsigned int      conv_ebcdic_to_ascii();
extern unsigned char *b_asc_to_ebc();
extern unsigned char *b_ebc_to_asc();
extern unsigned char *str_asc_to_ebc();
extern unsigned char *str_ebc_to_asc();
extern unsigned char *strn_asc_to_ebc();
extern unsigned char *strn_ebc_to_asc();

/* SUNLU6.2 Connection Control */
extern int lu62_close();
extern int lu62_get_readfds();
extern int lu62_set_service_name();
extern int lu62_open();
extern int lu62_post();
extern int lu62_set_processing_mode();
extern int lu62_set_post_control();
extern int lu62_wait_conversation();
extern int lu62_wait_server();

/* Type Independent Verbs */
extern int lu62_abort();
extern int lu62_accept();
extern int lu62_listen();
extern int lu62_register_tp();
extern int lu62_get_tp_properties();
```

Code Example C-1 (38 of 39)

```
extern int lu62_get_type();
extern int lu62_wait();

/* Basic Conversation Verbs */
extern int lu62_allocate();
extern int lu62_confirm();

extern int lu62_confirmed();
extern int lu62_deallocate();
extern int lu62_flush();
extern int lu62_get_attributes();
extern int lu62_post_on_receipt();
extern int lu62_prep_to_receive();
extern int lu62_receive_and_wait();
extern int lu62_receive_immediate();
extern int lu62_request_to_send();
extern int lu62_send_data();
extern int lu62_send_ps_data();/* Sync Level syncpoint addition */
extern int lu62_send_error();
extern int lu62_test();
extern int lu62_test_rts_received();

/* Mapped Conversation Verbs */
extern int lu62_mc_allocate();
extern int lu62_mc_confirm();
extern int lu62_mc_confirmed();
extern int lu62_mc_deallocate();
extern int lu62_mc_flush();
extern int lu62_mc_get_attributes();
extern int lu62_mc_post_on_receipt();
extern int lu62_mc_prep_to_receive();
extern int lu62_mc_receive_and_wait();
extern int lu62_mc_receive_immediate();
extern int lu62_mc_request_to_send();
extern int lu62_mc_send_data();
extern int lu62_mc_send_error();
extern int lu62_mc_test();
extern int lu62_mc_test_rts_received();

/* CNOS Verbs */
extern int lu62_change_session_limit();
extern int lu62_initialize_session_limit();
```

Code Example C-1 (39 of 39)

```
extern int lu62_reset_session_limit();

/* Session Control Verbs */
extern int lu62_activate_session();
extern int lu62_deactivate_conv_group();
extern int lu62_deactivate_session();


/* LU Definition Verbs */
extern int lu62_define_local_lu();
extern int lu62_define_mode();
extern int lu62_define_remote_lu();
extern int lu62_define_tp();
extern int lu62_delete();
extern int lu62_display_local_lu();
extern int lu62_display_mode();
extern int lu62_display_remote_lu();
extern int lu62_display_tp();

extern int lu62_request_notification();
extern int lu62_stop_notification();
extern int lu62_poll_notification();
extern int lu62_receive_notification();

#ifndef BRX_ALLOC_H
#define BRX_ALLOC_H
extern char*sun_malloc();  /* (unsigned size) */
extern char*sun_calloc();/* (unsigned nelem, unsigned size) */
extern int sun_free();/* (char *buf) */
extern bit16    sun_get_bit16();/* (bit8 *bp) */
extern bit32    sun_get_bit32();/* (bit8 *bp) */
extern void     sun_set_bit16();/* (bit16 val, bit8 *bp) */
extern void     sun_set_bit32();/* (bit32 val, bit8 *bp) */
#endif /* BRX_ALLOC_H */

#endif /* SUNLU62_H */
```

Sample LU6.2 Programs D

Code Example D-1 (1 of 32)

```
/*
 * THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT
 * NOTICE AND SHOULD NOT BE CONSIDERED AS A COMMITMENT BY Sun
 * SYSTEMS.
 */

/******************************************************************************
*
* Prog Name:    tp_sr.c
*
* Description:
*
*   SunCPIC Sample Program - send and receive.  This TP is partnered
*   with tp_rs.c.
*
* Usage:
*
*       tp_sr [-h <server> -l <local_lu> -r <partner_lu> -m <mode_name>
*              -p <remote_tp> -c -t <trace_flag>]
*
*       where,
*
*       -h <server>     identifies the LU62 Server host.  Default is the
*                       local host.
```

Code Example D-1 (2 of 32)

```
*
*       -l <local_lu>   local LU name
*
*       -r <partner_lu> Partner LU name
*
*       -m <mode_name>  Mode name
*
*       -p <remote_tp>  remote TP name
*
*   -c      Send and Confirm (default is Send and Flush)
*
*       -t <trace_flag> trace options word.
*
* Creation Date: 03/15/92
*
* Change Log:
*
*****************************************************************************/

#include <stdio.h>
#include <signal.h>
#include <ctype.h>
#include <string.h>
#include <errno.h>
#include <sys/param.h>
#include <sys/types.h>
#include <netinet/in.h>

#include "sunlu62.h"


/************************************************************************
 *
 * Program constants and globals
 *
 ************************************************************************
 */
#define MAX_SND_MSG_SIZE    4096    /* MAX size of transmitted messages     */
#define MAX_RCV_MSG_SIZE    4096    /* MAX size of received messages        */

typedef enum {
    RECEIVER = 0,
```

Code Example D-1 (3 of 32)

```
    SENDER,
    DEALLOCATED
} prog_state_e;

char    *Prog_name;
char    *Host = ""; /* LU62 Server host */

char    *LocalLUName   = "LUA";

char    *TPName        = "TPB";
char    *PartnerLUName = "PLUB";
char    *ModeName      = "MODEAB";
char    *UserId        = "username";
char    *Password      = "password";
char    *Profile       = "group";

lu62_processing_mode_e Processing_Mode= PM_BLOCKING;
lu62_return_control_e Return_Control= RC_WHEN_SESSION_ALLOCATED;
lu62_conv_type_e Conv_Type = CONVERSATION_BASIC;
lu62_sync_level_e Sync_Level= SYNC_LEVEL_NONE;
lu62_security_e Security= SECURITY_NONE;

bit32   Trace_flag;

/* send and receive buffers are malloc'ed during initialization */
static char     *rbuffer;
static char     *sbuffer;

typedef struct ll_record
    bit16 len;
    bit16 seq_num;
    char  data[81];
ll_record_t;

static ll_record_t LL_buf1 = { 0, 0,
"123456789012345678901234567890123456789012345678901234567890123456789012345678\n"};
#define LL_BUF1_LEN     84

static ll_record_t LL_buf2 = \n"}; = { 0, 0,
"ABCDEFGHIJKLMNOPQRSTUVWXYZ
#define LL_BUF2_LEN     31
```

Code Example D-1 (4 of 32)

```
/*----------------------------------------------------------------------
 * session
 *----------------------------------------------------------------------
 */
void
session()
{
    prog_state_e prog_state;
    ll_record_t *recv_buf, *send_buf;
    int send_len;
    int rc;
    bit16 send_seq_num = 0;

    static lu62_open_req_t     open_req      = { 0, 0};
    static lu62_close_req_t    close_req     = { 0, 0};
    static lu62_allocate_t     alloc_req     = { 0, 0};
    static lu62_confirm_t      cfm_req       = { 0, 0};
    static lu62_confirmed_t    cfmd_req      = { 0, 0};
    static lu62_deallocate_t   deall_req     = { 0, 0};
    static lu62_flush_t        flush_req     = { 0, 0};
    static lu62_send_data_t    send_data_req = { 0, 0};
    static lu62_receive_t      recv_data_req = { 0, 0};

    rbuffer = (char *)malloc(MAX_RCV_MSG_SIZE);
    sbuffer = (char *)malloc(MAX_SND_MSG_SIZE);

    /* SENDER set up */

    prog_state = SENDER;

    /* convert bit16 fields to network order */
    LL_buf1.len = htons(LL_BUF1_LEN);
    LL_buf2.len = htons(LL_BUF2_LEN);

    bcopy(&LL_buf1, sbuffer, LL_BUF1_LEN);
    bcopy(&LL_buf2, sbuffer + LL_BUF1_LEN, LL_BUF2_LEN);
    send_len = LL_BUF1_LEN + LL_BUF2_LEN;
    send_buf = (ll_record_t *)sbuffer;

    rc = open_lu(&open_req);
    if (rc == LU62_ERROR)
        exit(1);
```

Code Example D-1 (5 of 32)

```
    /* establish LU port context for upcoming verbs */
    close_req.port_id = open_req.port_id;
    alloc_req.port_id = open_req.port_id;

    rc = allocate_conv(&alloc_req);
    if (rc == LU62_ERROR)
        exit(1);

    /* display attributes */
    rc = get_attributes(alloc_req.conv_id);
    if (rc == LU62_ERROR)
        exit(1);

    /* display tp_properties */
    rc = get_tp_properties(alloc_req.conv_id);
    if (rc == LU62_ERROR)
        exit(1);

    /* establish conversation context for upcoming verbs */
    cfm_req.conv_id       = alloc_req.conv_id;
    cfmd_req.conv_id      = alloc_req.conv_id;
    deall_req.conv_id     = alloc_req.conv_id;
    flush_req.conv_id     = alloc_req.conv_id;
    send_data_req.conv_id = alloc_req.conv_id;
    recv_data_req.conv_id = alloc_req.conv_id;

    /* RECEIVER set up */
    recv_buf = (ll_record_t *)rbuffer;

    while (rc == LU62_OK
       &&  prog_state != DEALLOCATED) {
        switch (prog_state) {

        case RECEIVER:
            bzero(rbuffer, MAX_RCV_MSG_SIZE);
            recv_data_req.data = (bit8 *)recv_buf;
            recv_data_req.length = MAX_RCV_MSG_SIZE;
            rc = receive_ll(&recv_data_req);

            if (rc == LU62_OK) {

                switch (recv_data_req.what_received) {
```

Code Example D-1 (6 of 32)

```
                case WR_CONFIRM:
                    printf("CONFIRM_RECEIVED \n");
                    rc = confirmed_conv(&cfmd_req);
                    break;
                case WR_SEND:
                    printf("SEND_RECEIVED \n");
                    prog_state = SENDER;
                    break;
                case WR_CONFIRM_SEND:
                    printf("CONFIRM_SEND_RECEIVED \n");
                    rc = confirmed_conv(&cfmd_req);
                    prog_state = SENDER;
                    break;
                case WR_CONFIRM_DEALLOCATE:
                    printf("CONFIRM_DEALLOC_RECEIVED
                    rc = confirmed_conv(&cfmd_req);
                    prog_state = DEALLOCATED;
                    break;
               }
            }
            break;

        case SENDER:
            send_seq_num += 1;
            send_buf->seq_num = htons(send_seq_num);
            send_data_req.data = (bit8 *)send_buf;
            send_data_req.length = send_len;
            rc = send_data_conv(&send_data_req);
            if (rc == LU62_OK)
                if (Sync_Level == SYNC_LEVEL_CONFIRM)
                    rc = confirm_conv(&cfm_req);
            prog_state = RECEIVER;
            break;

        case DEALLOCATED:
            break;
        }
    }

    close_lu(&close_req);
}
```

Code Example D-1 (7 of 32)

```
/*-----------------------------------------------------------------------
 * usage
 *-----------------------------------------------------------------------
 */
usage()
{
    printf("usage: tp_sr [options]
    printf("options are:
    printf("    -h<server          = %s>\n", Host);
    printf("    -l<lu_name         = %s>\n", LocalLUName);
    printf("    -r<partner_lu_name = %s>\n", PartnerLUName);
    printf("    -m<mode_name       = %s>\n", ModeName);
    printf("    -p<tp_name         = %s>\n", TPName);
    printf("    -c (send and confirm)\n")’
    printf("    -t (trace) = %x\n", Trace_flag);
    exit(1);

}

/*-----------------------------------------------------------------------
 * main
 *-----------------------------------------------------------------------
 */
main(argc, argv)
int  argc;
char *argv[];
{
    extern char *optarg;
    extern int optind;
    char c;

    int errflg = 0;

    char *cp;
    long strtol();

    signal(SIGPIPE, SIG_IGN);

    /* default is no tracing */
    Trace_flag = 0;

    /*
```

Code Example D-1 (8 of 32)

```
     * Process command line args
     * minimal error checking!
     */
    Prog_name = argv[0];
    for (; cp = strchr(Prog_name, '/'); Prog_name = ++cp);

    while((c = getopt(argc, argv, "h:l:r:m:p:ct:")) != EOF)
        switch(c) {

        case 'h':      /* server host */
            Host = optarg;
            break;
        case 'l':
            LocalLUName = optarg;
            break;
        case 'r':
            PartnerLUName = optarg;
            break;
        case 'm':
            ModeName = optarg;
            break;
        case 'p':
            TPName = optarg;
            break;
        case 'c':
            Sync_Level = SYNC_LEVEL_CONFIRM;
            break;
        case 't':      /* API tracing options */
            Trace_flag = strtol(optarg, (char **) NULL, 0);
            break;
        case '?':      /* help: doesn't work too well with csh */
            errflg++;
            break;
        default:       /* the rest are errors */
            errflg++;
            break;
   }

    if (errflg) {
        usage();
   }
```

Code Example D-1 (9 of 32)

```
    lu62_set_trace_flag(Trace_flag);

    session();

    exit(0);
}


/*
 * THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT
 * NOTICE AND SHOULD NOT BE CONSIDERED AS A COMMITMENT BY Sun
 * SYSTEMS.
 */

/****************************************************************************
*
* Prog Name:    tp_rs.c
*
* Description:
*
*   SunCPIC Sample Program - receive and send.  This TP is partnered
*   with tp_sr.c.
*
* Usage:
*
*       tp_rs [-h <server> -l <local_lu> -p <local_tp> -t <trace_flag>]
*
*       where,
*
*       -h <server>     identifies the LU62 Server host.  Default is the
*                       local host.
*
*       -l <local_lu>   local LU name
*
*       -p <local_tp>   TP name to register for incoming attach
*
*       -r <res_id>resource id to use as own TP instance in accept
*
*       -t <trace_flag> trace options word.
*
* Creation Date: 03/15/92
*
```

Code Example D-1 (10 of 32)

```
* Change Log:
*
*****************************************************************************/

#include <stdio.h>
#include <signal.h>
#include <ctype.h>
#include <string.h>
#include <errno.h>
#include <sys/param.h>
#include <sys/types.h>
#include <netinet/in.h>

#include "sunlu62.h"


/************************************************************************
 *
 * Program constants and globals
 *
 ************************************************************************
 */
#define MAX_SND_MSG_SIZE    4096    /* MAX size of transmitted messages     */
#define MAX_RCV_MSG_SIZE    4096    /* MAX size of received messages        */

typedef enum {
    RECEIVER = 0,
    SENDER,
    DEALLOCATED
} prog_state_e;

char    *Prog_name;
char    *Host = ""; /* LU62 Server host */

char    *LocalLUName   = "LUB";

char    *TPName        = "TPB";
char    *PartnerLUName;
char    *ModeName;
char    *UserId;
char    *Password;
```

Code Example D-1 (11 of 32)

```
bit32   OwnTPInstance  = 0;

lu62_processing_mode_e Processing_Mode= PM_BLOCKING;
lu62_return_control_e Return_Control= RC_WHEN_SESSION_ALLOCATED;
lu62_conv_type_e Conv_Type;
lu62_sync_level_e Sync_Level;
lu62_security_e Security;

bit32   Trace_flag;

/* send and receive buffers are malloc'ed during initialization */
static char     *rbuffer;
static char     *sbuffer;

typedef struct ll_record {
    bit16 len;
    bit16 seq_num;
    char  data[81];
} ll_record_t;

static ll_record_t LL_buf1 = { 0, 0
"123456789012345678901234567890123456789012345678901234567890123456789012345678\n"};
#define LL_BUF1_LEN     84

static ll_record_t LL_buf2 = { 31, 0.
"ABCDEFGHIJKLMNOPQRSTUVWXYZ\n"};
#define LL_BUF2_LEN     31


/*---------------------------------------------------------------------
 * session
 *---------------------------------------------------------------------
 */
void
session()
{
    prog_state_e prog_state;
    ll_record_t *recv_buf, *send_buf;
    int send_len;
    int rc;
    bit16 send_seq_num = 0;
```

Code Example D-1 (12 of 32)

```
    static lu62_open_req_t     open_req      = {0,};
    static lu62_close_req_t    close_req     = {0,};
    static lu62_register_tp_t  register_req  = {0,};
    static lu62_accept_t       accept_req    = {0,};
    static lu62_get_attributes_t getattr_req = {0,};
    static lu62_confirm_t      cfm_req       = {0,};
    static lu62_confirmed_t    cfmd_req      = {0,};
    static lu62_deallocate_t   deall_req     = {0,};
    static lu62_flush_t        flush_req     = {0,};
    static lu62_send_data_t    send_data_req = {0,};
    static lu62_receive_t      recv_data_req = {0,};

    rbuffer = (char *)malloc(MAX_RCV_MSG_SIZE);
    sbuffer = (char *)malloc(MAX_SND_MSG_SIZE);

    rc = open_lu(&open_req);
    if (rc == LU62_ERROR)
        exit(1);

    /* establish LU port context for upcoming verbs */
    close_req.port_id     = open_req.port_id;
    accept_req.port_id    = open_req.port_id;

    if (OwnTPInstance) {
        accept_req.own_tp_instance = OwnTPInstance;
    }
    else {
        register_req.port_id  = open_req.port_id;
        rc = register_tp(&register_req, TPName);
        if (rc == LU62_ERROR)
            exit(1);

    }
    rc = accept_conv(&accept_req);
    if (rc == LU62_ERROR)
        exit(1);

    /* display conversation type */
    rc = get_conv_type(accept_req.conv_id);
    if (rc == LU62_ERROR)
        exit(1);
```

Code Example D-1 (13 of 32)

```
    /* display attributes */
    rc = get_attributes(accept_req.conv_id);
    if (rc == LU62_ERROR)
        exit(1);

    /* display tp_properties */
    rc = get_tp_properties(accept_req.conv_id);
    if (rc == LU62_ERROR)
        exit(1);

    /* determine sync level */
    getattr_req.conv_id = accept_req.conv_id;
    rc = lu62_get_attributes(&getattr_req);
    if (rc == LU62_ERROR)
        exit(1);
    Sync_Level = getattr_req.sync_level;

    /* establish conversation context for upcoming verbs */
    cfm_req.conv_id       = accept_req.conv_id;
    cfmd_req.conv_id      = accept_req.conv_id;
    deall_req.conv_id     = accept_req.conv_id;
    flush_req.conv_id     = accept_req.conv_id;
    send_data_req.conv_id = accept_req.conv_id;
    recv_data_req.conv_id = accept_req.conv_id;

    /* RECEIVER set up */
    prog_state = RECEIVER;
    recv_buf = (ll_record_t *)rbuffer;

    /* SENDER set up */

    /* convert bit16 fields to network order */
    LL_buf1.len = htons(LL_BUF1_LEN);
    LL_buf2.len = htons(LL_BUF2_LEN);

    bcopy(&LL_buf1, sbuffer, LL_BUF1_LEN);
    bcopy(&LL_buf2, sbuffer + LL_BUF1_LEN, LL_BUF2_LEN);
    send_len = LL_BUF1_LEN + LL_BUF2_LEN;
    send_buf = (ll_record_t *)sbuffer;

    while (rc == LU62_OK
       &&  prog_state != DEALLOCATED) {
```

Code Example D-1 (14 of 32)

```
        switch (prog_state) {

        case RECEIVER:
            bzero(rbuffer, MAX_RCV_MSG_SIZE);
            recv_data_req.data = (bit8 *)recv_buf;
            recv_data_req.length = MAX_RCV_MSG_SIZE;
            rc = receive_ll(&recv_data_req);

            if (rc == LU62_OK) {

                switch (recv_data_req.what_received) {
                case WR_CONFIRM:
                    printf("CONFIRM_RECEIVED
                    rc = confirmed_conv(&cfmd_req);
                    break;
                case WR_SEND:
                    printf("SEND_RECEIVED
                    prog_state = SENDER;
                    break;
                case WR_CONFIRM_SEND:
                    printf("CONFIRM_SEND_RECEIVED
                    rc = confirmed_conv(&cfmd_req);
                    prog_state = SENDER;
                    break;
                case WR_CONFIRM_DEALLOCATE:
                    printf("CONFIRM_DEALLOC_RECEIVED
                    rc = confirmed_conv(&cfmd_req);
                    prog_state = DEALLOCATED;
                    break;
          }
        }
            break;

        case SENDER:
            send_seq_num += 1;
            send_buf->seq_num = htons(send_seq_num);
            send_data_req.data = (bit8 *)send_buf;
            send_data_req.length = send_len;
            rc = send_data_conv(&send_data_req);
            if (rc == LU62_OK)
                if (Sync_Level == SYNC_LEVEL_CONFIRM)
                    rc = confirm_conv(&cfm_req);
```

Code Example D-1 (15 of 32)

```
            prog_state = RECEIVER;
            break;

        case DEALLOCATED:
            break;
        }
    }

    close_lu(&close_req);
   }


/*------------------------------------------------------------------------
 * usage
 *------------------------------------------------------------------------
 */
usage()

    printf("usage: tp_rs [options]
    printf("options are:
    printf("    -h<server = %s>, Host); %s>\n",
    printf("    -l<lu_name = %s>, LocalLUName); %s>\n",
    printf("    -p<tp_name = %s>, TPName); %s>\n",
    printf("    -r<res_id  = %d>, OwnTPInstance); %s>\n",
    printf("    -t (trace) = %x, Trace_flag); %s>\n",
    exit(1);



/*------------------------------------------------------------------------
 * main
 *------------------------------------------------------------------------
 */
main(argc, argv)
int  argc;
char *argv[];

    extern char *optarg;
    extern int optind;
    char c;

    int errflg = 0;
```

Code Example D-1   (16 of 32)

```
    char *cp;
    long strtol();

    signal(SIGPIPE, SIG_IGN);

    /* default is no tracing */
    Trace_flag = 0;

    /*
     * Process command line args
     * minimal error checking!
     */
    Prog_name = argv[0];
    for (; cp = strchr(Prog_name, '/'); Prog_name = ++cp);

    while((c = getopt(argc, argv, "h:l:p:r:t:")) != EOF)
        switch(c) {

        case 'h':      /* server host */
            Host = optarg;
            break;
        case 'l':
            LocalLUName = optarg;
            break;
        case 'p':
            TPName = optarg;
            break;
        case 'r':
            OwnTPInstance = strtol(optarg, (char **) NULL, 0);
            break;
        case 't':      /* API tracing options */
            Trace_flag = strtol(optarg, (char **) NULL, 0);
            break;
        case '?':      /* help: doesn't work too well with csh */
            errflg++;
            break;
        default:       /* the rest are errors */
            errflg++;
            break;
        }

    if (errflg)
```

Code Example D-1 (17 of 32)

```
        usage();


    lu62_set_trace_flag(Trace_flag);

    session();

    exit(0);
 }


/*
 * THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT
 * NOTICE AND SHOULD NOT BE CONSIDERED AS A COMMITMENT BY Sun
 * SYSTEMS.
 */

/****************************************************************************
*
* Prog Name:    tp_calls.c
*
* Description: SunLU62 API Sample Program Calls
*
*       Simple wrap-around routines are provided for the LU6.2
*       basic verbs.  These routines trace entry and exit.  For
*       all blocking verbs, a request structure is passed so that,
*       in non-blocking mode, the same operation can be outstanding
*       on multiple conversations.
*
*       Exit tracing is performed using tpi_dis routines, which are
*       not normally exposed to the user.
*
* Creation Date: 3/17/91
*
* Change Log:
*
****************************************************************************/

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <signal.h>
```

Code Example D-1 (18 of 32)

```
#include <errno.h>
#include <sys/param.h>
#include <sys/types.h>

#include "sunlu62.h"


/************************************************************************
 *
 * Program constants and globals
 *
 ************************************************************************
 */
extern char *Prog_name;
extern char *Host;      /* LU62 Server host                     */

extern char *LocalLUName;
extern char *PartnerLUName;
extern char *ModeName;
extern char *TPName;
extern char *UserId;
extern char *Password;

extern lu62_processing_mode_e  Processing_Mode;
extern lu62_return_control_e   Return_Control;
extern lu62_conv_type_e        Conv_Type;
extern lu62_sync_level_e       Sync_Level;
extern lu62_security_e         Security;

extern bit32 Trace_flag;


/*----------------------------------------------------------------------
 * get_attributes
 *----------------------------------------------------------------------
 */
int
get_attributes(conv_id)
bit32 conv_id;
{
    int rc;
    lu62_get_attributes_t attrib;
```

Code Example D-1 (19 of 32)

```
    printf("\n0x%x: lu62_get_attributes:\n", conv_id);

    attrib.conv_id = conv_id;
    rc = lu62_get_attributes(&attrib);

    printf(tpi_dis_get_attributes_return(&attrib, rc));

    return(rc);



/*-------------------------------------------------------------------------
 * get_conv_type
 *------------------------------------------------------------------------
 */
int
get_conv_type(conv_id)
bit32 conv_id;
{
    int rc;
    lu62_get_type_t conv_type;

    printf("\n0x%x: lu62_get_type:\n", conv_id);

    conv_type.conv_id = conv_id;
    rc = lu62_get_type(&conv_type);

    printf(tpi_dis_get_type_return(&conv_type, rc));

    return(rc);
}


/*-------------------------------------------------------------------------
 * get_tp_properties
 *------------------------------------------------------------------------
 */
int
get_tp_properties(conv_id)
bit32 conv_id;
{
    int rc;
```

Code Example D-1 (20 of 32)

```
    lu62_get_tp_properties_t props;

    printf("\n0x%x: lu62_get_tp_properties:\n", conv_id);

    props.conv_id = conv_id;
    rc = lu62_get_tp_properties(&props);

    printf(tpi_dis_get_tp_properties_return(&props, rc));

    return(rc);
 }


/*----------------------------------------------------------------------
 * open_lu
 *----------------------------------------------------------------------
 */
int
open_lu(rqp)
lu62_open_req_t *rqp;
{
    int rc;

    printf("\nlu62_open %s:\n", LocalLUName);

    /*
     * Set-up for lu62_open.
     *
     */
    strncpy(rqp->host, Host, MAXHOSTNAMELEN);
    strncpy(rqp->lu_name, LocalLUName, LU62_LU_NAME_LEN);

    rqp->processing_mode = Processing_Mode;

    rc = lu62_open(rqp);

    printf(tpi_dis_open_return(rqp, rc));

    return(rc);
}
```

Code Example D-1 (21 of 32)

```
/*----------------------------------------------------------------------
 * close_lu
 *----------------------------------------------------------------------
 */
int
close_lu(rqp)
lu62_close_req_t *rqp;
{
    int rc;

    printf("\nlu62_close on LU 0x%x:\n", rqp->port_id);

    rc = lu62_close(rqp);

    printf(tpi_display_return(rqp->return_code, rc));

    return(rc);
}


/*----------------------------------------------------------------------
 * register_tp
 *----------------------------------------------------------------------
 */
int
register_tp(rqp, tp_name)
lu62_register_tp_t *rqp;
char *tp_name;
{
    int rc;

    printf("\n lu_register_tp %s on LU 0x%x:\n", tp_name, rqp->port_id);

    /*
     * Set-up for lu62_register_tp.
     *
     */
    bcopy(tp_name, rqp->tp_name, LU62_TP_NAME_LEN);
    rc = lu62_register_tp(rqp);
    if (rc == LU62_ERROR)
        error_string("lu62_register_tp", rqp->return_code, rc);
```

Code Example D-1 (22 of 32)

```
    return(rc);
 }


/*----------------------------------------------------------------------
 * listen_conv
 *----------------------------------------------------------------------
 */
int
listen_conv(rqp)
lu62_listen_t *rqp;
{
    int rc;

    printf("\n lu_listen LU 0x%x:\n", rqp->port_id);

    rc = lu62_listen(rqp);

    printf(tpi_dis_listen_return(rqp, rc));

    return(rc);
}


/*----------------------------------------------------------------------
 * accept_conv
 *----------------------------------------------------------------------
 */
int
accept_conv(rqp)
lu62_accept_t *rqp;
{
    int rc;

    printf("\n lu_accept on LU 0x%x:\n", rqp->port_id);

    rc = lu62_accept(rqp);

    printf(tpi_dis_accept_return(rqp, rc));

    return(rc);
}
```

Code Example D-1 (23 of 32)

```
/*----------------------------------------------------------------------
 * allocate_conv
 *----------------------------------------------------------------------
 */
int
allocate_conv(rqp)
lu62_allocate_t *rqp;
{
    int rc;

    printf("\n lu_allocate on LU 0x%x:\n", rqp->port_id);

    /*
     * Set-up for lu62_allocate.
     *
     */
    strncpy(rqp->remote_tp_name, TPName, LU62_TP_NAME_LEN);
    strncpy(rqp->lu_name, PartnerLUName, LU62_LU_NAME_LEN);
    strncpy(rqp->mode_name, ModeName, LU62_MODE_NAME_LEN);

    rqp->type           = Conv_Type;
    rqp->flush          = FLUSH_YES;
    rqp->return_control = Return_Control;
    rqp->sync_level     = Sync_Level;
    rqp->security       = Security;
    if (Security == SECURITY_PROGRAM)
        strncpy(rqp->user_id, UserId, LU62_MAX_USER_ID_LEN);
        strncpy(rqp->passwd,  Password, LU62_MAX_PASSWD_LEN);


    rc = lu62_allocate(rqp);

    printf(tpi_dis_allocate_return(rqp, rc));

    return(rc);
}


/*----------------------------------------------------------------------
 * abort_conv
 *----------------------------------------------------------------------
```

Code Example D-1 (24 of 32)

```
 */
int
abort_conv(rqp)
lu62_abort_t *rqp;
{
    int rc;

    printf("\n0x%: lu62_abort:\n", rqp->conv_id);

    rc = lu62_abort(rqp);

    printf(tpi_dis_abort_return(rqp, rc));

    return(rc);
}


/*-----------------------------------------------------------------------
 * confirm_conv
 *-----------------------------------------------------------------------
 */
int
confirm_conv(rqp)
lu62_confirm_t *rqp;
{
    int rc;

    printf("\n0x%x: lu62_confirm:\n", rqp->conv_id);

    rc = lu62_confirm(rqp);

    printf(tpi_dis_confirm_return(rqp, rc));

    return(rc);
}


/*-----------------------------------------------------------------------
 * confirmed_conv
 *-----------------------------------------------------------------------
 */
int
```

Code Example D-1 (25 of 32)

```
confirmed_conv(rqp)
lu62_confirmed_t *rqp;
{
    int rc;

    printf("\n0x%x: lu62_confirmed:\n", rqp->conv_id);

    rc = lu62_confirmed(rqp);

    printf(tpi_dis_confirmed_return(rqp, rc));

    return(rc);
}


/*-----------------------------------------------------------------------
 * deallocate_conv
 *-----------------------------------------------------------------------
 */
int
deallocate_conv(rqp)
lu62_deallocate_t *rqp;
{
    int rc;

    printf("\n0x%x: lu62_deallocate:\n", rqp->conv_id);

    rc = lu62_deallocate(rqp);

    printf(tpi_dis_deallocate_return(rqp, rc));

    return(rc);
}


/*-----------------------------------------------------------------------
 * flush_conv
 *-----------------------------------------------------------------------
 */
int
flush_conv(rqp)
lu62_flush_t *rqp;
```

Code Example D-1 (26 of 32)

```
{
    int rc;

    printf("\n0x%x: lu62_flush:\n", rqp->conv_id);

    rc = lu62_flush(rqp);

    printf(tpi_dis_flush_return(rqp, rc));

    return(rc);
}


/*----------------------------------------------------------------------
 * post_conv
 *----------------------------------------------------------------------
 */
int
post_conv(rqp)
lu62_post_on_receipt_t *rqp;
{
    int rc;

    printf("\0x%x: lu62_post_on_receipt:\n", rqp->conv_id);

    rc = lu62_post_on_receipt(rqp);

    printf(tpi_dis_post_on_receipt_return(rqp, rc));

    return(rc);
}


/*----------------------------------------------------------------------
 * prep_to_receive_conv
 *----------------------------------------------------------------------
 */
int
prep_to_receive_conv(rqp)
lu62_prep_to_receive_t *rqp;
{
    int rc;
```

Code Example D-1 (27 of 32)

```
    printf("\nx%x: lu62_prep_to_receive:\n", rqp->conv_id);

    rc = lu62_prep_to_receive(rqp);

    printf(tpi_dis_prep_to_receive_return(rqp, rc));

    return(rc);

}

/*------------------------------------------------------------------------
 * request_to_send_conv
 *------------------------------------------------------------------------
 */
int
request_to_send_conv(rqp)
lu62_request_to_send_t *rqp;
{
    int rc;

    printf("\n0x%x: lu62_request_to_send:\n", rqp->conv_id);

    rc = lu62_request_to_send(rqp);

    printf(tpi_dis_request_to_send_return(rqp, rc));

    return(rc);
}


/*------------------------------------------------------------------------
 * send_data_conv
 *------------------------------------------------------------------------
 */
int
send_data_conv(rqp)
lu62_send_data_t *rqp;
{
    int rc;

    printf("\n0x%x: lu62_send_data:\n", rqp->conv_id);
```

Code Example D-1   (28 of 32)

```
    rc = lu62_send_data(rqp);

    printf(tpi_dis_send_data_return(rqp, rc));

    return(rc);
}


/*----------------------------------------------------------------------
 * send_error_conv
 *----------------------------------------------------------------------
 */
int
send_error_conv(rqp)
lu62_send_error_t *rqp;
{
    int rc;

    printf("\n0x%x: lu62_send_error:\n", rqp->conv_id);

    rc = lu62_send_error(rqp);

    printf(tpi_dis_send_error_return(rqp, rc));

    return(rc);
}


/*----------------------------------------------------------------------
 * test_posted_conv
 *----------------------------------------------------------------------
 */
int
test_posted_conv(rqp)
lu62_test_t *rqp;
{
    int rc;

    printf("\n0x%x: lu62_test:\n", rqp->conv_id);

    rqp->test = TEST_POSTED;
```

Code Example D-1   (29 of 32)

```
    rc = lu62_test(rqp);

    printf(tpi_dis_test_return(rqp, rc));

    return(rc);
}


/*---------------------------------------------------------------------
 * test_rts_conv
 *---------------------------------------------------------------------
 */
int
test_rts_conv(rqp)
lu62_test_t *rqp;
{
    int rc;

    printf("\n0x%x: lu62_test:\n", rqp->conv_id);

    rqp->test = TEST_REQUEST_TO_SEND_RECEIVED;

    rc = lu62_test(rqp);

    printf(tpi_dis_test_return(rqp, rc));

    return(rc);
}


/*---------------------------------------------------------------------
 * wait_all
 *
 * Wait for all incomplete verbs to complete
 *---------------------------------------------------------------------
 */
int
wait_all(timeout)
struct timeval *timeout;
{
    int rc;
    int conv_id;
```

Code Example D-1 (30 of 32)

```
    printf("_wait_server:

    while (1) {
        rc = lu62_wait_server(timeout, &conv_id);
        printf(tpi_dis_wait_return(conv_id, rc));

        if (rc == LU62_ERROR)
            break;
    }

    return(rc);
}


/*----------------------------------------------------------------------
 * receive_ll
 *
 * RECEIVE_AND_WAIT to get the next logical record
 *----------------------------------------------------------------------
 */
int
receive_ll(rqp)
lu62_receive_t *rqp;
{
    int rc;
    bit16 ll = 0;
    bit8 *buffer;
    int buflen, len;

    printf("\n0x%x:: lu62_receive_and_wait:\n", rqp->conv_id);

    /*
     * Set-up for lu62_receive_and_wait.
     *
     */
    rqp->fill = FILL_LL;
    buflen = rqp->length;
    buffer = rqp->data;

    rc = lu62_receive_and_wait(rqp);
    printf(tpi_dis_receive_return(rqp, rc));
```

Code Example D-1 (31 of 32)

```
    if ((rqp->return_code == LU62_OK)
    && ((  rqp->what_received == WR_DATA_COMPLETE)
       || (rqp->what_received == WR_DATA_INCOMPLETE))) {
        lu62_dump_buffer((FILE *)stdout, rqp->length, buffer, ASCII_DATA);
    }

    return(rc);
}


/*-----------------------------------------------------------------------
 * receive_buffer
 *
 * RECEIVE_IMMEDIATE to receive full buffers of data until no more data
 *-----------------------------------------------------------------------
 */
int
receive_buffer_immed(rqp)
lu62_receive_t *rqp;
{
    int rc;
    bit8 *buffer;
    int buflen;
    int len = 0;
    int more_data = TRUE;

    rqp->fill = FILL_BUFFER;
    buflen = rqp->length;
    buffer = rqp->data;

    while (more_data) {

        more_data = FALSE;

        rqp->length = buflen;
        rqp->data = (bit8 *)buffer;

        printf("x%x: lu62_receive_immediate:, rqp->conv_id);
        rc = lu62_receive_immediate(rqp);
        printf(tpi_dis_receive_return(rqp, rc));

        if ((rqp->return_code == LU62_OK)
```

Code Example D-1 (32 of 32)

```
        &&  (rqp->what_received == WR_DATA)) {
            lu62_dump_buffer((FILE *)stdout,rqp->length,buffer,ASCII_DATA);
            more_data = TRUE;
        }
    }

    return(rc);
}
```

SunLink LU6.2 Configuration Examples

E

Code Example E-1 (1 of 8)

```
// SunLink LU6.2/sunPU2.1 SNA Server Sample Configuration
//
// This sample configuration allows standalone testing of SunLink LU6.2.
// Two LU6.2s (LUA and LUB) are defined for intra-node (LOCAL)
// communication.  TPs TPx, MPx, and XPx are defined to each LU to
// handle, basic, mapped and both conversation types, respectively.
// The COPR TP is defined to each LU with control operator privileges.
//
//      XPA                                 XPB
//      MPA                                 MPB
//      COPR   LUA <---- LOCAL ----> LUB   COPR
//      TPA                                 TPB
//

CP  Name=SUNCP,
    NQ_CP_NAME=IBMLAN.SUNCP;


// LU6.2 Logical Unit LUA

LU  NAME=LUA   // Local name (8 char max)
    NQ_LU_NAME=IBMLAN.LUA// Network Qualified Name
```

Code Example E-1 (2 of 8)

```
    SESS_LMT=12// Max LU sessions
    LUTYPE=6.2
    ;

PTNR_LU NAME=PLUB// Local name (8 char max)
    LOC_LU_NAME=LUA// Associated Local LU
    NQ_LU_NAME=IBMLAN.LUB// Network Qualified Name
    ;

MODE NAME=MODEAB// Mode Name (8 char max)
    DLC_NAME=LOCAL// Associated DLC
    PTNR_LU_NAME=PLUB// Associated Partner LU
    LCL_MAX_SESS_LMT=4// Max Session Limit
    MIN_CW_SESS=2// Min Conwinners
    MIN_CL_SESS=2// Min Conlosers
    ;

TP   TP_NAME=COPR// TP Name (8 char max)
        LOC_LU_NAME=LUA// Associated Local LU
        PRIVILEGE=CNOS// Privileged COPR verbs
        PRIVILEGE=SESSION_CONTROL
        PRIVILEGE=DISPLAY
    ;

TP   TP_NAME=XPA// TP Name (8 char max)
    LOC_LU_NAME=LUA// Associated Local LU
    CONV_TYPE=BASIC// Conversation Type
    CONV_TYPE=MAPPED// Conversation Type
    SYNC_LVL=NONE// Sync Level
    SYNC_LVL=CONFIRM// Sync Level
    TP_PATH="xterm -e '../SUNWappc/examples/cpic_rs -l LUA -p XPA'"
    ;

TP   TP_NAME=TPA
    LOC_LU_NAME=LUA
    CONV_TYPE=BASIC
    SYNC_LVL=NONE
    SYNC_LVL=CONFIRM
    TP_PATH="xterm -e '../SUNWappc/examples/tp_rs -l LUA -p TPA'"
    ;

TP  TP_NAME=MPA
```

Code Example E-1 (3 of 8)

```
    LOC_LU_NAME=LUA
    CONV_TYPE=MAPPED
    SYNC_LVL=CONFIRM
    TP_PATH=""
    ;


// LU6.2 Logical Unit LUB

LU  NAME=LUB   // Local name (8 char max)
    NQ_LU_NAME=IBMLAN.LUB// Network Qualified Name
    SESS_LMT=12// Max LU sessions
    LUTYPE=6.2
    ;

PTNR_LU NAME=PLUA// Local name (8 char max)
    LOC_LU_NAME=LUB// Associated Local LU
    NQ_LU_NAME=IBMLAN.LUA// Network Qualified Name
    ;

MODENAME=MODEAB// Mode Name (8 char max)
    DLC_NAME=LOCAL// Associated DLC
    PTNR_LU_NAME=PLUA// Associated Partner LU
    LCL_MAX_SESS_LMT=4// Max Session Limit
    MIN_CW_SESS=2// Min Conwinners
    MIN_CL_SESS=2// Min Conlosers
    ;

TP   TP_NAME=COPR// TP Name (8 char max)
        LOC_LU_NAME=LUB// Associated Local LU
        PRIVILEGE=CNOS// Privileged COPR verbs
        PRIVILEGE=SESSION_CONTROL
        PRIVILEGE=DISPLAY
    ;

TP   TP_NAME=XPB// TP Name (8 char max)
    LOC_LU_NAME=LUB// Associated Local LU
    CONV_TYPE=BASIC// Conversation Type
    CONV_TYPE=MAPPED// Conversation Type
    SYNC_LVL=NONE// Sync Level
    SYNC_LVL=CONFIRM// Sync Level
    TP_PATH="xterm -e '../SUNWappc/examples/cpic_rs'"
```

Code Example E-1 (4 of 8)

```
    ;

TP   TP_NAME=TPB
    LOC_LU_NAME=LUB
    CONV_TYPE=BASIC
    SYNC_LVL=NONE
    SYNC_LVL=CONFIRM
    TP_PATH="xterm -e '../SUNWappc/examples/tp_rs'"
    ;

TP   TP_NAME=MPB
    LOC_LU_NAME=LUB
    CONV_TYPE=MAPPED
    SYNC_LVL=CONFIRM
    TP_PATH=""
    ;




// Token Ring Peer-to-Peer System A
//
// This sample configuration defines LU6.2 LUA and a partner LUB.
// LUA and LUB support MODEAB sessions over data link DLC1.
// TPs TPA, MPA, and XPA are defined to handle, basic, mapped and
// both conversation types, respectively.  The COPR TP is defined
// to the LU with control operator privileges.
//
// The physical connection is realized via a Token Ring interface
// adapter.
//
// You will need to configure the following parameters:
// - TRLINE SOURCE_ADDRESS
// - DLC LCLLSAP
// - DLC RMTLSAP
// - DLC RMTMACADDR
//

CP  NAME=SUNCPA// Local name (8 char max)
    NQ_CP_NAME=IBMLAN.SUNCPA// Network Qualified Name
    ;
```

Code Example E-1 (5 of 8)

```
TRLINENAME=MAC1// Sun specific name
    SOURCE_ADDRESS=x'400000000001'// sysA_mac_addr
    ;

DLC NAME=DLC1  // User defined name (8 char max)
    LINK_NAME=MAC1// Line name this station is on
    LCLLSAP=x'04'// Local Link Service Access Point
    RMTLSAP=x'04'// Remove Link Service Access Point
    RMTMACADDR=x'400000000002'// sysB_mac_addr
    ;

LU  NAME=LUA   // Local name (8 char max)
    NQ_LU_NAME=IBMLAN.LUA// Network Qualified Name
    SESS_LMT=12// Max LU sessions
    LUTYPE=6.2
    ;

PTNR_LU NAME=PLUB// Local name (8 char max)
    LOC_LU_NAME=LUA// Associated Local LU
    NQ_LU_NAME=IBMLAN.LUB// Network Qualified Name
    ;

MODE NAME=MODEAB// Mode Name (8 char max)
    DLC_NAME=DLC1// Associated DLC
    PTNR_LU_NAME=PLUB// Associated Partner LU
    LCL_MAX_SESS_LMT=4// Max Session Limit
    MIN_CW_SESS=2// Min Conwinners
    MIN_CL_SESS=2// Min Conlosers
    ;

TP   TP_NAME=COPR// TP Name (8 char max)
        LOC_LU_NAME=LUA// Associated Local LU
        PRIVILEGE=CNOS// Privileged COPR verbs
        PRIVILEGE=SESSION_CONTROL
        PRIVILEGE=DISPLAY
    ;

TP   TP_NAME=XPA// TP Name (8 char max)
    LOC_LU_NAME=LUA// Associated Local LU
    CONV_TYPE=BASIC// Conversation Type
    CONV_TYPE=MAPPED// Conversation Type
    SYNC_LVL=NONE// Sync Level
```

Code Example E-1 (6 of 8)

```
    SYNC_LVL=CONFIRM// Sync Level
    TP_PATH="xterm -e '../SUNWappc/examples/cpic_rs -l LUA -p XPA'"
    ;

TP   TP_NAME=TPA
    LOC_LU_NAME=LUA
    CONV_TYPE=BASIC
    SYNC_LVL=NONE
    SYNC_LVL=CONFIRM
    TP_PATH="xterm -e '../SUNWappc/examples/tp_rs -l LUA -p TPA'"
    ;

TP  TP_NAME=MPA
    LOC_LU_NAME=LUA
    CONV_TYPE=MAPPED
    SYNC_LVL=CONFIRM
    TP_PATH=""
    ;


// Token Ring Peer-to-Peer System B
//
// This sample configuration defines LU6.2 LUB and a partner LUA.
// LUA and LUB support MODEAB sessions over data link DLC2.
// TPs TPB, MPB, and XPB are defined to handle, basic, mapped and
// both conversation types, respectively.  The COPR TP is defined
// to the LU with control operator privileges.
//
// The physical connection is realized via a Token Ring interface
// adapter.
//
// You will need to configure the following parameters:
// - TRLINE SOURCE_ADDRESS
// - DLC LCLLSAP
// - DLC RMTLSAP
// - DLC RMTMACADDR
//

CP  NAME=SUNCPB// Local name (8 char max)
    NQ_CP_NAME=IBMLAN.SUNCPB// Network Qualified Name
    ;
```

Code Example E-1 (7 of 8)

```
TRLINENAME=MAC2// Sun specific name
    SOURCE_ADDRESS=x'400000000002'// sysB_mac_addr
    ;

DLC NAME=DLC2  // User defined name (8 char max)
    LINK_NAME=MAC2// Line name this station is on
    LCLLSAP=x'04'// Local Link Service Access Point
    RMTLSAP=x'04'// Remove Link Service Access Point
    RMTMACADDR=x'400000000001'// sysA_mac_addr
    ;

LU  NAME=LUB   // Local name (8 char max)
    NQ_LU_NAME=IBMLAN.LUB// Network Qualified Name
    SESS_LMT=12// Max LU sessions
    LUTYPE=6.2
    ;

PTNR_LU NAME=PLUA// Local name (8 char max)
    LOC_LU_NAME=LUB// Associated Local LU
    NQ_LU_NAME=IBMLAN.LUA// Network Qualified Name
    ;

MODENAME=MODEAB// Mode Name (8 char max)
    DLC_NAME=DLC2// Associated DLC
    PTNR_LU_NAME=PLUA// Associated Partner LU
    LCL_MAX_SESS_LMT=4// Max Session Limit
    MIN_CW_SESS=2// Min Conwinners
    MIN_CL_SESS=2// Min Conlosers
    ;

TP   TP_NAME=COPR// TP Name (8 char max)
        LOC_LU_NAME=LUB// Associated Local LU
        PRIVILEGE=CNOS// Privileged COPR verbs
        PRIVILEGE=SESSION_CONTROL
        PRIVILEGE=DISPLAY
    ;

TP   TP_NAME=XPB// TP Name (8 char max)
    LOC_LU_NAME=LUB// Associated Local LU
    CONV_TYPE=BASIC// Conversation Type
    CONV_TYPE=MAPPED// Conversation Type
    SYNC_LVL=NONE// Sync Level
```

Code Example E-1 (8 of 8)

```
    SYNC_LVL=CONFIRM// Sync Level
    TP_PATH="xterm -e '../SUNWappc/examples/cpic_rs'"
    ;

TP   TP_NAME=TPB
    LOC_LU_NAME=LUB
    CONV_TYPE=BASIC
    SYNC_LVL=NONE
    SYNC_LVL=CONFIRM
    TP_PATH="xterm -e '../SUNWappc/examples/tp_rs'"
    ;

TP   TP_NAME=MPB
    LOC_LU_NAME=LUB
    CONV_TYPE=MAPPED
    SYNC_LVL=CONFIRM
    TP_PATH=""
```

LU6.2 Sync-Point F

SunLink P2P LU6.2 9.1 provides limited support for LU6.2 sync-point services. Transaction programs can establish sync-level sync-point conversations but the *IBM SNA Transaction Programmer's Reference Manual* sync-point verbs are not recognized. Instead, TPs exchange sync-point flows using the `lu62_send_ps_data` verb, which requires that the TP build the PS header and manage the sync-point state transitions. An external sync-point manager (TP 06f2) can send sync-point recovery GDS variables (Exchange lognames and Compare states) on basic conversations transparently to SunLink P2P LU6.2 9.1.

F.1 Sync-Point Flows

The structure and function of sync-point flow is described below.

F.1.1 Configuring for Sync-Point

Local LUs and TPs must be configured to support sync-point flows.

```
LU  NAME=LUA
    NQ_LU_NAME=IBMLAN.LUA
    SESS_LMT=128
    SYNC_LVL=SYNCPT
    LUTYPE=6.2;
```

The LU SYNC_LVL parameter is used when a session is activated (BIND). It represents the maximum sync-level supported by the LU. The default is CONFIRM.

```
TP  TP_Name=XPA
    LOC_LU_NAME=LUA
    CONV_TYPE=BASIC
    CONV_TYPE=MAPPED
    SYNC_LVL=NONE
    SYNC_LVL=CONFIRM
    SYNC_LVL=SYNCPT
    TP_PATH="tp -l LUA -p XPA'";
```

The TP SYNC_LVL parameter is used when checking incoming FMH5 attach requests. Attaches are rejected (FMH7 sense 0x10086041) if the requested sync-level is not configured. TP SYNC_LVL parameters are ORed together; in this example, TP XPA supports all sync-levels.

F.1.2 Logical Unit of Work ID

FMH5 attach requests for sync-level sync-point conversations must contain a logical unit of work identifier (LUW ID). The LUW ID can be supplied by the transaction program that initiates a SYNC_LEVEL_SYNCPT conversation as a parameter to lu62_(mc_)allocate. If the LUW ID is not supplied, it is assigned by LU6.2 presentation services on the caller's behalf.

The format of the LUW ID is described under the FMH5 attach request in Chapter 11 of the *IBM SNA Formats* manual. It consists of three parts: the fully qualified local LU name (2 - 18 bytes, including the length byte); an instance ID (6 bytes); and a sequence number (2 bytes). The system time (seconds since epoch) is copied into the high order word of the instance ID. The remaining two bytes of the instance ID are used to differentiate LUW IDs that are allocated during the same second. The two-byte sequence number is always set to 0.

The lu62_(mc_)get_attributes verb is issued by the allocating or accepting the program to retrieve the LUW ID. The LUW ID is also returned with lu62_listen LISTEN_ATTACH responses.

F.1.3 Sending PS Headers

The local TP sends PS header messages using the `lu62_send_ps_data` verb. This verb may be issued on both basic and mapped conversations. The supplied data contains the complete PS header, including the two-byte LL field containing an invalid length of 1, which identifies the record as a PS header. The `lu62_send_ps_data` parameter checks ensure that at least the LL field is sent and that the LL length is 1.

F.1.4 Forget Flows

The `lu62_send_ps_data_t` request structure contains a *forget* indicator. This indicator is set by the TP (the sync-point agent) when it issues the sync-point "committed" message. The SunLink SNA PU2.1 9.1 server sets a corresponding flag for the half-session to indicate that the next normal flow received on the session is an implied forget. Receipt of this implied forget is notified to the sync-point manager (TP 06f2) via the `lu62_listen` verb (it is assumed that an `lu62_listen` verb is always pending for TP 06f2). The `lu62_listen response_type` indicates `LISTEN_FORGET` and the `sess_id` field identifies the session.

The next normal flow can be a real forget message. The SunLink SNA PU2.1 9.1 server treats this like any other normal flow in that it sends an implied forget notification to the sync-point manager, and posts the record for receipt by the TP. The TP can discard the forget message, knowing that the sync-point manager was informed of the implied forget and has wiped its session log.

F.1.5 Receiving PS Headers

PS headers may be received by `lu62_receive_and_wait` or `lu62_receive_immediate` with a `fill_type` of `FILL_LL`, or on mapped conversations by `lu62_mc_receive_and_wait` or `lu62_mc_receive_immediate`. Receipt of a PS header is indicated by a `what_received` value of `WR_PS_DATA_COMPLETE`. The complete PS header is returned in the receive buffer, including the two-byte LL field containing an invalid length of 1. The LL field is *not* stripped on mapped receives. A `what_received` value of `WR_PS_DATA_INCOMPLETE` is indicated if the supplied receive buffer is too small. It is possible to receive `WR_LL_TRUNCATED` for PS headers if the sender splits the LL field; `lu62_send_ps_data` will not do this.

F.1.6 Deallocate Unbind

Under certain circumstances the TP needs to force a session unbind, preferably without using the COPR `lu62_deactivate_session` verb. An additional `deallocate_type`, `DA_UNBIND`, is provided by the `lu62_(mc_)deallocate` verb for this purpose. State checking for Deallocate(Unbind) is the same as for Deallocate(Abend).

F.2 Sync-Point Recovery

The sync-point manager, TP 06f2, sends and receives sync-point GDS variables (exchange logname and Compare states) on basic conversations to perform sync-point recovery. Building and parsing GDS variables is the responsibility of the sync-point manager.

SunLink LU6.2 9.0 to 9.1 Instructions

G

SunLink P2P LU6.2 9.1 provides a migration path for you to use your SunLink 9.0 P2P LU6.2 applications. This chapter provides instructions on how to link SunLink LU6.2 9.0 to SunLink LU6.2 9.1 applications. Also discussed are the differences between SunLink LU6.2 9.0 and SunLink LU6.2 9.1.

G.1 Linking the Application

To run the 9.0 version applications with the 9.1 version server, you need to relink your object files with the following libraries:

- `sunlu62.a`
- `sunp2p.a`

G.2 Differences Between Version 9.0 and 9.1

The following lists the differences between the SunLink 9.0 and SunLink 9.1 products.

Note – Version 9.0 functions and parameters appear in **boldface**. The control operator verbs are *not* supported. New verbs with the format `lu62_8` are provided in their place and are described in this manual.

allocate: SUB_DELAYED_ALLOCATION_PERMITTED is no longer supported.

appc_error, get_appc_error_msg (and *_mc_*): No longer supported. Use tracing as specified elsewhere in this manual

rd_iso_ebcdic_table, wr_iso_ebcdic_table: No longer supported. Use gateway command line commands to specify ISO-EBCDIC translation.

get_attributes: The following attributes are no longer supported:
- tp_name - returned as '\0'.
- pu_name - returned as '\0'.
- lu_local_address - returned as '\0'.

cnos: No longer supported. Must use one of the following:
- lu62_change_session_limit
- lu62_init_session_limit
- lu62_reset_session_limit

display_local_lu: No longer supported. Must use lu62_display_local_lu

display_mode: No longer supports send_pac_window, rcv_pac_window

These values are set to 7 in display_mode return value. You can also use lu62_display_mode.

display_partner_lu: No longer supports remote_is_sccp

The value is set to inverse of parallel_session_supported. You can also use lu62_display_remote_lu.

display_tp: No longer supported. You must use lus62_display_tp

deallocate: No longer supports SUB_NODE

send_error: No longer supports dont_flush_fmh7_on_ec, sense_code_in_fmh

tp_accept: No longer supports security_acceptance. It is always set to SECURITY_NONE.

tp_listen: No longer supports queue_depth, sync_level, conv_type, security

These are now specified in the configuration file. They are ignored by the API.

`tp_start`: Gateway name should be set to the host name of the workstation running the gateway.

`tp_wait_remote_start`: No longer supports `security_acceptance.` It is always set to `SECURITY_NONE`.

G.2.1 F10 Key Conflict

In 3270x, under OpenWindows, F10 key switches to the FILE MENU instead of sending the IBM PF10 key.

▼ To solve this conflict:

1. **Copy the appropriate file from `/usr/dt/lib/bindings` (for sun or sun_at if you are on an x86) to `~/.motifbind`.**
2. **Change the two entries that use the F10 key to some other key (F9 for example).**
3. **Run the following command:**

```
% xmbind ~/.motifbind
```

   If you have any motif clients already running, you will need to restart them in order for them to get the new key bindings.

4. **If you want to check the bindings on your desktop, run the following:**

```
% xprop -root | grep BIND
```

Index