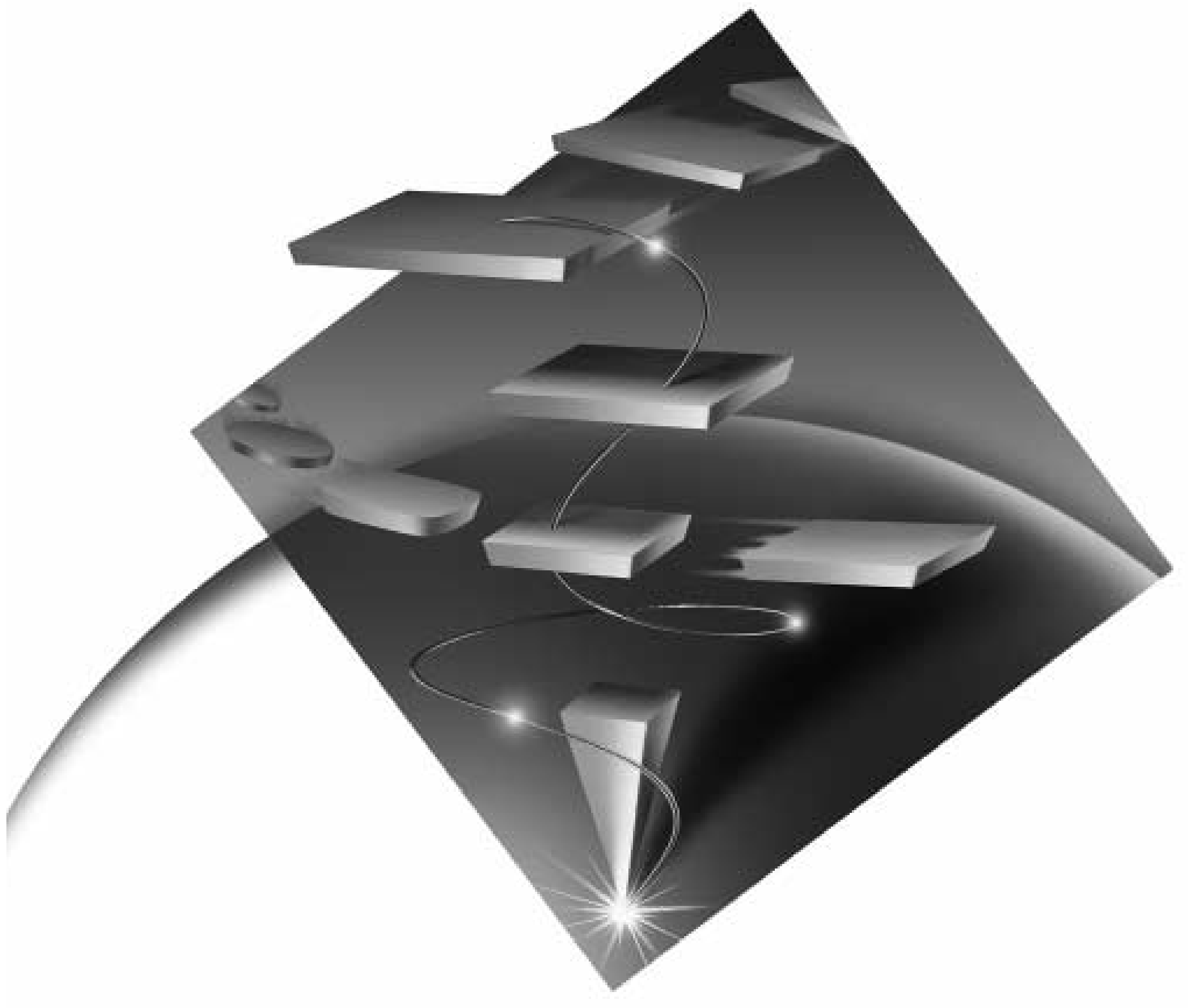
eNetwork Communications Server
Version 5.01
for Windows** NT

IBM

Client/Server Communications Programming



SC31-8425-01

eNetwork Communications Server
Version 5.01
for Windows** NT

Client/Server Communications Programming

SC31-8425-01

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page xiii.

Second Edition (September 1997)

This level applies to Version 5.01 Communications Server and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for readers’ comments may be provided at the back of this publication, or you may address your comments to the following address:

International Business Machines Corporation
Department CGM
P.O. Box 12195
Research Triangle Park, North Carolina
27709-2195

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1994, 1997. All rights reserved.
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Figures

Tables

© Copyright IBM Corp. 1994, 1997

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program or service is not intended to state or imply that only IBM's product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, New York 10594
United States of America

For license inquires regarding double-byte character set (DBCS) information, contact:

IBM World Trade
Asia Corporation Commercial Relations
IBM Kamiya-cho Building,
4-3-9 Toranomon, Minato-ku
Tokyo, Japan

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Site Counsel
IBM Corporation
P.O. Box 12195
3039 Cornwallis Road
Research Triangle Park, NC
27709-2195
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement.

This document is not intended for production use and is furnished as is without any warranty of any kind, and all warranties are hereby disclaimed including the warranties of merchantability and fitness for a particular purpose.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

ACF/VTAM
Advanced Peer-to-Peer Networking
AFP
AIX
AIXwindows
Application System/400
APPN
AS/400
CallPath
CallPath/2
CallPath SwitchServer/2
CICS
CUA
C/2
IBM
IMS
NetView
Operating System/2
OS/2
RACF
SAA
SP
System/370
S/370
VM/ESA
VTAM

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

C-bus is a trademark of Corollary, Inc.

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

Java and HotJava are trademarks of Sun Microsystems, Inc.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

About This Book

This book is divided into four parts.

- Part 1. APPC API, describes how to develop programs that use the Communications Server advanced program-to-program communications (APPC) interface. APPC refers to an implementation of Systems Network Architecture (SNA) for logical unit (LU) type 6.2. Throughout this book, unless otherwise noted, APPC represents the Communications Server implementation of APPC.

  APPC provides a distributed transaction processing capability in which two or more programs cooperate to carry out some processing function. This capability involves communication between the programs so they can share resources, such as processor cycles, databases, work queues, and physical interfaces such as keyboards and displays.
- Part 2. LUA API, describes how to develop programs that use the IBM conventional logical unit application (LUA) interface (in this book LUA also refers to request unit interface (RUI)), that gives access to SNA LU types 0, 1, 2, and 3.
- Part 3. Common Services API, includes the verbs that make up the Common Services API.
- Part 4. EHNAPPC API, includes router windows API, functions, structures, and return codes.

This book is for users of client and server applications provided by Communications Server. Client APIs are provided for Windows NT, Windows 95, Windows 3.1, and OS/2 platforms. In this book, *Windows* refers to Windows NT.

Who Should Read This Book

This book is intended for programmers and developers who are writing either APPC or LUA applications.

This book assumes the reader is familiar with *SNA Transaction Programmer's Reference Manual for LU Type 6.2*.

How to Use This Book

- Chapter 1. Introducing APPC, describes advanced program-to-program communications (APPC).
- Chapter 2. Fundamental APPC Concepts, describes APPC transaction programs.
- Chapter 3. Using the Attach Manager, describes how to use the attach manager.
- Chapter 4. Writing a Transaction Program, describes how to write a transaction program.
- Chapter 5. Implementing APPC Transaction Programs, describes the APPC extensions.
- Chapter 6. Implementing CPI-C Programs, describes CPI-C programs.
- Chapter 7. APPC Entry Points, describes the procedure entry points for the APPC API.
- Chapter 8. APPC Verbs, describes the syntax of each APPC verb. A copy of the structure that holds the information for each verb is included and each entry is described, followed by a list of possible return codes.
- Chapter 9. Fundamental Concepts of the IBM Conventional LU Application, describes the fundamental LUA programming concepts in this book.

© Copyright IBM Corp. 1994, 1997

- Chapter 10. Features of the LUA Verbs, describes the features of LUA verbs.
- Chapter 11. Implementing LUA Programs, describes some of the aspects of writing LUA application programs.
- Chapter 12. RUI Entry Points, describes procedure entry points for LUA.
- Chapter 13. RUI Verbs, describes details for each LUA verb.
- Chapter 14. SLI Entry Points, describes the procedure entry points for SLI.
- Chapter 15. SLI Verbs, describes the SLI verbs.
- Chapter 16. Entry Points, describes procedure entry points.
- Chapter 17. Common Services Verbs (CSV), describes common services verbs.
- Chapter 18. EHNAPPC Application Program Interface, describes the EHNAPPC API.
- Chapter 19. Data Transform Windows Application Program Interface, describes data transform windows APIs.
- Appendix A. APPC Common Return Codes, contains descriptions of the APPC common return codes.
- Appendix B. LUA Verb Return Codes, contains descriptions of the LUA common return codes.
- Appendix C. APPC Conversation State Transitions, describes the conversation states in which each APPC verb can be issued, and the state change that occurs on completion of the verb.

Icons

This book uses *icons* (pictures) in the text to help you find different types of information.

This icon represents information that applies to basic APPC verbs. See Chapter 8. APPC Verbs for more information on basic verbs.

This icon represents information that applies to mapped APPC verbs. See Chapter 8. APPC Verbs for more information on mapped verbs.

Also, in this book when it is necessary to communicate special information, the ringing phone icon appears.

Conventions Used in This Book

The following conventions are used throughout the Communications Server library. Some of the conventions listed might not be used in this particular book.

Text Conventions

Bold	Bold type indicates verbs, functions, and parameters that you can use in a program or at a command prompt. These values are case sensitive and should be entered exactly as they appear in the text.
Italics	Italic type indicates the following things: • A variable that you supply a value for. • The names of window controls, such as lists, check boxed, entry fields, push buttons, and menu choices. They appear in the text as they appear in the window. • Book titles. • A letter is being used as a letter or a word is being used as a word. **Example:** When you see an *a*, make sure it is not supposed to be an *an*.
Bold italics	Bold italic type is used to emphasize a word.
UPPERCASE	Uppercase indicates constants, file names, keywords, and options that you can use in a program or at a command prompt. You can enter these values in uppercase or lowercase.
Double quotation marks	Double quotation marks indicate messages you see in a window. An example of this would be the messages that appear in the operator information area (OIA) of an emulator session.
`Example type`	Example type indicates information that you are instructed to type at a command prompt or in a window.

Number Conventions

Binary numbers	Represented as BX'xxxx xxxx' or BX'x' except in certain instances where they are represented with text (“A value of binary `xxxx xxxx` is...”).
Bit positions	Start with 0 at the rightmost position (least significant bit).
Decimal numbers	Decimal numbers over 4 digits are represented in metric style. A space is used rather than a comma to separate groups of 3 digits. For example, the number sixteen thousand, one hundred forty-seven is written 16 147.
Hexadecimal numbers	Represented in text as hex `xxxx` or X'xxxx' (“The address of the adjacent node is hex 5D, which is specified as X'5d'”)

Double-Byte Character Set Support

Communications Server supports double-byte character sets (DBCS), in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, the typing, displaying, and printing of DBCS characters require hardware and programs that support DBCS.

Where information applies specifically to DBCS, it is noted in this information unit.

ASCII refers to PC single-byte code in this book. ASCII should be considered as JISCII in Japan.

Where to Find More Information

Related Publications

For information about SNA, APPN, or LU 6.2 architecture, refer to the following IBM documents:

- *IBM Systems Network Architecture: LU 6.2 Reference: Peer Protocols* , SC31-6808
- *IBM Systems Network Architecture: APPN Architecture Reference*, SC30-3422
- *IBM Systems Network Architecture: Management Services* SC30-3346
- *IBM Systems Network Architecture: Formats* , GA27-3136
- *IBM APPN Architecture and Product Implementations Tutorial* GG24-3669
- *IBM AS/400 Advanced Peer-to-Peer Networking* , GG24-3287
- *IBM Communications Manager/2: System Management Programming Reference* , SC31-6173
- *IBM Communications Manager/2: APPC Programming Guide and Reference* , SC31-6160
- *IBM System/370 Principles of Operation* , GA22-7000
- *IBM Systems Network Architecture: Technical Overview* GC30-3073
- *IBM Systems Network Architecture: VTAM Programming for LU Type 6.2* , SC30-3400
- *IBM Systems Network Architecture: Concepts and Products* GC30-3072
- *IBM Systems Network Architecture Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2* , SC30-3269
- *IBM Systems Network Architecture: Introduction to APPC* GG24-1584
- *IBM Systems Network Architecture: Transaction Programmer’s Reference Manual for LU Type 6.2* , GC30-3084
- *IBM Systems Network Architecture: Introduction to Sessions between Logical Units* , GC20-1869
- *IBM Systems Network Architecture Format and Protocol Reference Manual: Architectural Logic* , SC30-3112

Part 1. APPC API

Chapter 1. Introducing APPC

Communications Server provides Advanced Peer-to-Peer Networking (APPN) end-node support for workstations, enabling them to communicate more flexibly with other systems in the network.

Communications Server provides advanced program-to-program communications (APPC) to support communications between distributed processing programs, called *transaction programs* (TPs). APPN extends this capability to a networking environment. The transaction programs can be located at any node in the network that provides APPC.

Communications Server improves APPC throughput in local area network (LAN) environments and supports APPC over various protocols such as: IBM Token-Ring Network, Synchronous Data Link Control (SDLC), Twinaxial, and Ethernet.

Figure 1 illustrates the functional structure of the Communications Server implementation of APPC.

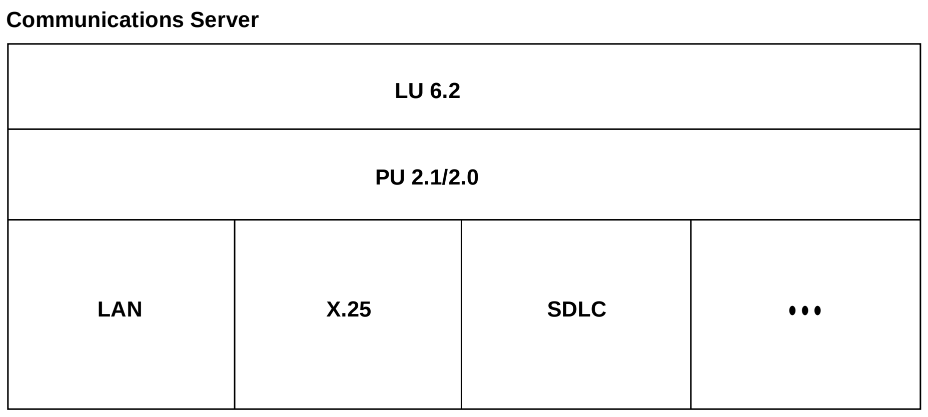


Figure 1. Communications Server APPC Implementation

SNA Communications Support

Communications Server supports Systems Network Architecture (SNA) type 2.1 nodes (including SNA type 2.0 and SNA type 2.1 support for logical units [LUs] other than SNA LU 6.2). This support lets you write programs to communicate with many other IBM SNA products.

You can write programs without knowing the details of the underlying network. All you need to know is the name of the partner LU; you do not need to know its location. SNA determines the partner LU location and the best path for routing data. A change to the underlying network, the addition of a new adapter, or the relocation of a machine, does not affect APPC programs. A program might, however, need to establish link connections over switched SDLC connections.

When Communications Server starts, it establishes local LU and logical link definitions, which are stored in a configuration file. The system management application programming interface (API) provides functions that control configuration definition and adapter and link activation. Refer to *Communications Server: System*

Management Programming for information about these functions. Users can use the configuration and node operations functions while Communications Server runs. Refer to *Communications Server: Quick Beginnings* and *Communications Server: System Management Programming* for more information about configuration and node operations.

SNA LU Type 6.2 Support

LU 6.2 is an architecture for program-to-program communications. Communications Server supports all base LU 6.2 functions. Some of the optional SNA LU 6.2 functions are:

- Basic and mapped conversations
- Half-duplex or full-duplex conversation styles
- Synchronization level of confirm
- Security support at session and conversation levels
- Multiple LUs
- Parallel sessions, including the ability to use a remote system to change the number of sessions
- Concurrent multiple links, each of which can be connected to a subarea network (one per PU definition)

Chapter 2. Fundamental APPC Concepts

This book describes the APPC API supported by Communications Server. Its purpose is to provide:

- A brief overview of the structure of the APPC API
- Reference information about the specific syntax of the verbs that flow across the interface

What Is a Transaction Program?

A transaction program is a block of code, or part of an application program, that uses APPC communications functions. Application programs use these functions to communicate with application programs on other systems that support APPC. A transaction program has a 64-byte name (**tp_name**).

Your transaction program can obtain LU 6.2 services through either of the following APIs:

- APPC—Advanced Program-to-Program Communication allows transaction programs to exchange information across an IBM SNA network using the syntax and verbs defined by IBM for using an LU 6.2 session.
- CPI-C—Common Programming Interface for Communications (CPI-C) allows transaction programs to exchange information across an IBM SNA network using the syntax defined in the Common Programming Interface component of the SAA by IBM for using an LU 6.2 session. Because this API is implemented for many platforms, CPI-C applications can be easily ported.

Transaction programs issue APPC verbs to invoke APPC functions. See “Chapter 5. Implementing APPC Transaction Programs” on page 37, for details about how transaction programs issue APPC verbs. Transaction programs can issue CPI Communications calls to invoke CPI Communications functions. The CPI Communications calls let application programs take advantage of the consistency that SAA provides. See “CPI Communications Transaction Programs” on page 8 for information about the CPI Communications calls.

APPC Transaction Programs

An APPC transaction program is not an application; it is a section of an application. A single application can contain many transaction programs. Every transaction program has a unique 8-byte identification number (**tp_id**).

APPC supplies verbs that start and stop transaction programs within applications. APPC also supplies a full set of conversation verbs that you can use to implement the function of your transaction program.

A transaction program issues a request to APPC, in the form of a verb, to perform some action for an application program. A verb is a formatted request that a transaction program issues and APPC executes. A program uses APPC verb sequences to communicate with another program. Two programs that communicate with each other can be located at different systems or on the same system. The APPC API is the same in both cases.

You can consider the set of APPC verbs described in this book as a programming language in which you write conversations. APPC verbs are coded as control

© Copyright IBM Corp. 1994, 1997

blocks; each verb’s syntax is precisely defined. Your programs pass verb control blocks to the APPC API to gain access to APPC facilities.

When a transaction program exchanges data with another transaction program, they are called *partner* transaction programs.

CPI Communications Transaction Programs

A CPI Communications transaction program is similar to an APPC transaction program; both types of transaction programs use APPC support. Rather than issuing verbs, however, a CPI Communications transaction program invokes each CPI Communications function with a call to the function that passes the appropriate parameters on the call.

Most CPI Communications calls correspond to APPC verbs. For example, the calls that allocate outbound conversations and accept (receive) conversations, and the calls that send and receive data on the conversation, provide functions that are similar to those of the corresponding APPC verbs. The exceptions are the calls that initialize a conversation before allocating the conversation and the calls that set and extract individual conversation characteristics.

Refer to *CPI Communications Reference* for details about the support that Communications Server provides for CPI Communications programs.

Client Transaction Programs

Typically, a program begins a conversation because it requires a service from another program. This program is called a client transaction program. The client transaction program requests the conversation through the LU 6.2 API.

Often the client transaction program is started by a human user; however, the client transaction program could actually be a server transaction program responding to another program’s request. In any conversation, the client transaction program is always running before the conversation begins. The client transaction program startup and termination are not directly related to the conversation. The client transaction program initiates the conversation, and it can continue to run after the conversation is over.

Server Transaction Programs

The server transaction program delivers the service that is requested by the client transaction program.

The server transaction program can run continuously, waiting for clients to begin conversations with it. But frequently, the server transaction program handles a single transaction, and is started by the APPC API to handle one specific conversation. The server transaction program begins execution when a conversation is requested, and it terminates when the conversation is finished.

An important feature of the LU 6.2 architecture is that it can start server transaction programs when client transaction programs request them. You can design your server programs to this model and arrange for them to be started on demand.

What Is a Logical Unit?

Every transaction program gains access to an SNA network through a *logical unit* (LU). An LU is SNA software that accepts verbs from your programs and acts on those verbs. A transaction program issues APPC verbs to its LU. These verbs cause commands and data to flow across the network to a partner LU. An LU also acts as an intermediary between the transaction programs and the network to manage the exchange of data between transaction programs. A single LU can provide services for multiple transaction programs. Multiple LUs can be active simultaneously.

LU Types

Communications Server supports LU types 0, 1, 2, 3, and 6.2. LU types 0, 1, 2, and 3 support communication between host application programs and different kinds of devices, such as terminals and printers. Refer to Part 2. LUA API, for details on writing these programs.

LU 6.2 supports communications between two programs located at type 5 subarea nodes, type 2.1 peripheral nodes, or both, and between programs and devices. APPC is an implementation of the LU 6.2 architecture, which is described in this part of the book.

Communication occurs only between LUs of the same LU type. For example, an LU 2 communicates with another LU 2; it does not communicate with an LU 3.

Dependent and Independent LUs

A *dependent LU* depends on a system services control point (SSCP) to activate a session. A dependent LU needs an active SSCP-LU session, which the dependent LU uses to start an LU-LU session with an LU in a subarea node. A dependent LU can have only one session at a time with the subarea LU. For communications with a transaction program at a subarea node, each dependent LU can have only one conversation at a time, and each dependent LU can support communications for only one transaction program at a time.

An *independent LU* does not depend on an SSCP to activate a session. An independent LU supports multiple concurrent sessions with other LUs in a subarea node, so you can have multiple conversations and support multiple transaction programs for communications with subarea transaction programs. LUs between peripheral nodes also use this support.

The distinction between a dependent LU and an independent LU is meaningful only when discussing a session between an LU in a peripheral node and an LU in a subarea node. Otherwise, dependent and independent LUs both support multiple concurrent sessions and conversations when communicating between type 2.1 peripheral nodes (for example, between two workstations). A Communications Server LU can support a single session with a dependent LU or multiple sessions with an independent LU.

What Is an LU Name?

An LU is a point of access to the Systems Network Architecture (SNA) network. An LU has a name and other characteristics that are *configured* (formally recorded) throughout the SNA network. Sometimes this configuration is static, done by the network administrator and recorded in configuration files. Sometimes the configuration is dynamic, prepared by programs from file or user input.

To open a conversation, a client transaction program must specify both the name of the server transaction program and the name of the LU where the server transaction program can be reached. Sometimes these names are embedded in the client transaction program. In other cases, the names are stored externally to the client transaction program or are specified dynamically.

What Is a Session?

Before transaction programs can communicate with each other, their LUs must be connected in a mutual relationship called a *session*. A session connects two LUs, so it is called an *LU-LU* session. Figure 2 illustrates this communication relationship. Multiple concurrent sessions between the same two LUs are called *parallel* LU-LU sessions.

Sessions act as conduits that manage the movement of data between a pair of LUs in an SNA network. Specifically, sessions deal with things such as the quantity of data transmitted, data security, network routing, and traffic congestion.

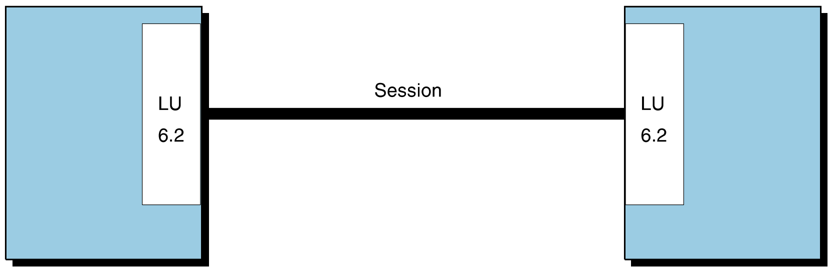


Figure 2. A Session between Two LUs

Sessions are maintained by their LUs. Normally, your transaction programs do not deal with session characteristics. You define session characteristics when you:

- Configure your system
- Use the management verbs

What Is a Conversation?

The communication between transaction programs is called a *conversation*. Conversations occur across LU-LU sessions. A conversation starts when a transaction program issues an APPC verb or CPI Communications call that allocates a conversation. The conversation style associated with the conversation indicates the style of data transfer to be used, two-way alternate or two-way simultaneous. A conversation that specifies a two-way alternate style of data transfer is also known as a *half-duplex* conversation. A conversation that specifies a two-way simultaneous style of data transfer is referred to as a *full-duplex* conversation.

Figure 3 shows a conversation after it has been set up.

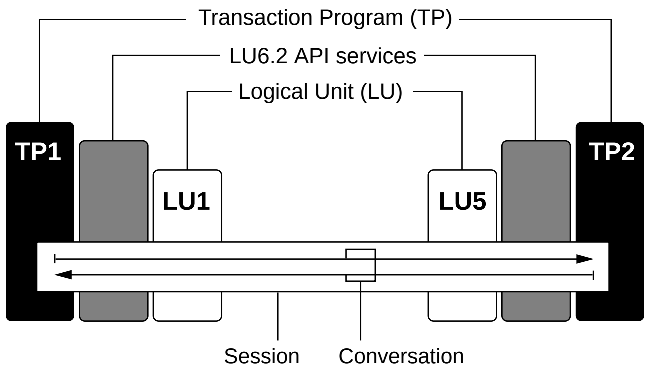


Figure 3. Parts of a Conversation

When a full-duplex conversation is allocated to a session, a send-receive relationship is established between the transaction programs connected to the conversation, and a two-way alternate data transfer occurs where information is transferred in both directions, one direction at a time. Like a telephone conversation, one transaction program calls the other, and they “converse”, one transaction program talking at a time, until a transaction program ends the conversation. One transaction program issues verbs to send data, and the other transaction program issues verbs to receive data. When it finishes sending data, the sending transaction program can transfer send control of the conversation to the receiving transaction program. One transaction program decides when to end the conversation and informs the other when it has ended.

When a duplex conversation is allocated to a session, both transaction programs connected to the conversation are started in send-and-receive state, and a two-way simultaneous data transfer occurs where information is transferred in both directions at the same time. Both transaction programs can issue verbs to send and receive data simultaneously with no transfer of send control required. The conversation ends when both transaction programs indicate they are ready to stop sending data, and each transaction program has received the data sent by the partner. If an error condition occurs, one transaction program can decide to end both sides of the conversation abruptly.

Conversations can exchange control information and data. The transaction program should select the conversation style best-suited for its application.

Figure 4 on page 12 shows a conversation between two transaction programs as it occurs over a session.

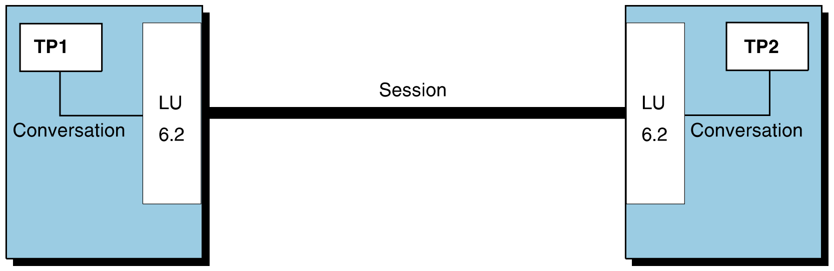


Figure 4. A Conversation between Two Transaction Programs

A session can support only one conversation at a time, but one session can support many conversations in sequence. Because multiple conversations can reuse sessions, a session is a long-lived connection compared to a conversation. When a program allocates a conversation and all applicable sessions are in use, the LU puts the incoming Attach (allocation request) on a queue. It completes the allocation when a session becomes available. See Chapter 3. Using the Attach Manager,

Two LUs can also establish parallel sessions with each other to support multiple concurrent conversations. A parallel session occurs when either transaction program allocates a conversation, and a session exists but is being used by a conversation. The LU can request a new session to satisfy the allocation.

Figure 5 shows three parallel sessions between two LUs; each session carries a conversation.

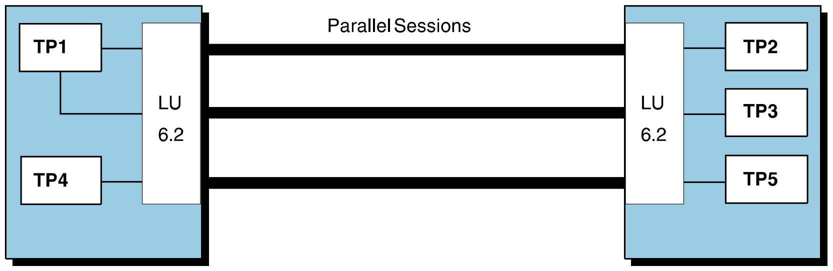


Figure 5. Parallel Sessions between LUs

An APPC conversation is a *half-duplex* conversation. At any instant, only one of the two partner transaction programs has the right to send data. That transaction program is in *send state*. The other transaction program has the responsibility to receive data. It is said to be in *receive state*. At specified times, the transaction programs exchange these duties. When the conversation is first set up, the client transaction is in send state and the server program is in receive state.

Relationships among Sessions, Conversations, and LUs

A connection between LUs is called a *session*. This connection can pass through intermediate network nodes. However, LU 6.2 programs do not need to account for the details of the connection. It makes no difference to the client transaction program whether the server transaction program is in the same room or thousands of miles away. The LU 6.2 API is responsible for starting and ending sessions between LUs of type 6.2.

Though a session can carry only one conversation at a time, it can be reused for another conversation when the first one is finished. The LU 6.2 software determines whether to terminate a session when the conversation ends, or to keep the session open and reuse it.

Some LUs can handle multiple, parallel sessions. Each session is independent. Some possible relationships among machines, LUs, sessions, and transaction programs are illustrated in Figure 6.

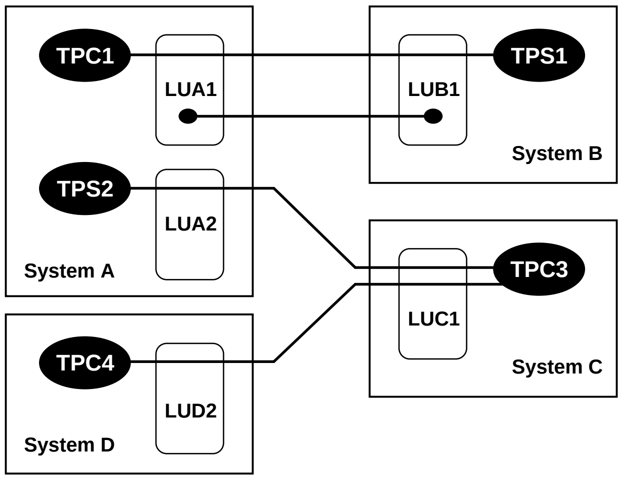


Figure 6. Relationships between Programs and LUs

Figure 6 depicts two parallel sessions between LUA1 in System A and LUB1 in System B. One session carries a conversation between client TPC1 and server TPS1. The other session is not in use for a conversation at this time.

In System C, LUC1 also supports two parallel sessions. Both are in use by client TPC3, which is carrying on a conversation with server TPS2 in System A. TPC3 also has a conversation in progress with TPC4 in System D. This figure illustrates that a transaction program is not limited to a single conversation. The figure also shows that a program can be both a client and a server. A possible scenario for the conversations could be that program TPC4 started program TPC3 in order to request a service. To deliver that service, TPC3 requested a service from TPS2.

Examples of APPC Operations

Table 1 describes possible LU 6.2 operations in abstract terms.

Table 1. LU 6.2 Operations

Operation	**What the Operation Does**
Send	Sends a block of data to the other program.
Receive	If currently in send state, transmits any buffered output data and enters receive state. Waits for data to arrive and receives it.
Await confirmation	Transmits any buffered output data. Waits until the partner program confirms that it has received and processed all data.
Confirm	Sends the partner program confirmation that all data has been received and processed.

Table 1. LU 6.2 Operations (continued)

Operation	What the Operation Does
Error	If in receive state, purges any buffered input data and enters send state. If currently in send state, purges any buffered output data. Causes the partner program's current operation to end with a special return code.
Close	If currently in send state, transmits any buffered output data. Ends the conversation.

Both LU 6.2 APIs offer these services (and others), and both offer services that allow you to combine two or more of these basic operations to improve performance. The following sections use these terms when discussing the types of conversations to avoid contrasting the details of each API. For example, the term *Send* used in Table 1 on page 13 can represent the APPC verbs SEND_DATA, or MC_SEND_DATA, or the CPI-C function CMSEND.

Types of APPC Conversations

This section discusses the types of APPC conversations.

- One-way
- Confirmed-delivery
- Inquiry
- Database update

One-Way Conversation

In the one-way conversation, the simplest type of conversation, the client transaction program passes some data to the server and the server notes it, as summarized in Table 2.

Table 2. Actions in One-Way Conversation

Client Actions	Server Actions
Send one or more records.	
Close.	*Receive* and process the records.
	Close.

This minimal sort of conversation is used with data whose delivery is not critical; for example, to periodically update a status display, to record usage levels, or log a condition.

Confirmed-Delivery Conversation

In the next simplest type of conversation, the confirmed-delivery conversation, the client transaction program sends a record and the server confirms its receipt, as summarized in Table 3.

Table 3. Actions in Confirmed-Delivery Conversation

Client Actions	Server Actions
Send one or more records.	
Await confirmation.	
	Receive and process the records.
	Confirm the records.

Table 3. Actions in Confirmed-Delivery Conversation (continued)

Client Actions	Server Actions
Close.	*Close*.

This type of conversation can be used in a credit-authorization system (the client sends an account number and purchase amount, and the server's confirmation authorizes the sale) among its other uses. For example, the client transaction program could send a database record of any kind, and the server could confirm that the database had been updated. Because there is no upper limit on how much data the client can send, this type of conversation could be used to send an entire file of data in batch mode. In this type of conversation the client transaction program receives only the confirmation; it needs no other data returned to it.

The difference between a *Confirm* operation and a *Send* is that *Confirm* transmits only the shortest possible SNA message, the positive response that all data has been received and processed.

Inquiry Conversation

In an inquiry conversation, the client sends one request for information and the server generates one response, as summarized in Table 4. (Both the inquiry and the response can comprise any number of logical records.) This type of conversation appears in many kinds of data processing applications.

Table 4. Actions in Inquiry Conversation

Client Actions	Server Actions
Send one or more records.	
Receive.	
	Receive and process the records.
	Send a response consisting of one or more records.
Continue to *Receive* until all response data has arrived.	*Close*.
Close.	

When you design transactions to this model, the server transaction programs are very simple. Each handles one instance of one type of query and then terminates. The client transaction program requests a conversation with the server transaction program that can answer the desired type of query. The LU 6.2 API services locate and start a copy of that server transaction program.

Database Update Conversation

In the database update conversation, the client transaction program requests a copy of data, modifies it, and returns it to be stored. The server transaction program locks the data for the client's use until the update is complete. Table 5 summarizes client and server actions.

Table 5. Actions in Database Update Conversation

Client Actions	Server Actions
Send a request for data (a record key).	
Receive.	
	Receive the key value.
	Fetch the record and lock it.

Table 5. Actions in Database Update Conversation (continued)

Client Actions	Server Actions
	Send a copy of the record.
	Receive.
Process the received record.	
Send the updated record.	
Await confirmation.	
	Update the database with the received record.
	Confirm the update.
Close.	*Close*.

Refer to Table 1 on page 13 to clarify this process. When the client transaction program first issues *Receive*, three things occur:

- LU 6.2 send buffer is flushed of any remaining logical records sent by the client.
- The client transaction program, that began in send state, switches to receive state. The right to send passes to the server transaction program.
- The client transaction program waits until data arrives. (Nonblocking receive operations are available also.)

Similarly, the second *Receive* issued by the server flushes its buffer and transfers the right to send back to the client transaction program.

Conversations That Have Errors

Conversation errors are inevitable, and your transaction program must be prepared to detect and respond to them. A transaction program uses the *Error* operation, described in Table 1 on page 13, to signal the discovery of an error. Table 6 summarizes an inquiry conversation in which the server finds a logical error in the inquiry.

Table 6. Inquiry Conversation with Error

Client Actions	Server Actions
Send one or more records.	
Receive.	
	Receive and process some of the inquiry records. Find a mistake.
	Error.
	Send diagnostic error message.
Return code to *Receive* indicates *Error* by partner.	*Close*.
Receive diagnostic message, display to user	
Close	

The main purpose of the *Error* operation is to purge all unsent and unreceived data that might be in API buffers at either transaction program. The *Error* operation also gives the right to send to the transaction program which discovered the error, so the transaction program can transmit diagnostic data to its partner. Your transaction program must specify the contents of the diagnostic message and the operations that follow.

Conversation Types

Communications Server LU 6.2 supports two types of conversations, mapped and basic, and therefore provides a separate set of verbs for each. The conversation type you use depends on whether you need full access to the SNA *general data stream (GDS)* as provided by basic conversations. The GDS defines what is known as a GDS variable. A GDS variable consists of one or more logical records. Each logical record begins with a *logical length (LL)* field that specifies the overall length of the logical record (data). The first logical record of a GDS variable also includes, immediately after the logical length field, an *identification (ID)* field that specifies the type of GDS variable.

Mapped Conversations

Use *mapped conversations* for transaction programs that are the final users of the data exchanged. A mapped conversation enables advanced program-to-program communication in an easy-to-use record-level manner. Because a transaction program using mapped conversations does not require GDS headers to describe the data, the program does not have to build or interpret these headers. When the transaction program uses mapped conversations, Communications Server LU 6.2 builds and interprets GDS variables.

In a mapped conversation, the programs exchange records in whatever format you design.

- Each send operation takes a record of a specified length from 0 bytes to 65,535 bytes. Communications Server formats the record into a single GDS variable.
- A receive operation returns all or part of one sent record (GDS variable without header fields), depending on how much buffer space the program allocates. The return code indicates when the final part of a record sent by the partner program has been received.

The APPC API takes full responsibility for the following tasks:

- Blocking and buffering multiple records
- Formatting data as SNA GDS variables
- Buffering at the receiving program
- Deblocking and delivery to the Receive operation

Basic Conversations

In a basic conversation, transaction programs exchange logical records from 0 to 32,765 bytes in length.

- Each send operation takes a buffer containing from 0 to 65,535 bytes of logical records. The buffer can contain one or more logical records and parts of records. Logical records can be broken across send calls.
- A receive operation can be used to accept either a single logical record or a buffer filled with one or more logical records and parts of records.

Summary

Two transaction programs use LU 6.2 to exchange data in a conversation. One, the client transaction program, is typically started by a user. The other, the server transaction program, can be started automatically to render a service to the client. A transaction program uses one of two APIs: APPC, or CPI-C, which have different styles and similar, but not identical, sets of services.

The conversation takes place over a session between two LUs. An LU represents a point at which a transaction program can access the SNA network. A session represents the connection between two LUs, without regard to their location or the distance between them.

Chapter 3. Using the Attach Manager

An important LU 6.2 feature is the ability of a program in one node to start corresponding programs in other nodes. The *attach manager* handles incoming requests to start programs.

This chapter considers programs in your (local) workstation that start at the request of partner programs. The local program is referred to as *remotely started*. Workstation users and administrators want to control which programs can be remotely started for security and resource control. Users at remote nodes should not start programs that destroy data or use the local workstation's memory at critical times. The attach manager acts as a gate keeper, handling incoming requests to start programs on the local workstation.

The attach manager takes its name from an SNA message, called an *Attach*, that flows between a pair of LUs. An Attach flows when a program that uses the partner LU initiates a conversation. The LU 6.2 component in the local workstation passes any Attach it receives to its attach manager for handling. A received Attach is called an *incoming allocation request* or *incoming Attach*. In this chapter, the phrase *incoming allocation request* means that the SNA Attach is generated by a partner LU.

The attach manager does the following things:

- Enables remote nodes to start applications in the local workstation. Multiple instances of a program can be started in series (queued) or in parallel (nonqueued).
- Passes parameters to remotely started programs.
- Starts programs in windows or in the background.
- Uses security guidelines to verify incoming allocation requests.
- Forwards the incoming allocation request to the client workstations.
- Checks the conversation type (that is, basic or mapped) and synchronization level of incoming allocation requests.
- For server programs, specifies timeout values for holding incoming allocation requests and locally issued APPC **RECEIVE_ALLOCATE** verbs or CPI Communications Accept_Conversation or Accept_Incoming (CMACCP, CMACCI) calls.

Figure 7 illustrates the attach manager function.

© Copyright IBM Corp. 1994, 1997

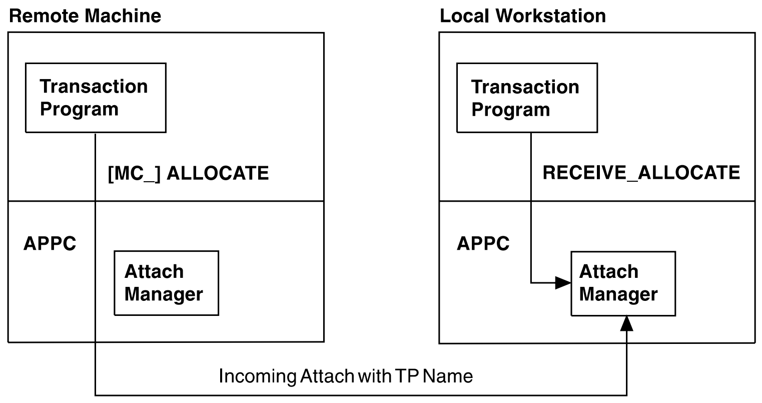


Figure 7. Attach Manager Function in APPC

In a communicating pair of transaction programs, only the node that receives allocation requests needs the attach manager. The attach manager manages three kinds of input:

- Incoming allocation requests (Attaches) from partner transaction programs
- APPC **RECEIVE_ALLOCATE** verbs or CPI Communications CMACCP and CMACCI calls from local programs
- Configuration definitions for transaction programs, user IDs, and passwords

The *TP name* is a key piece of information in an incoming allocation request. The attach manager uses the transaction program name to decide which program to start in the local workstation. Programmers and administrators at both nodes need to agree on each transaction program name. A program that issues an allocation request supplies a transaction program name as a parameter to the APPC **[MC_]ALLOCATE** or **[MC_]SEND_CONVERSATION** verb.

When an Attach is received, the transaction program name in the Attach is matched against transaction program names from the transaction definitions. If a match is found, the executable name from that definition is started or routed to a client workstation. If a match is not found, then the name of the executable is assumed to be the same as that which is specified in the Attach appended with **.EXE**.

Differentiating between an Application and a Transaction Program

The term *transaction program* has a special meaning in APPC. A transaction program is not an application; it is a section of an application.

A transaction program starts either when an application successfully issues an APPC **RECEIVE_ALLOCATE** or **TP_STARTED** verb. Both of these methods identify the transaction program as a new transaction program that APPC needs to know about. APPC reserves a group of memory blocks for the transaction program and creates a unique transaction program identifier, **tp_id**, which it returns to the calling program.

An application must supply a **tp_id** on all conversation verbs that it issues while it is a part of that transaction program. When an application issues a **TP_ENDED** verb, APPC clears its buffers for that transaction program and marks the **tp_id** as not valid. When an application terminates, APPC ends any active transaction programs associated with that process. When the attach manager receives an allocation request and ensures it is valid, and if a **RECEIVE_ALLOCATE** is not pending, it starts the application that corresponds to the incoming transaction program name. Notice that it starts a program, not a transaction program. Generally, the application then issues a verb that establishes it as a transaction program. Given mutual consent between the sending node and the local workstation, you can configure the attach manager to start *any* application in the local workstation.

Similarly, conversations have their own identifiers, and a transaction program must be established before a conversation can be allocated. Many conversations can use a single **tp_id** concurrently (such as in multiple threads) or sequentially (where one conversation follows another). When a transaction program ends, APPC deallocates any active conversations.

Transaction Program Definitions

Communications Server configuration uses two naming levels to identify the remotely started program:

- The 64-character name of the local program known by the partner transaction program (**tp_name**)
- The file specification of the local program to be started (filespec)

Using two names enables flexible reconfiguration that can increase the portability of your APPC programs among workstations.

TP name

The name that a partner transaction program sends in the allocation request to the attach manager in the local workstation.

The partner transaction program and the local program must both know the transaction program name. The transaction program name is a supplied parameter for **RECEIVE_ALLOCATE** verbs in programs on local LUs. Partner transaction programs supply a transaction program name with APPC **[MC_]ALLOCATE** or **[MC_]SEND_CONVERSATION** verbs.

Path name

The transaction program file specification (path name) names the program to start locally. The transaction program file specification contains the executable file's drive, path, file name, and extension.

Multiple transaction program definitions can specify the same transaction program file specification. The attach manager must determine whether to run one or multiple instances of a program, so a given transaction program file specification must be configured as either queued or nonqueued in *all* definitions that name it. For example, if a definition that specifies **MYTP.EXE** is configured as “queued—attach manager started”, **MYTP.EXE** cannot be configured as nonqueued in another transaction program definition. However, the transaction program filespec is case sensitive.

Identifying the Transaction Program Name on Both Machines

If the program identified by the attach manager cannot be started, the attach manager rejects the allocation request; the program that issued an allocation request is notified that the attach manager could not start the program.

Users or administrators define transaction programs during Communications Server configuration to build the list of defined transaction program names. Each unique transaction program name to be accepted from a partner requires a transaction program definition in the local (accepting) workstation unless you are willing to accept the default. The transaction program definition contains information about the transaction program. Similarly, during configuration, a list of security information (allowable passwords and user IDs) is built from the LU 6.2 conversation security information. Refer to *Communications Server: Quick Beginnings* for Communications Server configuration information. Following is a description of the configuration data that must be specified to define a transaction program.

Defining Conversation Attributes

The conversation parameters **sync_level**, **conv_type**, and **security_rqd** do not directly influence how the attach manager starts a program. However, the attach manager uses the parameters to determine whether to reject an incoming allocation request before queuing it, or checking for corresponding **RECEIVE_ALLOCATE** verbs.

Synchronization Level

Specify whether your transaction program will support the verbs and parameters for confirmation processing when you define **sync_level**. These APPC verbs are **[MC_]CONFIRM** and **[MC_]CONFIRMED**. Certain parameters on the **[MC_]ALLOCATE**, **[MC_]SEND_CONVERSATION**, **[MC_]PREPARE_TO_RECEIVE**, and **[MC_]DEALLOCATE** are for confirmation processing. For Common Programming Interface Communications (CPIC) users, **sync_level** can be set by the **Set_Sync_Level** (CMSSL) call.

Incoming allocation requests contain a field that indicates whether a partner transaction program issues verbs or parameters for confirmation processing. The attach manager checks the field on the incoming allocation request against the configured value in its list of transaction program definitions. If the values do not match, attach manager rejects the incoming allocation request. The possible configuration choices are:

NONE The transaction program does not issue any verb that relates to confirmation processing, in any of its conversations.

CONFIRM
The transaction program can perform confirmation processing on its conversations. The transaction program can issue verbs and recognize returned values that relate to confirmation. If the transaction program contains any of the verbs for confirmation processing, define **sync_level**(CONFIRM) to guarantee a compatible session.

EITHER
The transaction program can participate in conversations with partners that do or do not specify confirmation processing. Do not pick EITHER if the transaction program being configured requires confirmation processing.

Conversation Type and Style

The **conv_type** parameter is used to determine both the conversation type and conversation style of the program to be started. The conversation type attribute determines whether the program to be started supports basic or mapped records when it sends and receives data. The conversation style attribute determines whether the program to be started supports half-duplex conversations. The attach manager checks whether a transaction program uses basic or mapped verbs and if it uses half-duplex or full-duplex.

The conversation types are:

BASIC
The transaction program issues only basic conversation verbs for its conversations.

MAPPED
The transaction program issues only mapped conversation verbs for its conversations.

EITHER
The transaction program issues either basic or mapped conversation verbs for a conversation, depending on what arrives on the incoming allocation request.

The duplex types are:

HALF The transaction program supports half-duplex conversations only.

FULL The transaction program supports full-duplex conversations only.

EITHER
The transaction program supports either full or half duplex conversations.

Conversation Security for an Incoming Allocation Request

A transaction program definition can specify that incoming allocation requests must supply a password and user ID. The password and user ID are optional parameters on the **[MC_]ALLOCATE** and **[MC_]SEND_CONVERSATION** verbs or the CPIC calls Set_Conversation_Security_UserID (CMSCSU) and Set_Conversation_Security_PassWord (CMSCSP). If a local transaction program definition specifies conversation security, the attach manager validates the password and user ID of incoming allocation requests. The attach manager rejects the allocation request if a user ID and password are not present, or if they do not match a valid combination of passwords and user IDs.

The attach manager checks the validity of any incoming allocation requests that arrive with a password and user ID, even if the transaction program definition specifies that conversation security is not required. The allocation request is rejected if the password and user ID do not match a valid combination in the list. Thus, if a password or user ID arrives in an allocation request, it is never ignored.

Conversation Security for an Outgoing Allocation Request

A remotely started transaction program (one started by another transaction program) can validate a user ID and password before it allocates a conversation to a third transaction program. In such a case, the **security**(SAME) parameter in the **[MC_]ALLOCATE** and **[MC_]SEND_CONVERSATION** verbs can indicate that the conversation security is already verified. The second Attach automatically gets the user ID from the first Attach, that started the first conversation.

APPC can obtain the current user ID and send it, with an indicator that the user ID was validated. In the Attach for a locally started transaction program that uses the

security(SAME) parameter in either the **[MC_]ALLOCATE** or the **[MC_]SEND_CONVERSATION** verb, the partner must be able to accept the already validated indication.

See *Communications Server: System Management Programming* for more information about using the user ID and password.

Using the Attach Manager on the Communications Server

The following sections describe how to start programs located on the Communications Server machine.

Starting the Attach Manager

Users can start and stop the attach manager while the SNA node is active. Each time the attach manager starts, it begins to process incoming Attaches. When the attach manager stops, it purges any queued Attaches. Refer to *Communications Server: System Management Programming* for the applicable verbs.

The attach manager needs to be started only in nodes that run remotely started transaction programs. The attach manager does not need to be started in a node if all transaction programs in the node initiate conversations (that is, they all issue APPC **[MC_]ALLOCATE** or **[MC_]SEND_CONVERSATION** verbs). Communications Server node operations facility enables authorized users to start or stop the attach manager at any time. Authorized programs issue the Enable Attach Manager and Disable Attach Manager node operations verbs to start or stop the attach manager.

Starting Programs with the Attach Manager

When the attach manager starts a program on a workstation, it uses the **load_type** field in the defined transaction program list to decide how to run the program. A remotely started program can be configured to start in one of the following ways:

Console
An application that displays a window or runs as a full DOS application.

Background
The program starts in a background (detached) process. A background process should not issue any input or output calls to the keyboard, the mouse, or the display. If your program is completely debugged and requires no interactive user input, this option provides the fastest performance.

If the attach manager cannot start the program (for example, Communications Server cannot provide sufficient memory), the attach manager rejects the incoming allocation request.

If a transaction program issues a **RECEIVE_ALLOCATE** call and specifies a transaction program name that has not previously been defined, the system performs an implicit definition of the transaction program and assigns default values to the parameters.

The defaults used are:

Attach timeout	= 0	(no timeout is applied)

Receive Allocate timeout	= 0	(no timeout is applied)
Attach Manager dynamically loaded	= Yes	(the transaction program can be loaded by the attach manager)

These defaults mean that if you issue a call to **RECEIVE_ALLOCATE** as previously described, it will not complete until an attempt is made to attach to the named transaction program, or you can cancel the call.

Matching Incoming Allocation Requests with RECEIVE_ALLOCATE Verbs

A remotely started program in a local workstation normally issues an APPC **RECEIVE_ALLOCATE** verb to start both a transaction program and a conversation. The APPC **RECEIVE_ALLOCATE** verb specifies the same transaction program name that the remote transaction program specified in its APPC **[MC_]ALLOCATE** or **[MC_]SEND_CONVERSATION** verb. APPC passes the **RECEIVE_ALLOCATE** verb to the attach manager for processing. When the attach manager sees a **RECEIVE_ALLOCATE** verb that matches a received Attach (and the attach manager performs several cross-checks), it signals APPC that a conversation can begin. At this point, the attach manager ends its involvement in the conversation.

During transaction program configuration, you have two choices for handling multiple incoming allocation requests for the same program. You can run multiple instances of the same program concurrently in the local workstation (*nonqueued* operation), or you can run one instance of the same program at a time (*queued* operation). These values are configured in the **queued** and **dynamic load** parameters, that have the following options:

- Nonqueued—attach manager started
- Queued—attach manager started
- Operator started

Nonqueued Programs

When a program is configured as nonqueued, each incoming allocation request causes the attach manager to load and execute another instance of the program associated with the incoming transaction program name.

The attach manager holds valid incoming allocation requests indefinitely, waiting for a matching **RECEIVE_ALLOCATE** verb from the program it started. If that program fails to issue a **RECEIVE_ALLOCATE** verb (for example, it loops in the code that precedes the **RECEIVE_ALLOCATE** verb), the attach manager holds the allocation request until the process terminates.

Queued Programs

Queued programs can start in one of two ways:

Attach manager started
The program is started by the attach manager.

Operator started
The program is to be started by another program in the workstation or by an operator.

The attach manager maintains two queues for each queued transaction program name in the defined transaction program list. One queue is for incoming allocation requests; the other is for **RECEIVE_ALLOCATE** verbs. For example, when an incoming allocation request arrives, the attach manager starts the corresponding local program or sends a message to the operator. The node holds the incoming allocation request until the program that the attach manager started issues a matching **RECEIVE_ALLOCATE** verb or until a timeout occurs. The node uses the value configured for the **incoming_alloc_timeout** parameter to determine when time-outs occur. Other allocation requests can arrive for that transaction program or for another transaction program. The other programs wait in their respective queues until a matching **RECEIVE_ALLOCATE** verb is issued, or until they time out.

Local programs can issue **RECEIVE_ALLOCATE** verbs before any matching allocation request arrives. The attach manager holds the **RECEIVE_ALLOCATE** verb on its respective queue and waits for an allocation request to arrive from a partner LU. Each queue has a timeout value; the **rcv_alloc_timeout** parameter specifies how long a **RECEIVE_ALLOCATE** verb can wait on a queue before the verb times out. The attach manager returns queued **RECEIVE_ALLOCATE** verbs to the associated programs with an ALLOCATE_NOT_PENDING return code. The timeout value for **RECEIVE_ALLOCATE** verbs can be 0 to enable programs to check whether any allocation requests are queued, and, if not, to continue other processing.

The **RECEIVE_ALLOCATE** verb can be issued as a nonblocking verb. This enables the transaction program to service multiple conversations from a single thread in a single process.

When **RECEIVE_ALLOCATE** is issued as a nonblocking verb, the attach manager returns control to the transaction program immediately; the transaction program need not remain in a wait state while waiting for the matching incoming allocation request to arrive. Instead, the transaction program can perform other work, and choose when to wait for the matching incoming allocation request.

The transaction program can queue multiple nonblocking **RECEIVE_ALLOCATE** verbs for different conversations. The maximum number of verbs that can be queued is limited only by resource constraints. A nonblocking **RECEIVE_ALLOCATE** verb will remain on the attach manager's **RECEIVE_ALLOCATE** verb queue until either the matching allocation request arrives or the verb times out, that is, the **rcv_alloc_timeout** value has been reached.

The attach manager saves the information that identifies the transaction program when a queued program issues a valid **RECEIVE_ALLOCATE** verb call for a transaction program. When the queued program ends, the attach manager examines the queue of allocation requests for that transaction program. If the queue is not empty, the attach manager starts a new instance of the program, or sends a message that directs the operator to start the program.

You should configure the maximum size of the incoming allocation request queue for each transaction program. Resource constraints limit the number of queued **RECEIVE_ALLOCATE** verbs.

The following two cases summarize queued operations:

Case 1:
One or more incoming allocation requests arrive before a

RECEIVE_ALLOCATE verb or CPI Communications CMACCP call is issued for a given transaction program. The attach manager queues the incoming allocation requests (for a time specified by a configured timeout value) until a **RECEIVE_ALLOCATE** verb is issued. The first incoming allocation request satisfies the **RECEIVE_ALLOCATE** verb.

Case 2:
A **RECEIVE_ALLOCATE** verb is issued before an incoming allocation request arrives for a given transaction program. The attach manager queues the **RECEIVE_ALLOCATE** verb (for a time specified by a configured timeout value) until an incoming allocation request arrives. In certain cases, more than one **RECEIVE_ALLOCATE** verb might be issued and queued before an incoming allocation request arrives. Each new incoming allocation request satisfies the next **RECEIVE_ALLOCATE** verb in the queue.

Table 7 on page 27 provides a summary of the verbs and incoming allocation requests associated with **queued** and **dynamic load** parameter values.

Table 7. Verb Processing and Transaction Program Name Configuration

	Transaction Program Operation		
Verb Processing	**Nonqueued—attach manager started**	**Operator started**	**Queued—attach manager started**
Incoming allocation request with pending **RECEIVE_ALLOCATE** verb.	Cannot occur; no queue of pending **RECEIVE_ALLOCATE** verbs.	OK **RECEIVE_ALLOCATE** verb.	OK **RECEIVE_ALLOCATE** verb.
Incoming allocation request without pending **RECEIVE_ALLOCATE** verb.	Load and execute another program instance. Hold incoming allocation request. Wait for **RECEIVE_ALLOCATE** verb.	Put incoming allocation request on queue unless queue is full. Wait for **RECEIVE_ALLOCATE** verb or for allotted time to expire.	If program is not started, load and execute it. Put incoming allocation request on queue unless queue is full. Wait for **RECEIVE_ALLOCATE** verb or for allotted time to expire.
RECEIVE_ALLOCATE verb with incoming allocation request pending.	OK **RECEIVE_ALLOCATE** verb.	OK **RECEIVE_ALLOCATE** verb.	OK **RECEIVE_ALLOCATE** verb.
RECEIVE_ALLOCATE verb with no pending incoming allocation request.	Cannot occur; pending allocation requests for nonqueued operations cannot run out of time.	Hold **RECEIVE_ALLOCATE** verb. Wait for incoming allocation request or for allotted time to expire.	Hold **RECEIVE_ALLOCATE** verb. Wait for incoming allocation request or for the allotted time to expire.
Transaction program operation ends.	Nothing happens.	Nothing happens.	If there is a pending allocation request, reload the program; otherwise, reload on the next incoming allocation request.

Using the Attach Manager on the Communications Server SNA API Clients

The following sections describe how to start programs that are located on SNA API machines.

Defining Transaction Programs for SNA API Clients

Transaction programs located at client machines require transaction program definitions on both the Communications Server and client machines in order to be remotely started. Following is the transaction program information required on the server:

- Transaction program name
- Conversation type
- Synchronization level
- Whether or not conversation security is required

Communications Server will verify this information when the incoming allocate arrives. In addition, the local LU that receives the incoming allocation request must be enabled to route the request to the client machine.

The client attach manager must have a transaction program defined so that it knows how to start the requested program. Following is the transaction program information required on the client:

- Transaction program name
- The local LU that receives the incoming allocation request
- The path name of the program
- Any parameters that need to be passed to the transaction program

Once these definitions are complete and the client attach manager is started, incoming allocates for transaction programs located on client machines will be routed to the client for processing.

Starting the SNA API Client Attach Manager

To start the client attach manager, click the attach manager icon located in the Communications Server for SNA client folder. This will connect the attach manager to the configured Communications Server and send the list of transaction definitions that have been defined for that client.

Chapter 4. Writing a Transaction Program

This chapter describes issues to consider when planning and writing transaction programs to APPC. When developing a transaction program, you must choose between certain design alternatives. The following list describes the design issues to consider:

- Choosing either basic or mapped conversations
- Deciding whether to start conversations with or without confirmation
- Using the security features
- Providing for conversion of ASCII names and data (if necessary)

The first part of this chapter provides background information on the application protocols, conversation states, Communications Server support tasks, and data formats. The rest of this chapter describes specific requirements for developing a transaction program.

Application Protocols

Communications Server LU 6.2 enables program-to-program communication. The design of your program depends on the protocols that you define and the communication that your program must accomplish.

In addition to any rules that you define for your program, Communications Server LU 6.2 defines rules that your program must follow when using a conversation. To enforce these rules, Communications Server LU 6.2 manages the state of your conversation and allows your program to perform certain operations only when the conversation is in the correct state. For example:

- Your program cannot send data unless it has permission to send.
- Your program cannot receive data unless the partner program has permission to send.
- Your program cannot use a conversation after it has been deallocated.

Communications Server LU 6.2 manages and enforces the conversation states shown in Table 8.

Table 8. Conversation States

State	Definition
Reset	The conversation does not exist.
Send	The program can send data, request confirmation, or deallocate the conversation.
Receive	The program can receive information from the partner program.
Confirm	The program can reply to a confirmation request.

Available Communications Server LU 6.2 Services

This section describes the Communications Server LU 6.2 services that your transaction program can use to communicate with another transaction program.

Allocate a Conversation

Requests the local LU to start a conversation with a partner transaction program in a partner LU.

Corresponding verbs: ALLOCATE and MC_ALLOCATE.

Send Data

Sends data to the partner program.

Corresponding verbs: SEND_DATA and MC_SEND_DATA.

Force Data in the Internal Buffers to Be Sent

Forces the LU to send to the partner program all data it is holding in an internal buffer.

Note: You do not normally have to use this service to cause the LU to send the data. The LU automatically sends the data it stores in an internal buffer when the buffer is full or when it determines that your program has finished sending.

Corresponding verbs: FLUSH and MC_FLUSH.

Receive Data

Receives data from the partner program.

Corresponding verbs: RECEIVE_AND_WAIT, RECEIVE_IMMEDIATE, MC_RECEIVE_AND_WAIT, and MC_RECEIVE_IMMEDIATE.

Request Permission to Send

Requests from the partner program permission to send data.

Corresponding verbs: REQUEST_TO_SEND and MC_REQUEST_TO_SEND.

Grant Permission to Send

Gives the partner program permission to send data.

Corresponding verbs: PREPARE_TO_RECEIVE and MC_PREPARE_TO_RECEIVE.

Request Confirmation

Requests the partner program to confirm that all data has been received and processed successfully.

Corresponding verbs: CONFIRM and MC_CONFIRM.

Accept or Reject Confirmation

Sends a reply to a confirmation request.

Corresponding verbs: CONFIRMED, MC_CONFIRMED, SEND_ERROR, and MC_SEND_ERROR.

Request to Be Posted When Information Is Available

Requests that the LU post an event when the conversation has information available to be received.

Corresponding verb: RECEIVE_AND_POST.

Report an Error

Reports that an error has occurred.

Corresponding verbs: SEND_ERROR and MC_SEND_ERROR.

Obtain Conversation Attributes

Obtains the attributes of a conversation. These attributes include
- Name of the local LU
- Name of the partner LU
- Name of the session's transmission service mode
- Type of confirmation protocols supported by the conversation
- Type of conversation

Corresponding verbs: GET_ATTRIBUTES, MC_GET_ATTRIBUTES, and GET_TYPE.

Deallocate a Conversation

Ends a conversation with the partner program.

Corresponding verbs: DEALLOCATE and MC_DEALLOCATE.

Choosing a Conversation Type

This section discusses issues you should consider when choosing between basic and mapped conversations.

Consistency of Conversation Type

The conversation type you use, designated by the ALLOCATE verb, must be consistent for the entire conversation. You cannot use basic conversation verbs for some requests and mapped conversation verbs for other requests. Communications Server LU 6.2 rejects the verbs if you change from one type of verb to another within a conversation. A remotely initiated transaction program can issue the GET_TYPE verb to determine the conversation type.

A program can issue only basic conversation verbs for a basic conversation. A program using a mapped conversation can issue either basic or mapped verbs. It must, however, issue verbs of only one format, either basic or mapped.

You can provide your own mapped conversation support using only basic conversation verbs for a conversation designated as mapped. If you choose to provide your own mapped conversation support, your program must conform to the mapped conversation formats and protocols.

See the *SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2* and the *Systems Network Architecture LU 6.2 Reference: Peer Protocols* for more information on mapped conversation formats and protocols.

Sending Data

Use a basic conversation when you need to optimize your program's performance by sending the data from a buffer that contains more than one logical record or a partial logical record. Basic conversations can improve your program's execution efficiency by enabling your program to send several logical records with one request.

To use the basic conversation, your program must provide a 2-byte *logical length field (LL field)* at the beginning of every logical record where

- The last 15 bits of the LL field contain a binary value equal to the length of the logical record, including the 2-byte length field. The 15-bit limit restricts the value to a maximum of 32,767 (32,765 bytes of user data plus the 2-byte length field). If you use a value larger than 32,767, Communications Server LU 6.2 cannot detect the error and uses the last 15 bits of the LL field anyway.

  The smallest value possible is 2 (the LL field followed by no data). If you use a value that is less than 2, Communications Server LU 6.2 indicates an error.
- Communications Server LU 6.2 ignores the first bit of the LL field. This bit is a concatenation indicator. If the concatenation indicator is set, the transaction program must append the data from the following logical record to the data received up to that point. This concatenation process should continue until the transaction program receives a record in which the concatenation indicator is not set. This definition allows you to use higher level records (GDS variables) that are longer than 32,767 bytes.
- You must manage the reversal of byte values in your PC.

  The PC stores all numeric 16- or 32-bit values with the low-order (least significant) byte stored in the lower numbered address. Therefore, if a transaction program computes the length of a logical message and stores that value as the LL field, the 2 bytes appear in memory with the low-order byte first, and your PC will send the bytes in this order (incorrectly) over the communication line.

  The transaction program is responsible for putting all transaction-level data, including LL fields, in the correct order (high-order byte first).

Use a mapped conversation if you do not need to send partial logical records or more than one logical record. When you send data with the mapped conversation verbs, Communications Server LU 6.2 assumes that the buffer contains exactly one complete higher level record (GDS variable). The mapped conversation support automatically provides length fields in the correct byte-reversed order and uses concatenated logical records as needed.

Receiving Data

Use a basic conversation when you need to receive more than one logical record in one buffer. This option can improve your program's execution efficiency by enabling it to receive several logical records with one request (the BUFFER option).

When you use this basic conversation feature, Communications Server LU 6.2 places the logical records in your buffer with the 2-byte LL fields intact. The bytes are reversed from normal IBM-compatible PC order.

Your program must examine the returned fields of the verb to determine if it has received a complete logical record and, if so, where the next logical record begins. Communications Server LU 6.2 provides the rest of an incomplete logical record after a subsequent request to receive data.

If you want to receive one higher/user level record with a single request, use a mapped conversation. As you receive data with the mapped conversation verbs, Communications Server LU 6.2 ends the receive operation when your program receives a complete higher/user level record or when your buffer is full. Communications Server LU 6.2 supplies a return code when it fills your buffer before your program has received a complete logical record.

Your program can receive the rest of the higher/user level record by issuing a subsequent request to receive data. The Communications Server LU 6.2 mapped conversation support removes any length fields and automatically concatenates logical records as necessary.

Reporting Errors and Abnormal Termination

Use a basic conversation for the following reasons:

- To distinguish between errors detected by your program and errors detected by an application that is using your program
- To distinguish between an abnormal termination caused by your program and one caused by an application using your program

When reporting an error or when abnormally terminating a conversation with an LU service program, the basic conversation verbs enable you to indicate which program detected the error. When the partner LU reports the error to the partner program with a return code, the value of the return code indicates where Communications Server LU 6.2 detected the error.

If you do not need to distinguish between errors detected by your program and errors detected by other applications, use a mapped conversation. The mapped conversation verbs assume that your program detected the error.

Sending an Error Log Data Record

Use a basic conversation to send a log record when you detect an error or abnormally terminate a conversation. The basic conversation verbs enable you to specify an error log GDS variable when you report an error or abnormally terminate a conversation. Communications Server LU 6.2 sends this log record to the local log and to the partner LU to be recorded in that log. This feature is useful when your program detects a critical or unrecoverable error and you want the program to record the event to help determine the problem.

If you send an error log GDS variable, the format of the record must conform to the formats defined by SNA. See the IBM *Systems Network Architecture Formats* for more information on the error log GDS variable format.

Use a mapped conversation if you do not need to send a log record when you detect an error or abnormally terminate a conversation. The mapped conversation verbs assume that your program does not need to record error data in the log to help determine the problem.

Abnormally Terminating because of a Timeout

To indicate that your program has abnormally terminated the conversation because of a timeout, use a basic conversation. When abnormally terminating your conversation, the basic conversation verbs enable you to indicate that your program is abnormally terminating the conversation because the partner program has not done the necessary processing in the time allowed. When Communications Server LU 6.2 reports the error to the partner transaction program, the return code value indicates that a timeout caused the abnormal termination.

If you do not need to report the cause of an abnormal termination, use a mapped conversation. The mapped conversation verbs assume that your program requested the abnormal termination because of a critical or unrecoverable error.

Requesting Confirmation

Requesting confirmation is an efficient way to determine that the partner program has received all the data sent so far. If you plan to request confirmation during the conversation, the allocation transaction must indicate this fact when you request the allocation of the conversation.

If you use conversation verbs that do not request confirmation, you should not request the allocation of a conversation supporting confirmation services.

You can write a transaction program to participate in conversations that use confirmation requests and in conversations that do not use confirmation requests.

Choosing a Transaction Program Name

When you name a transaction program, choose a name that has a first character with an EBCDIC code greater than an EBCDIC blank (X'40'). Transaction program names containing first characters with EBCDIC codes less than X'40' are reserved for service transaction programs. Transaction program names can include up to 64 characters.

Using the Security Features

Communications Server LU 6.2 provides one of two types of security functions: partner LU verification and end-user verification. Partner LU verification is a session-level security protocol that takes place at the time the session is activated. End-user verification is a conversation-level security protocol that takes place at the time a conversation is started.

Partner LU Verification (Session-Level Security)

Partner LU verification is performed by an exchange of security information between the two LUs. This exchange is called session-level security. This level of security is generally required when the communications network is not physically secure. The local LU and the remote LU each provide a password, and the Communications Server LU 6.2 performs encryption for password verification. It is recommended, but not required, that each LU pair have a unique password.

End-User Verification (Conversation-Level Security)

End-user verification is used to enable the requested application subsystem to verify the identity of the requester before providing access to the requested transaction program and its resources. The security information exchanged can include a user ID and a password. The user IDs provided by conversation-level security can also be used for auditing and accounting purposes.

In conversation-level security, the requesting transaction program provides the security information on the ALLOCATE verb, and the remote application subsystem performs the verification. If the requesting transaction program has not supplied the correct user ID and password, the remote application subsystem rejects the request.

An intermediate transaction program (one started by another transaction program) that requires conversation-level security can be used to access an additional transaction program that requires conversation-level security. In this case, an already-verified indicator is set in the allocation request for the additional transaction program. The user ID saved from the first request, which initiated the intermediate transaction program, is automatically supplied in the second request.

Converting between EBCDIC and ASCII

Communications Server LU 6.2 assumes that the interface between it and the transaction program (or the application subsystem) uses EBCDIC characters where specified by the verb. Your program can send some data or parameters to a system services control point (SSCP) or partner LU, where they will be compared to table values expressed in EBCDIC. These values include the transaction program name, the partner LU name supplied on ALLOCATE, the mode name, the user ID, and the user password.

Incoming names are in EBCDIC, and your program must translate them to ASCII (if the application subsystem stores them in ASCII). Therefore, both the application subsystem and the transaction program must be prepared to perform conversions between ASCII and EBCDIC.

Do not avoid this conversion by defining private protocols between two PCs. Private protocols can cause unexpected difficulties and hinder communication with other types of computers. For example, using an ASCII “k” (X'4B') in an LU or a network name can cause protocol errors or prevent the name from being recognized because the EBCDIC period (.) used in a fully qualified name also has the value X'4B'.

LU name, mode name, net name, and partner LU name must be 8 EBCDIC characters long (padded on the right with blanks if the name is less than 8 characters). These names must be type A symbol strings, with the first character non-numeric, consisting of any combination of

- Uppercase letters
- Numerics
- Special characters $, #, and @

Whether a transaction program needs to translate data depends on a private agreement between the partner transaction programs. If your program is communicating with a node that normally uses EBCDIC, you should convert data to EBCDIC as appropriate.

As a convenience, Communications Server LU 6.2 provides the CONVERT verb, which converts ASCII codes to EBCDIC or EBCDIC codes to ASCII. For more information, see “CONVERT” on page 283.

Chapter 5. Implementing APPC Transaction Programs

This chapter describes the implementation of APPC Transaction Programs using the dynamic link library (DLL) file provided.

The Communications Server implementation of APPC is designed to be binary compatible with Microsoft** NT SNA Server on Windows machines, and similar to the implementation of the APPC interface of OS/2 Communications Manager/2 Version 1.0.

Writing Transaction Programs

Communications Server provides a dynamic link library (DLL) file that handles APPC verbs.

The DLL is reentrant; multiple application processes and threads can call the DLL concurrently.

APPC verbs have a straightforward language interface. Your program fills in fields in a block of memory called a *verb control block* (VCB). Then it calls the APPC DLL and passes a pointer to the verb control block. When its operation is complete, APPC returns, having used and then modified the fields in the VCB. Your program can then read the returned parameters from the verb control block.

Table 9 shows source module usage of supplied header files and libraries needed to compile and link APPC programs. Some of the header files may include other required header files.

Table 9. Header Files and Libraries for APPC

Operating System	Header File	Library	DLL Name
WINNT & WIN95	WINAPPC.H	WAPPC32.LIB	WAPPC32.DLL
WIN3.1	WINAPPC.H	WINAPPC.LIB	WINAPPC.DLL
OS/2	APPC_C.H	APPC.LIB	APPC.DLL

Option Sets Supported

Communications Server supports the following APPC option sets. See *SNA Transaction Programmer's Reference* for LU type 6.2 for a fuller description of each option set.

101	Flush the LU send buffer.
102	Get attributes.
103	Post on receipt with test for posting (through the **RECEIVE_AND_POST** verb).
104	Post on receipt with wait (through the **RECEIVE_AND_POST** verb).
105	Prepare to receive.
106	Receive immediate.
109	Get transaction program name and instance identifier.
110	Get conversation type.
112	Full-duplex conversation and expedited data.

© Copyright IBM Corp. 1994, 1997

Option 112 is only supported for applications residing on the server.

113	Nonblocking support.
201	Queued allocation of a contention-winner session.
203	Immediate allocation of a session.
204	Conversations between programs located at the same LU.
205	Queued allocation or when session is free.
211	Session level LU-LU verification.
212	User ID verification.
213	Program-supplied user ID and password.
214	User ID authorization.
241	Send PIP data.
242	Receive PIP data.
243	Accounting.
244	Long locks.
245	Test for request-to-send received.
247	User control data.
251	Extract translation and conversation correlator.
290	Logging of data in a system log.
291	Mapped conversation LU services component.
401	Reliable one-way brackets.
501	**CHANGE_SESSION_LIMIT** verb.
502	**ACTIVATE_SESSION** verb.
504	**DEACTIVATE_SESSION** verb.
505	**LU**-*definition* verb.
601	**MIN_CONWINNERS_TARGET** parameter.
602	**RESPONSIBLE(TARGET)** parameter.
603	**DRAIN_TARGET(NO)** parameter.
604	**FORCE** parameter.
605	LU-LU session limit.
606	Locally known LU names.
607	Uninterpreted LU names.
610	Maximum RU size bounds.
612	Contention winner automatic activation limit.
613	Local maximum (LU,mode) session limit.

616 CPSVCMG mode name support.

Full-Duplex VCBs

This is only supported for applications residing on the server.

To identify definitions for the format 1 VCB that are needed for full-duplex conversations and to send and receive expedited data, the transaction program must define a compiler constant called WINAPPC_FORMAT_1 before including the WINAPPC.H header file. This can be achieved in C language as follows:

```
#define  WINAPPC_FORMAT_1
#include  <winappc.h>
```

If this constant is not defined, only the format zero versions of the VCBs will be accessible from the application.

Queue-Level Nonblocking

This is only supported for applications residing on the server.

The Communications Server APPC API supports queue-level nonblocking. This support is provided through the APPC entry point.

Nonblocking operation enables control to be returned to the application if processing of a verb cannot be completed immediately, so that the application can continue other processing until it is notified that the outstanding verb has completed. Queue-level nonblocking means that the application can issue nonblocking verbs for different queues and have the verbs processed simultaneously by Communications Server. The application can also issue a succession of nonblocking verbs for a given queue without waiting for any of the verbs to complete.

Communications Server maintains six queues for nonblocking verbs:

- An allocate queue (one for each active transaction program)
- A send/receive queue (one per conversation, half-duplex only)
- A send queue (one per full-duplex conversation)
- A receive queue (one per full-duplex conversation)
- A send-expedited queue (one per conversation)
- A receive-expedited queue (one per conversation)

All six queue types can hold an unlimited number of verbs. Nonblocking verbs are queued if another (blocking or nonblocking) verb is being processed by Communications Server. Verbs in an allocate queue are processed concurrently, whereas verbs in the other queues are processed one at a time, in the order in which they are received by Communications Server.

The application notifies Communications Server that it wants a verb to be processed in nonblocking mode by setting a flag in the **opext** field, **AP_NON_BLOCKING**. The application can supply an event handle with any nonblocking verb that is used to notify the application of asynchronous verb completion. This handle is passed to Communications Server in the **SECONDARY_RC** field. If no handle is specified, the application is notified that the verb has completed when the next verb on that queue specifies that a handle completes.

It is guaranteed that all preceding verbs with no handle are complete when the event is signaled after completion of a verb on the same queue that does not specify a handle.

When a nonblocking verb returns the flag **AP_OPERATION_INCOMPLETE_FLAG**, it is set in the **opext** field.

The APPC verbs that can be issued in nonblocking mode on the allocate queue are:

(MC_)ALLOCATE

(MC_)SEND_CONVERSATION

The APPC verbs that can be issued in nonblocking mode on the send/receive queue are:

(MC_)CONFIRM

(MC_)CONFIRMED

(MC_)DEALLOCATE

(MC_)FLUSH

(MC_)PREPARE_TO_RECEIVE

(MC_)RECEIVE_AND_WAIT

(MC_)RECEIVE_IMMEDIATE

(MC_)SEND_DATA

(MC_)SEND_ERROR

The APPC verbs that can be issued in nonblocking mode on the send queue (for full-duplex conversations) are:

(MC_)DEALLOCATE

(MC_)FLUSH

(MC_)SEND_DATA

(MC_)SEND_ERROR

The APPC verbs that can be issued in nonblocking mode on the receive queue (for full-duplex conversations) are:

(MC_)RECEIVE_AND_WAIT

(MC_)RECEIVE_IMMEDIATE

The APPC verb that can be issued in nonblocking mode on the receive-expedited queue (for full-duplex conversations) is:

(MC_)RECEIVE_EXPEDITED_DATA

The APPC verbs that can be issued in nonblocking mode on the send-expedited queue are:

(MC_)REQUEST_TO_SEND

(MC_)SEND_EXPEDITED_DATA

The following APPC verbs are always processed asynchronously but are not associated with any queue:

(MC_)RECEIVE_AND_POST

(MC_)TEST_RTS_AND_POST

The Communications Server APPC verbs that cannot be issued in nonblocking mode (and are processed in blocking mode if the application sets the nonblocking flag) are:

(MC_)GET_ATTRIBUTES

GET_TP_PROPERTIES

GET_TYPE

RECEIVE_ALLOCATE

TEST_RTS

TP_ENDED

TP_STARTED

CNOS

An application cannot issue verbs in nonblocking mode for the send/receive queue or the send-expedited queue until an **ALLOCATE** or **RECEIVE_ALLOCATE** verb has returned successfully (Communications Server returns AP_PARAMETER_CHECK, and AP_BAD_CONV_ID).

A nonblocking verb issued for the send/receive queue or the send-expedited queue, with another (blocking or nonblocking) verb currently outstanding on the same queue, is added to that queue, and is only processed when the outstanding verb has completed.

A blocking verb issued when any other verb (for the same conversation) is outstanding is rejected by Communications Server (with **primary_rc** AP_TP_BUSY). Note that **RECEIVE_AND_POST** is treated as a blocking verb in this respect, but **TEST_RTS_AND_POST** can be issued with other verbs outstanding on the same conversation (and is not placed in any of the nonblocking verb queues). A blocking verb issued when there are no verbs on the same queue is processed as normal even though there may be verbs on other queues. Note that TEST_RTS, GET_ATTRIBUTES, GET_STATE and GET_TYPE are not associated with a queue and may be executed at any time and will never return AP_TP_BUSY.

Default Local LU

Communications Server supports default local LUs for both dependent and independent LU 6.2. The default LU is used when the **TP_STARTED** verb (see “TP_STARTED” on page 78) is issued with a blank **lu_alias** field. For independent LU 6.2, the default LU is the control point LU. For dependent LU 6.2, a local LU pool is used. See *Communications Server: System Management Programming* for details on the **DEFINE_LOCAL_LU** verb. Communications Server chooses an LU from the default pool, or uses the control point LU, as follows:

- If LUs have been configured as members of the default local LU pool, Communications Server chooses an LU from the pool that is not in use. If all the LUs in the pool are in use, the **TP_STARTED** verb fails.
- If No LUs have been configured as members of the default local LU pool, Communications Server uses the control point LU.

Chapter 6. Implementing CPI-C Programs

This chapter documents the details of the Communications Server support for the CPI-C interface. It covers these main areas:

- Techniques for compiling and linking CPI-C programs
- Methods of preparing and executing CPI-C programs
- Features of the CPI-C versions supported by Communications Server

The Communications Server implementation of CPIC is designed to be binary compatible with Microsoft** NT SNA Server on Windows machines, and similar to the implementation of the CPIC interface of OS/2 Communications Manager/2 Version 1.0.

Writing CPIC Programs

Communications Server provides a dynamic link library (DLL) file that handles CPIC calls.

The DLL is reentrant; multiple application processes and threads can call the DLL concurrently.

Table 10 shows source module usage of supplied header files and libraries needed to compile and link CPIC programs. Some of the header files may include other required header files.

Table 10. Header Files and Libraries for CPIC

Operating System	Header File	Library	DLL Name
WINNT & WIN95	WINCPIC.H	WCPIC32.LIB	WCPIC32.DLL
WIN3.1	WINCPIC.H	WINCPIC.LIB	WINCPIC.DLL
OS/2	CPIC_C.H	CPIC16.LIB or CPIC32.LIB	CPIC.DLL

CPI-C Versions

The CPI-C interface has gone through several version changes and extensions. You should be aware of these versions for two reasons:

- If you are maintaining or porting an existing program, you need to know which function calls are portable and which you might need to change if you change versions.
- If you are writing a new program, you need to be aware when you are writing code that is dependent on a particular version.

Communications Server CPI-C Conformance Class Support

The following CPI-C 2.1 conformance classes are supported as defined by the IBM document *Common Programming Interface Communications CPI-C Reference* Version 2.1 (SC26-4399-08).

© Copyright IBM Corp. 1994, 1997

For details on which classes are supported by client and server applications, see the *ringing telephone*.

The **conversation conformance** class allows programs to start and end half-duplex conversations.

Starter Set calls:

CMACCP
Accept_Conversation

CMALLC
Allocate

CMDEAL
Deallocate

CMINIT
Initialize_Conversation

CMRCV
Receive

CMSEND
Send_Data

Advanced Function Calls:

CMCFM
Confirm

CMCFMD
Confirmed

CMECS
Extract_Conversation_State

CMECT
Extract_Conversation_Type

CMEMBS
Extract_Maximum_Buffer_Size

CMEMN
Extract_Mode_Name

CMESL
Extract_Sync_Level

CMFLUS
Flush

CMPTR
Prepare_To_Receive

CMRTS
Request_To_Send

CMSERR
Send_Error

CMSCT
Set_Conversation_Type

CMSDT
Set_Deallocate_Type

CMSF Set_Fill

CMSLD
Set_Log_Data

CMSMN
Set_Mode_Name

CMSPTR
Set_Prepare_To_Receive_Type

CMSRT
Set_Receive_Type

CMSRC
Set_Return_Control

CMSST
Set_Send_Type

CMSSL
Set_Sync_Level

Required sync_level values:

CM_NONE or CM_CONFIRM

CMSTPN
Set_TP_Name

CMTRTS
Test_Request_To_Send_Received

LU 6.2 conformance class allows a program to use LU 6.2 specific services:

CMEPLN
Extract_Partner_LU_Name

CMSED
Set_Error_Direction

CMSPLN
Set_Partner_LU_Name

The **conversation-level non-blocking** conformance class allows a program to regain control if a call cannot complete immediately.

CMCANC
Cancel_Conversation

CMSPM
Set_Processing_Mode

CMWAIT
Wait_For_Conversation

The **server** conformance class allows a program to register multiple transaction program names with CPI-C, to accept multiple incoming conversations, and to manage contexts for different clients.

CMACCI
Accept_Incoming

CMECTX
Extract_Conversation_Context

CMEXTX Extract_Conversation_Context is not supported for the Windows 3.1 client.

CMETPN
Extract_TP_Name

CMRLTP
Release_Local_TP_Name

CMINIC
Initialize_For_Incoming

CMSLTP
Specify_Local_TP_Name

The **data conversion** conformance class routine allows a program to call local routines to change the encoding of a character string from the local encoding to EBCDIC, or vice versa.

CMCNVI
Convert_Incoming

CMCNVO
Convert_Outgoing

The **security** conformance class allows a program to establish conversations that use access security information in side information or set directly by the program.

CMESUI
Extract_Security_User_ID

CMSCSP
Set_Conversation_Security_Password

CMSCST
Set_Conversation_Security_Type

Required conversation_security_type values:

CM_SECURITY_NONE

CM_SECURITY_PROGRAM

CM_SECURITY_PROGRAM_STRONG

CM_SECURITY_SAME

CMSCSU
Set_Conversation_Security_User_ID

Queue-Level Non-Blocking for regain of control if a call cannot complete.

CMCANC
Cancel_Conversation

CMSQPM
Set_Queue_Processing_Mode

CMWCMP
Wait_For_Completion

Queue-Level Non-Blocking is not supported for the Windows 3.1 client.

Callback Function for regaining control if a call cannot complete.

CMCANC
Cancel_Conversation

CMSQCF
Set_Queue_Callback_Function

Callback Function is not supported for the Windows 3.1 client or OS/2.

Secondary Information allows you to extract secondary error return information.

CMESI
Extract_Secondary_Information

Secondary Information is not supported for the Windows 3.1 client.

The following classes are supported only on the Communications Server:

Full-Duplex allows user access to full-duplex conversations.

CMESRM
Extract_Send_Receive_Mode

CMSSRM
Set_Send_Receive_Mode

Expedited Data exchanges expedited data with the partner program.

CMRCVX
Receive_Expedited_Data

CMSNDX
Send_Expedited_Data

The following Conformance Classes are not supported.

OSI TP services
Recoverable Transactions (for resource recovery interface)
Unchained Transactions (for recoverable transactions)

Distributed Security (user security services of distributed security server)
Directory (user designated information stored in a distributed directory)

CPI-C Functions

All the CPI-C functions supported by Communications Server are listed in Table 11. Use this table for reference when you are maintaining an old program or when you are writing a new program that must remain compatible with some existing system.

Table 11. Communications Server Client Support of CPI-C Functions

Function	**Long Name**	**Windows 3.1 Client**	**Windows 95 and Windows NT Clients**	**OS/2 Client**	**Windows NT Server**
cmaccp	Accept_Conversation	x	x	x	x
cmacci	Accept_Incoming	x	x	x	x
cmallc	Allocate	x	x	x	x
cmcanc	Cancel_Conversation	x	x	x	x
cmcfm	Confirm	x	x	x	x
cmcfmd	Confirmed	x	x	x	x
cmcnvi	Convert_Incoming	x	x	x	x
cmcnvo	Convert_Outgoing	x	x	x	x
cmdeal	Deallocate	x	x	x	x
xcmdsi	Delete_CPIC_Side_Information	-	-	-	x
cmectx	Extract_Conversation_Context	-	x	x	x
xcecst	Extract_Conversation_Security_Type	x	x	x	x
cmecst	Extract_Conversation_Security_Type	x	x	x	x
cmecs	Extract_Conversation_State	x	x	x	x
cmect	Extract_Conversation_Type	x	x	x	x
xcmesi	Extract_CPIC_Side_Information	x	x	x	x
cmembs	Extract_Maximum_Buffer_Size	x	x	x	x
cmemn	Extract_Mode_Name	x	x	x	x
cmepln	Extract_Partner_LU_Name	x	x	x	x
cmesi	Extract_Secondary_Information	-	x	x	x
cmesui	Extract_Security_User_ID	x	x	x	x
cmecsu	Extract_Security_User_ID	x	x	x	x
xcecsu	Extract_Security_User_ID	x	x	x	x
cmesrm	Extract_Send_Receive_Mode	-	-	-	x
cmesl	Extract_Sync_Level	x	x	x	x
xceti	Extract_TP_ID	-	x	x	x
cmetpn	Extract_TP_Name	x	x	x	x
cmflus	Flush	x	x	x	x
cminit	Initialize_Conversation	x	x	x	x
xcinct	Initialize_Conversation_For_TP	-	x	x	x
cminic	Initialize_For_Incoming	x	x	x	x
cmptr	Prepare_To_Receive	x	x	x	x
cmrcv	Receive	x	x	x	x
cmrcvx	Receive_Expedited	-	-	-	x
cmrltp	Release_Local_TP_Name	x	x	x	x
cmrts	Request_To_Send	x	x	x	x

Table 11. Communications Server Client Support of CPI-C Functions (continued)

Function	**Long Name**	**Windows 3.1 Client**	**Windows 95 and Windows NT Clients**	**OS/2 Client**	**Windows NT Server**
cmsend	Send_Data	x	x	x	x
cmsndx	Send_Expedited	-	-	-	x
cmserr	Send_Error	x	x	x	x
cmscsp	Set_Conversation_Security_Password	x	x	x	x
xcscsp	Set_Conversation_Security_Password	x	x	x	x
cmscst	Set_Conversation_Security_Type	x	x	x	x
xcscst	Set_Conversation_Security_Type	x	x	x	x
cmscsu	Set_Conversation_Security_User_ID	x	x	x	x
xcscsu	Set_Conversation_Security_User_ID	x	x	x	x
cmsct	Set_Conversation_Type	x	x	x	x
xcmssi	Set_CPIC_Side_Information	-	-	-	x
cmsdt	Set_Deallocate_Type	x	x	x	x
cmsed	Set_Error_Direction	x	x	x	x
cmsf	Set_Fill	x	x	x	x
cmsld	Set_Log_Data	x	x	x	x
cmsmn	Set_Mode_Name	x	x	x	x
cmspln	Set_Partner_LU_Name	x	x	x	x
cmsptr	Set_Prepare_To_Receive_Type	x	x	x	x
cmspm	Set_Processing_Mode	x	x	x	x
cmsqcf	Set_Queue_Callback_Function	-	x	x	x
cmsqpm	Set-Queue_Processing_Mode	-	x	x	x
cmsrt	Set_Receive_Type	x	x	x	x
cmsrc	Set_Return_Control	x	x	x	x
cmssrm	Set_Send_Receive_Mode	-	-	-	x
cmsst	Set_Send_Type	x	x	x	x
cmssl	Set_Sync_Level	x	x	x	x
cmstpn	Set_TP_Name	x	x	x	x
cmsltp	Specify_Local_TP_Name	x	x	x	x
xchwnd*	Specify_Windows_Handle	x	x	-	x
xcstp	Start_TP	-	x	x	x
cmtrts	Test_Request_To_Send_Received	x	x	x	x
cmwcmp	Wait_For_Completion	-	x	x	x
cmwait	Wait_For_Conversation	x	x	x	x
xcendt	End_TP	-	x	x	x
WinCPICCleanup*		x	x	-	x
WinCPICIsBlocking*		x	-	-	-
WinCPICSetBlockingHook*		x	-	-	-
WinCPICStartup*		x	x	-	x
WinCPICUnhookBlockingHook*		x	-	-	-

*** indicates:** WOSA function for Microsoft Windows
x indicates: Supported function
- indicates: Unsupported function

Specifying Service TP Names

This function is only supported for clients.

You must use special conventions when specifying a service transaction program name with the CMSTPN and CMSLTP functions. Usually, you specify standard TPs with the CPI-C functions. Service transaction programs are specialized transaction programs that provide common network and system services to other programs or users. Examples of service transaction programs include scheduler programs, directory services, and spoolers.

The conventions for specifying a service transaction program name with the CMSTPN and CMSL transaction program functions are

- Specify the name with from two to five bytes of ASCII characters.
- Specify the first byte of the name, for example, 0x23, with two bytes of ASCII characters.
  - Split the first byte of the name into two nibbles, for example, 2 and 3, and specify them in the low- order nibble of each ASCII byte.
  - Set the high-order nibble of each ASCII byte to 1, which indicates that you are specifying a service TP name. Continuing with the example, the first two bytes specified are 0x12 and 0x13.
- Specify the remaining zero to three bytes of the name as ASCII characters. For example, 007.

Therefore, specify a service transaction program name of 0x23 007, as 0x12 0x13 007.

Chapter 7. APPC Entry Points

The following sections describe the procedure entry points for APPC.

APPC

You can use this as a synchronous entry point for all of the APPC verbs. Alternatively, you can use this entry point to issue nonblocking verbs by putting an event handle in the secondary return code field and setting the queue-level nonblocking flag in the **opext** field (AP_NON_BLOCKING).

Syntax

```
void WINAPI APPC(long)
```

Input is a pointer to a verb control block.

Returned Values

Examine the primary return code and secondary return code for any errors.

Usage Notes

See also: “WinAsyncAPPCEx()” on page 55.

This is the only entry point supported for OS/2 programs.

WinAsyncAPPC()

This is an asynchronous entry point for all of the APPC verbs. An application uses this entry point if it chooses to be notified of completion through a Windows message. Communications Server provides this entry point for compatibility with existing applications.

Syntax

```
HANDLE WINAPI WinAsyncAPPC(HWND hWnd,
                           long vcb)
```

Parameter	**Description**
hwnd	Window handle to receive completion message.
vcb	Pointer to verb control block.

Returned Values

The return value specifies whether the asynchronous request completed successfully. If the request was successful, the actual return value is a handle. If the function was not successful, Communications Server returns a `0`.

Usage Notes

APPC verbs that can block are as follows:

- **[MC_]ALLOCATE**
- **[MC_]CONFIRM**
- **[MC_]CONFIRMED**
- **[MC_]DEALLOCATE**
- **[MC_]FLUSH**
- **[MC_]PREPARE_TO_RECEIVE**
- **RECEIVE_ALLOCATE**
- **[MC_]RECEIVE_AND_WAIT**
- **[MC_]REQUEST_TO_SEND**
- **[MC_]SEND_CONVERSATION**
- **[MC_]SEND DATA**
- **[MC_]SEND_ERROR**
- **TP_ENDED**
- **TP_STARTED**

The WinAsyncAPPC entry point permits the verb to be canceled, but does not support queue-level nonblocking. The APPC entry point supports queue-level nonblocking, but does not permit the application to cancel the verb.

This entry point does not support queue-level nonblocking. If the queue-level nonblocking flag AP_NON_BLOCKING is specified on the asynchronous interface, Communications Server ignores it. When using the asynchronous entry point, an application can have only one outstanding function in progress on a conversation at a time. An attempt to initiate a second function results in the error code AP_CONV_BUSY. If an application needs to be notified of asynchronous completion through an event handle, it can use either the **WinAsyncAPPCEx** or **APPC** entry point. The exceptions to the previous paragraph are **RECEIVE_AND_POST** and **RECEIVE_AND_WAIT**. To enable full use to be made of the asynchronous support, Communications Server alters asynchronously issued **RECEIVE_AND_WAIT** verbs to act like the **RECEIVE_AND_POST** verb. Specifically, while an asynchronous **RECEIVE_AND_POST** or **RECEIVE_AND_WAIT** is outstanding, an application can issue the following verbs on the same conversation:

- **REQUEST_TO_SEND**
- **GET_TYPE**
- **GET_ATTRIBUTES**
- **TEST_RTS**
- **DEALLOCATE** (AP_ABEND_PROG, AP_ABEND_SVC, or AP_ABEND_TIMER)
- **SEND_ERROR**
- **TP_ENDED**

This enables an application, such as a server, to use an asynchronous **RECEIVE_AND_WAIT** to receive data. While the **RECEIVE_AND_POST** or **RECEIVE_AND_WAIT** is outstanding, the application can still use **SEND_ERROR** and **REQUEST_TO_SEND**.

When the asynchronous operation is complete, the application’s window *hWnd* receives the message returned by **RegisterWindowMessage** with “**WinAsyncAPPC**” as the input string. The *wParam* argument contains the asynchronous task handle returned by the original function call. The *lParam* argument contains the original VCB pointer and can be used to determine the final return code.

WinAPPCCancelAsyncRequest permits an application to cancel any asynchronous APPC action, but terminates the related conversation or transaction program as appropriate. Any outstanding operations return with AP_CANCELED as the return code.

If the function returns successfully, Communications Server posts a **WinAsyncAPPC()** message to the application when the operation completes or the conversation is canceled.

See also:

“WinAsyncAPPCEx()” on page 55

“WinAPPCCancelAsyncRequest()” on page 57

WinAsyncAPPCEx()

This is an asynchronous entry point for all of the APPC verbs. Use this call to enable multiple sessions to be handled on the same thread.

Use this entry point if you want the application to be notified of completion through an event and your application requires the ability to cancel outstanding verbs; otherwise, use the APPC queue-level nonblocking entry point.

Syntax

```
HANDLE WINAPI WinAsyncAPPCEx(HANDLE handle,
                             long vcb);
```

Parameter	**Description**
handle	Handle of the event that the application will wait on.
vcb	Pointer to verb control block.

Returned Values

The return value specifies whether the asynchronous request was successful. If the function was successful, the actual return value is a handle. If the function was not successful, Communications Server returns a `0`.

Usage Notes

This verb is intended for use with **WaitForMultipleObjects** in the Win32** API.

APPC verbs that can block are as follows:

- **[MC_]ALLOCATE**
- **[MC_]CONFIRM**
- **[MC_]CONFIRMED**
- **[MC_]DEALLOCATE**
- **[MC_]FLUSH**
- **[MC_]PREPARE_TO_RECEIVE**
- **RECEIVE_ALLOCATE**
- **[MC_]RECEIVE_AND_WAIT**
- **[MC_]REQUEST_TO_SEND**
- **[MC_]SEND_CONVERSATION**
- **[MC_]SEND_DATA**
- **[MC_]SEND_ERROR**
- **TP_ENDED**
- **TP_STARTED**

This entry point does not support queue-level nonblocking. If the queue-level nonblocking flag AP_NON_BLOCKING is specified on the asynchronous interface, Communications Server ignores it. When using the asynchronous entry point, an application can have only one outstanding function in progress on a conversation at a time. An attempt to initiate a second function results in the error code AP_CONV_BUSY.

The **WinAsyncAPPCEx** entry point permits the verb to be canceled, but does not support queue-level nonblocking. The **APPC** entry point supports queue-level nonblocking, but does not permit the application to cancel the verb. The exceptions to the previous paragraph are **RECEIVE_AND_POST** and **RECEIVE_AND_WAIT**. To enable full use to be made of the asynchronous support, Communications Server alters asynchronously issued **RECEIVE_AND_WAIT** verbs to act like the **RECEIVE_AND_POST** verb. Specifically, while an asynchronous **RECEIVE_AND_POST** or **RECEIVE_AND_WAIT** is outstanding, an application can issue the following verbs on the same conversation:

- **REQUEST_TO_SEND**
- **GET_TYPE**
- **GET_ATTRIBUTES**
- **TEST_RTS**
- **DEALLOCATE** (AP_ABEND_PROG, AP_ABEND_SVC, or AP_ABEND_TIMER)
- **SEND_ERROR**
- **TP_ENDED**

This enables an application, and in particular, a server application, to use an asynchronous **RECEIVE_AND_WAIT** to receive data. While the **RECEIVE_AND_POST** or **RECEIVE_AND_WAIT** is outstanding, the application can still use **SEND_ERROR** and **REQUEST_TO_SEND**.

When the asynchronous operation is complete, Communications Server notifies the application by the signaling of the event. When the application receives the signal, it examines the primary return code and secondary return code for any error conditions.

See also:

“WinAsyncAPPC()” on page 53

“WinAPPCCancelAsyncRequest()” on page 57

“APPC” on page 52

WinAPPCCancelAsyncRequest()

This function cancels an outstanding **WinAsyncAPPC**-based request.

Syntax

```
int WINAPI WinAPPCCancelAsyncRequest(HANDLE handle);
```

Parameter
  Description

handle
  Supplied parameter; specifies the handle of the request to be canceled.

Returned Values

The return value specifies whether the asynchronous request was canceled. If the value is 0, Communications Server canceled the request. Otherwise, the value is one of the following error codes:

WAPPCINVALID
  The specified asynchronous task ID was not valid.

WAPPCALREADY
  The asynchronous routine to be canceled has already completed.

Usage Notes

An application program can cancel an asynchronous task that was previously issued with one of the **WinAsyncAPPC** functions prior to completion, by issuing the **WinAPPCCancelAsyncRequest()** call, and specifying the asynchronous event as returned by the initial function in the handle.

If the outstanding verb relates to a conversation (for example, **SEND_DATA** or **RECEIVE_AND_WAIT**), Communications Server purges the verb and deactivates the session. If the verb relates to a transaction program (for example, **RECEIVE_ALLOCATE** or **TP_STARTED**), Communications Server ends the transaction program. In both cases, although Communications Server deactivates conversations and sessions as cleanly as possible, it does not flush send buffers or waiting-for-confirmations or other pending actions. This call is synchronous. After the previously described processing is complete, Communications Server posts a completion message for the canceled verb.

If an attempt to cancel an existing asynchronous **WinAsyncAPPC** routine fails with an error code of WAPPCALREADY, the original routine has already completed. Either the application has dealt with the resulting notification, or the application has not dealt with the completion notification. It is not possible to cancel an asynchronous verb issued through the APPC queue-level nonblocking entry point.

See also: “WinAsyncAPPC()” on page 53.

WinAPPCCancelBlockingCall()

This function cancels any outstanding blocking operation for its thread. If Communications Server cancels an outstanding blocked call, it generates an error code of AP_CANCELLED. Use this call only from within a blocking hook function. Communications Server provides this function for backward compatibility with existing applications.

Syntax

```
Int WINAPI WINAPPCCancelBlockingCall(void);
```

Returned Values

The return value specifies whether the cancellation request was successful. If the value is 0, Communications Server canceled the request. Otherwise, the value is the following error code:

WAPPCINVALID
There is no outstanding blocking call.

Usage Notes

If the outstanding verb relates to a conversation (for example, **SEND_DATA** or **RECEIVE_AND_WAIT**), Communications Server purges the verb and deactivates the session. If the verb relates to a transaction program (for example, **RECEIVE_ALLOCATE** or **TP_STARTED**), Communications Server ends the transaction program. In both cases, although Communications Server deactivates conversations and sessions as cleanly as possible, it does not flush send buffers or waiting-for-confirmations or other pending actions. This call is synchronous. After the previously described processing is complete, the function is finished.

A multithreaded application can have multiple blocking operations outstanding, but only one per thread. To distinguish between multiple outstanding calls, **WinAPPCCancelBlockingCall()** cancels the outstanding operation on the current, or calling, application thread if one exists; otherwise, it fails. APPC suspends the calling application thread while an operation is outstanding. As a result, the thread on which the blocking operation was initiated does not regain control (and therefore, is not be able to issue a call to **WinAPPCCancelBlockingCall()**) unless the application has previously registered a blocking hook for the thread by using **WinAPPCSetBlockingHook**.

This is not supported for Windows 95 and Windows NT SNA API clients.

WinAPPCCleanup()

This function terminates and deregisters an application from the APPC API.

Syntax

```
BOOL WINAPI WinAPPCCleanup(void);
```

Returned Values

The return value specifies whether the deregistration was successful. If the value is not 0, Communications Server has successfully deregistered the application. If Communications Server has not deregistered the application, it returns a value of `0`.

Usage Notes

Use **WinAPPCCleanup()** to deregister a Communications Server application from the APPC API.

Communications Server terminates conversations that are still active and ends transaction programs. This function is equivalent to issuing **TP_ENDED(HARD)** on all transaction programs owned by the application.

See also: “WinAPPCStartup()” on page 61.

WinAPPCIsBlocking()

This function determines if a thread is executing while waiting for a previous blocking call to finish. Communications Server provides this function for backward compatibility with existing applications.

Syntax

```
BOOL WlNAPI WinAPPCIsBlocking(void);
```

The return value specifies the outcome of the function. If the value is not 0, an outstanding blocking call is awaiting completion. A value of `0` means there is no outstanding blocking call.

Usage Notes

The Communications Server DLL prohibits more than one blocking call per thread; it returns AP_THREAD_BLOCKING if this occurs. A thread that is executing a blocking call is not reentered unless a blocking hook function has been set. In this case, **WinAPPCIsBlocking** returns true only from within a blocking hook function.

See also:

“WinAPPCCancelBlockingCall()” on page 58

“WinAPPCSetBlockingHook()” on page 62

“WinAPPCUnhookBlockingHook()” on page 63

This is not supported for Windows 95 and Windows NT SNA API clients.

WinAPPCStartup()

This function enables an application to specify the version of Communications Server required and to retrieve version information from Communications Server. This call is not required.

Syntax

```
int WINAPI WinAPPCStartup(WORD wVersionRequired,
                          LPWAPPCDATA wappcdata);
```

Parameter
Description

wVersionRequired
Specifies the version of Communications Server support required. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.

wappcdata
Returns the version of APPC API and a description of API implementation.

Returned Values

The return value specifies whether Communications Server successfully registered the application and whether it can support the specified version number. If the value is `0`, Communications Server supports the specified version and it successfully registers the application. Otherwise, one of the following values is returned:

WAPPCSYSNOTREADY
The underlying network subsystem is not ready for network communication.

WAPPCVERNOTSUPPORTED
This particular Communications Server implementation does not support the version of Communications Server support requested.

WAPPCINVALID
Communications Server could not determine the specified version.

Usage Notes

WinAPPCStartup() is intended to help with compatibility of future releases of the API. This DLL supports Version 1.0.

See also: “WinAPPCCleanup()” on page 59.

WinAPPCSetBlockingHook()

This function enables an APPC implementation of the APPC API to block APPC function calls.

Communications Server provides this function for compatibility with existing applications.

Syntax

```
FARPROC WINAPI WinAPPCSetBlockingHook(FARPROC IpBlockFunc);
```

Parameter
    Description

IpBlockFunc
    Specifies the procedure instance address of the blocking function to be installed.

Returned Values

The return value points to the procedure instance of the previously installed blocking function. The application or library that calls the **SetBlockingHook** function should save this return value so that it can be restored if needed. (If nesting is not important, the application can simply discard the value returned by **WinAPPCSetBlockingHook()** and eventually use **WinAPPCUnhookBlockingHook** to restore the default mechanism.)

Usage Notes

A blocking function returns FALSE if it receives a WM_QUIT message so that Communications Server can return control to the application to process the message and terminate gracefully. Otherwise, the function returns TRUE.

No default blocking hook is implemented. If an application does not set a blocking hook, the application thread waits indefinitely for the blocking call to return.

If the blocking hook function does not return TRUE, Communications Server returns the blocking verb to the application with the primary return code AP_CANCELLED.

This function is implemented by thread. It provides for a particular thread to replace the blocking mechanism without affecting other threads.

See also:

"WinAPPCCancelBlockingCall()" on page 58

"WinAPPCIsBlocking()" on page 60

"WinAPPCUnhookBlockingHook()" on page 63

This is not supported for Windows 95 and Windows NT SNA API clients.

WinAPPCUnhookBlockingHook()

This function removes any previous blocking hook that has been installed.

Communications Server provides this function for backward compatibility with existing applications.

Syntax

```
BOOL WINAPI WinAPPCUnhookBlockingHook (void);
```

Returned Values

The return value specifies the outcome of the function. It is not 0 if Communications Server successfully reinstalled the default mechanism. The value is 0 if Communications Server did not reinstall the default mechanism.

Usage Notes

After the function is called, this application thread waits indefinitely for all future blocking calls to complete.

See also: “WinAPPCSetBlockingHook()” on page 62.

This is not supported for Windows 95 and Windows NT SNA API clients.

GetAppcConfig()

This function is not implemented. However, an entry point is provided for backward compatibility. If a valid set of parameters is specified, Communications Server returns APPC_CFG_SUCESS_NO_DEFAULT_REMOTE and puts a NULL terminator in the first byte of the RemLu buffer.

In many cases this call is not necessary because Communications Server is an APPN capable node. The partner LU name can be specified on ALLOCATE and a search for the LU will be initiated. However, applications can use the Node Operator Facility (NOF) interface to retrieve this information. For information on using the NOF interface, refer to *Communications Server: System Management Programming*.

GetAppcReturnCode()

This function converts the primary and secondary return codes in the VCB to a printable string. It provides a standard set of error strings for use by APPC applications.

Syntax

```
int WINAPI GetAppcReturnCode (struct appc_hdr *vcb,
                              UINT buffer_length,
                              unsigned char *buffer_addr);
```

Parameter
Description

vcb Supplied parameter; specifies the address of the verb control block.

buffer_length
Supplied parameter; specifies the length of the buffer pointed to by **buffer_addr**. The recommended length is 256.

buffer_addr
Supplied/returned parameter; specifies the address of the buffer that will hold the formatted, null-terminated string. Length of the string in the specified buffer.

Returned Values

0x20000001
The parameters are not valid; the function could not read from the specified verb control block or could not write to the specified buffer.

0x20000002
The specified buffer is too small.

Usage Notes

The descriptive error string returned in **buffer_addr** does not terminate with a new line character (**\n**).

Chapter 8. APPC Verbs

This chapter documents the syntax of each verb passed across the APPC API, and describes the parameters passed in and returned for each verb.

This chapter also provides reference information for the APPC basic and mapped conversation verbs that are provided for APPC duplex and half-duplex conversations. As you read through this chapter, you will discover that the basic and mapped verbs are very similar and that is why they have been combined into one chapter. However, there are some differences. Those differences are denoted as follows:

- When the information applies only to the basic verb, this symbol
- When the information applies only to the mapped verb, this symbol

One more clarification in the information! When the conversation verb can be basic or mapped, it is denoted as follows:

[MC_]VERBNAME

Verb Control Blocks

This section contains a general description of the fields and indications for each verb.

Common Fields

Each VCB has a number of common fields, as follows:

opcode
Verb operation code: an identifying field containing the name of the verb.

format
Identifies the format of the VCB. The value that this field must be set to in order to specify the current version of the VCB is documented individually under each verb.

primary_rc
Primary return code. Possible values for each verb are listed in the following sections.

secondary_rc
Secondary return code. This supplements the information provided by the primary return code. Possible values for each verb are listed in the following sections. Some VCBs also contain the following fields.

opext Verb extension code. This supplements the information provided by the verb operation code. If the verb signal is to be processed in nonblocking mode, the flag AP_NON_BLOCKING must be set. In the signals described below these common fields are included, but not explained individually.

TP Identifiers
An 8-byte transaction program identifier is assigned to each active transaction program. This identifier is assigned by Communications Server.

The transaction program identifier is used to route **TP_ENDED** verbs and as a correlator on conversation verbs.

The verb control blocks for each signal are described in the following section.

APPC API Support

Verbs Supported

Communications Server supports the following verbs at the APPC API.

Type Independent Verbs

GET_TP_PROPERTIES
GET_TYPE
RECEIVE_ALLOCATE
TP_ENDED
TP_STARTED

Conversation Verbs

[MC_]ALLOCATE
[MC_]CONFIRM
[MC_]CONFIRMED
[MC_]DEALLOCATE
[MC_]FLUSH
[MC_]GET_ATTRIBUTES
[MC_]PREPARE_TO_RECEIVE
[MC_]RECEIVE_AND_POST
[MC_]RECEIVE_AND_WAIT
[MC_]RECEIVE_EXPEDITED_DATA
[MC_]RECEIVE_IMMEDIATE
[MC_]REQUEST_TO_SEND
[MC_]SEND_CONVERSATION
[MC_]SEND_DATA
[MC_]SEND_ERROR
[MC_]SEND_EXPEDITED_DATA
[MC_]TEST_RTS
[MC_]TEST_RTS_AND_POST

GET_TP_PROPERTIES

GET_TP_PROPERTIES returns attributes associated with the transaction program.

VCB Structure

```
typedef struct get_tp_properties
{
 unsigned short   opcode;             /* verb operation code           */
 unsigned char    opext;              /* verb extension code           */
 unsigned char    format;             /* format                        */
 unsigned char    reserv2[2]          /* verb format                   */
 unsigned short   primary_rc;         /* primary return code           */
 unsigned long    secondary_rc;       /* secondary return code         */
 unsigned char    tp_id[8];           /* TP identifier                 */
 unsigned char    tp_name[64];        /* TP name                       */
 unsigned char    lu_alias[8];        /* LU alias                      */
 luw_id_overlay   luw_id;             /* LUW identifier      */
 unsigned char    fqlu_name[17];      /* fully qualified LU name       */
 unsigned char    reserv3[10];        /* reserved                      */
 unsigned char    user_id[10];        /* user id                       */
} GET_TP_PROPERTIES;
typedef struct luw_id_overlay
{
 unsigned char    fqlu_name_len;      /* fully qualified LU name length */
 unsigned char    fqlu_name[17];      /* fully qualified LU name       */
 unsigned char    instance[6];        /* instance number               */
 unsigned char    sequence[2];        /* sequence number               */
} LUW_ID_OVERLAY;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode
AP_GET_TP_PROPERTIES

tp_id Identifier for the local transaction program. The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

opext AP_BASIC_CONVERSATION

format
Identifies the format of the VCB. Set this field to one to specify the version of the VCB listed above.

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameters:

primary_rc
AP_OK

tp_name
Name of the local transaction program, that is, the transaction program issuing this verb. Communications Server does not check the character set of this field.

lu_alias
Alias of the local LU associated with the transaction program. This is an 8-byte string in a locally displayable character set. All 8 bytes are significant and must be set.

The **luw_id** field is a logical unit of work identifier associated with unprotected conversations (conversations with **sync_level** of AP_NONE or AP_CONFIRM_SYNC_LEVEL). The **luw_id_overlay** contains the following parameters:

luw_id_overlay.fq_lu_name_len
Length of fully qualified LU name associated with logical unit of work.

luw_id_overlay.fq_lu_name
Fully qualified LU name associated with Logical Unit of Work. This name is up to 17 bytes long and is right-padded with EBCDIC blanks. It is composed of two type-A EBCDIC character strings concatenated by an EBCDIC dot. (Each name can have a maximum length of 8 bytes with no embedded blanks. If the network ID is not present, then omit the dot.) If the name length is less than 17 bytes, **instance** and **sequence** immediately follow the name (note that this means the fields of the LUW_ID_OVERLAY structure cannot be used to access either **instance** or **sequence**).

luw_id_overlay.instance
Logical unit of work instance number. This is a binary string of length 6 bytes.

luw_id_overlay.sequence
Logical unit of work sequence number. This is a binary string of length 2 bytes.

If **luw_id_overlay.fq_lu_name_len** is less than 17, **luw_id_overlay** is right padded with EDCDIC blanks (after **instance** and **sequence**).

luw_id_overlay.fq_lu_name
Fully qualified name of the local LU associated with the transaction program. This name is 17 bytes long and is right-padded with EBCDIC blanks. It is composed of two type-A EBCDIC character strings concatenated by an EBCDIC dot. (Each name can have a maximum length of 8 bytes with no embedded blanks. If the network ID is not present, then omit the dot.)

user_id
User ID of the initiator of the transaction. This is a 10-byte type-AE EBCDIC character string, padded to the right with EBCDIC spaces.

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_TP_ID

The conditions generating the following possible primary return codes (**primary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_TP_BUSY

AP_UNEXPECTED_SYSTEM_ERROR

GET_TYPE

The **GET_TYPE** verb returns the conversation type (basic or mapped) of a particular conversation.

VCB Structure

```
typedef struct get_type
{
 unsigned short    opcode;             /* verb operation code      */
 unsigned char     opext;              /* verb extension code      */
 unsigned char     format;             /* format                   */
 unsigned short    primary_rc;         /* primary return code      */
 unsigned long     secondary_rc;       /* secondary return code    */
 unsigned char     tp_id[8];           /* TP identifier            */
 unsigned long     conv_id;            /* conversation identifier  */
 unsigned char     conv_type;          /* conversation type        */
 unsigned char     conv_style;         /* conversation style       */
} GET_TYPE;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode
AP_GET_TYPE

opext AP_BASIC_CONVERSATION

format
Identifies the format of the VCB. Set this field to one to specify the version of the VCB listed above.

tp_id Identifier for the local transaction program. The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

conv_id
Conversation identifier. The value of this parameter was returned by the **ALLOCATE** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameters:

primary_rc
AP_OK

conv_type
Conversation type of the conversation identified by **conv_id**.

AP_BASIC_CONVERSATION
AP_MAPPED_CONVERSATION

conv_style
Conversation style of the conversation identified by **conv_id**. This field requires the format 1 version of the VCB. See “Full-Duplex VCBs” on page 39 for more details on accessing format 1 VCBs.

AP_HALF_DUPLEX
AP_FULL_DUPLEX

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_TP_ID

AP_BAD_CONV_ID

The conditions generating the following possible primary return codes (**primary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_TP_BUSY

AP_UNEXPECTED_SYSTEM_ERROR

RECEIVE_ALLOCATE

The **RECEIVE_ALLOCATE** verb requests information needed to establish a new transaction program for a conversation with a partner transaction program that has issued an **ALLOCATE** or **MC_ALLOCATE** verb.

VCB Structure

```
typedef struct receive_allocate
{
 unsigned shor    opcode;                  /* verb operation code            */
 unsigned char    opext;                   /* verb extension code            */
 unsigned char    format;                  /* format                         */
 unsigned shor    primary_rc;              /* primary return code            */
 unsigned long    secondary_rc;            /* secondary return code          */
 unsigned char    tp_name[64];             /* TP name                        */
 unsigned char    tp_id[8];                /* TP identifier                  */
 unsigned long    conv_id;                 /* conversation identifier        */
 unsigned char    sync_level;              /* sync Level                     */
 unsigned char    conv_type;               /* conversation type              */
 unsigned char    user_id[10];             /* user ID                        */
 unsigned char    lu_alias[8];             /* LU alias                       */
 unsigned char    plu_alias[8];            /* partner LU alias               */
 unsigned char    mode_name[8];            /* mode name                      */
 unsigned char    reserv3[2];              /* reserved                       */
 unsigned long    conv_group_id;           /* conversation group ID          */
 unsigned char    fqplu_name[17];          /* fully qualified partner LU name */
 unsigned char    pip_incoming;            /* received PIP data              */
 unsigned char    conversation_style;      /* conversation style             */
 unsigned char    reserv4[3];              /* reserved                       */
 unsigned char    password[10];            /* security password              */
 unsigned char    reserv5[2];              /* reserved                       */
 unsigned char    dload_id[8];             /* user ID                        */
} RECEIVE_ALLOCATE;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode
AP_RECEIVE_ALLOCATE

opext AP_BASIC_CONVERSATION

format
Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

tp_name
Name of the transaction program. Communications Server does not check the character set of this field.

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameters:

primary_rc
AP_OK

tp_id Identifier for the local transaction program. This value is assigned by

Communications Server to the transaction program. The transaction program passes this identifier to Communications Server on all subsequent APPC verbs.

conv_id
Conversation identifier. This value identifies the conversation established between the two transaction programs.

sync_level
Synchronization level of the conversation.

AP_CONFIRM_SYNC_LEVEL

AP_NONE

conv_type
Conversation type of the conversation identified by **conv_id**.

AP_BASIC_CONVERSATION
AP_MAPPED_CONVERSATION

user_id
User ID supplied by the partner transaction program. This is a 10-byte type-AE EBCDIC character string, padded to the right with EBCDIC spaces.

lu_alias
Alias by which the local LU is known. This is an 8-byte string in a locally displayable character set. All 8 bytes are significant and must be set.

plu_alias
Alias by which the partner LU is known to the local transaction program. This is an 8-byte string in a locally displayable character set. All 8 bytes are significant and must be set.

mode_name
Name of a set of networking characteristics defined during configuration. This is an 8-byte alphanumeric type-A EBCDIC string (starting with a letter), padded to the right with EBCDIC spaces.

conv_group_id
Conversation group identifier for the session being used by this conversation.

fqplu_name
Fully qualified LU name for the partner LU. This name is 17 bytes long and is right-padded with EBCDIC blanks. It is composed of two type-A EBCDIC character strings concatenated by an EBCDIC dot. (Each name can have a maximum length of 8 bytes with no embedded blanks. If the network ID is not present, omit the dot.)

pip_incoming
Specifies whether the partner transaction program-supplied Program Initialization Parameters (PIP) on the [MC_]ALLOCATE request. Set to AP_YES or AP_NO. If AP_YES, the PIP data will be received on the first **[MC_]RECEIVE_*** verb issued on this conversation.

conv_style
Conversation style of the conversation identified by **conv_id**.

AP_HALF_DUPLEX
AP_FULL_DUPLEX

password
: Password associated with **user_id**. This is a 10-byte type-AE EBCDIC character string, padded to the right with EBCDIC spaces. This is required if Security=Program (AP_PGM or AP_PGM_STRONG); otherwise, it is optional.

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
: AP_PARAMETER_CHECK

secondary_rc
: AP_UNDEFINED_TP_NAME

The conditions generating the following possible primary return codes (**primary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_UNEXPECTED_SYSTEM_ERROR

TP_ENDED

The **TP_ENDED** verb notifies Communications Server that a specified transaction program has ended.

VCB Structure

```
typedef struct tp_ended
{
 unsigned short    opcode;             /* verb operation code   */
 unsigned char     opext;              /* verb extension code   */
 unsigned char     format;             /* format                */
 unsigned short    primary_rc;         /* primary return code   */
 unsigned long     secondary_rc;       /* secondary return code */
 unsigned char     tp_id[8];           /* TP identifier         */
 unsigned char     type;               /* type of TP ended      */
} TP_ENDED;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode
AP_TP_ENDED

opext AP_BASIC_CONVERSATION

format
Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

tp_id Identifier for the local transaction program. The value of this parameter was returned by the **TP_STARTED** verb for an invoking transaction program, or by the **RECEIVE_ALLOCATE** verb for an invoked transaction program.

type Type of **TP_ENDED**.

AP_HARD
AP_SOFT
AP_ABEND
AP_CANCEL

If type is AP_ABEND, Communications Server does not reply to the **TP_ENDED** signal.

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameter:

primary_rc
AP_OK

Returned Parameters

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_TP_ID

AP_BAD_TYPE

The conditions generating the following possible primary return codes (**primary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_TP_BUSY

AP_UNEXPECTED_SYSTEM_ERROR

TP_STARTED

The **TP_STARTED** verb notifies Communications Server that a program has requested resources for a transaction program initiated as a result of a local command, rather than an incoming allocation request.

VCB Structure

```
typedef struct tp_started
{
 unsigned short   opcode;              /* verb operation            */
 unsigned char    opext;               /* verb extension            */
 unsigned char    format;              /* format                    */
 unsigned short   primary_rc;          /* primary return code       */
 unsigned long    secondary_rc;        /* secondary return code     */
 unsigned char    lu_alias[8];         /* LU alias                  */
 unsigned char    tp_id[8];            /* TP identifier             */
 unsigned char    tp_name[64];         /* TP name                   */
} TP_STARTED;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode
AP_TP_STARTED

opext AP_BASIC_CONVERSATION

format
Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

lu_alias
Alias by which the local LU is known. If it is set to zero, Communications Server uses the control point LU. This is an 8-byte string in a locally displayable character set. All 8 bytes are significant and must be set. A blank **lu_alias** field is accepted. The control point LU is used in this case.

tp_name
Name of the transaction program. Communications Server does not check the character set of this field.

Returned Parameters

If the verb was executed successfully, Communications Server returns the following parameters:

primary_rc
AP_OK

tp_id Identifier for the local transaction program. This value is assigned by Communications Server to the transaction program. The transaction program passes this identifier to Communications Server on all subsequent APPC verbs.

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_INVALID_LU_NAME

AP_INVALID_ENABLE_POOL

The conditions generating the following possible primary return code (**primary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_UNEXPECTED_SYSTEM_ERROR

[MC_]ALLOCATE

The **[MC_]ALLOCATE** verb is issued by the invoking transaction program. This verb allocates a session between the local LU and the partner LU and then (in conjunction with the **RECEIVE_ALLOCATE** verb) establishes a conversation between the invoking transaction program and the invoked transaction program.

The **ALLOCATE** verb can establish either a basic or mapped conversation. Using the **ALLOCATE** verb to establish a mapped conversation enables the transaction program to use basic conversation verbs to communicate with a mapped conversation partner transaction program.

Communications Server generates a conversation identifier (**conv_id**) when this verb executes successfully. This identifier is a parameter that is required for all other APPC conversation verbs.

VCB Structure

```
typedef struct allocate
{
 unsigned short    opcode;                   /* verb operation code          */
 unsigned char     opext;                    /* verb extension code          */
 unsigned char     format;                   /* format                       */
 unsigned short    primary_rc;               /* primary return code          */
 unsigned long     secondary_rc;             /* secondary return code        */
 unsigned char     tp_id[8];                 /* TP identifier                */
 unsigned long     conv_id;                  /* conversation identifier      */
 unsigned char     conv_type;                /* conversation type            */
 unsigned char     sync_level;               /* sync level                   */
 unsigned char     reserv3[2];               /* reserved                     */
 unsigned char     rtn_ctl;                  /* return control               */
 unsigned char     conversation_style;       /* conversation style           */
 unsigned long     conv_group_id;            /* conversation group identifier */
 unsigned long     sense_data;               /* sense data                   */
 unsigned char     plu_alias[8];             /* partner LU alias             */
 unsigned char     mode_name[8];             /* mode name                    */
 unsigned char     tp_name[64];              /* partner TP name              */
 unsigned char     security;                 /* security level               */
 unsigned char     reserv5[11];              /* reserved                     */
 unsigned char     pwd[10];                  /* security password            */
 unsigned char     user_id[10];              /* security user_id             */
 unsigned short    pip_dlen;                 /* PIP data length              */
 unsigned char    *pip_dptr;                 /* pointer to PIP data          */
 unsigned char     reserv5a;                 /* reserved                     */
 unsigned char     fqplu_name[17];           /* fully qualified partner LU   */
                                             /* name                         */
 unsigned char     reserv6[8];               /* reserved                     */
} ALLOCATE;
```

```
typedef struct mc_allocate
{
 unsigned short     opcode;                /* verb operation code          */
 unsigned char      opext;                 /* verb extension code          */
 unsigned char      format;                /* format                       */
 unsigned short     primary_rc;            /* primary return code          */
 unsigned long      secondary_rc;          /* secondary return code        */
 unsigned char      tp_id[8];              /* TP identifier                */
 unsigned long      conv_id;               /* conversation identifier      */
 unsigned char      reserv3;               /* reserved                     */
 unsigned char      sync_level;            /* sync level                   */
 unsigned char      reserv4[2];            /* reserved                     */
 unsigned char      rtn_ctl;               /* return control               */
 unsigned char      conversation_style;    /* conversation style           */
 unsigned long      conv_group_id;         /* conversation group identifier */
 unsigned long       sense_data;            /* sense data                       */
 unsigned char       plu_alias[8];          /* partner LU alias                 */
 unsigned char       mode_name[8];          /* mode name                        */
 unsigned char       tp_name[64];           /* partner TP name                  */
 unsigned char       security;              /* security level                   */
 unsigned char       reserv6[11];           /* reserved                         */
 unsigned char       pwd[10];               /* security password                */
 unsigned char       user_id[10];           /* security user_id                 */
 unsigned short      pip_dlen;              /* PIP data length                  */
 unsigned char      *pip_dptr;              /* pointer to PIP data              */
 unsigned char       reserv6a;              /* reserved                         */
 unsigned char       fqplu_name[17];        /* fully qualified partner LU       */
                                            /* name                             */
 unsigned char       reserv7[8];            /* reserved                         */
} MC_ALLOCATE;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode
AP_B_ALLOCATE

AP_M_ALLOCATE

format
Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION. For nonblocking operation, this flag can be ORed together with AP_NON_BLOCKING.

tp_id Identifier for the local transaction program.

The value of this parameter was returned by the **TP_STARTED** verb for an invoking transaction program, or by the **RECEIVE_ALLOCATE** verb for an invoked transaction program.

conv_type
Type of conversation to allocate.

AP_BASIC_CONVERSATION
AP_MAPPED_CONVERSATION

If the **ALLOCATE** verb establishes a mapped conversation, the local transaction program must issue basic-conversation verbs and provide its own mapping layer to convert data records to logical records and logical records to data records. The partner transaction program can issue basic-conversation verbs and provide the mapping layer, or it can use mapped-conversation verbs (if the implementation of APPC that the partner transaction program is using supports mapped-conversation verbs). For further information, see *IBM Systems Network Architecture: LU 6.2 Reference: Peer Protocols*

sync_level
Synchronization level of the conversation.

AP_CONFIRM_SYNC_LEVEL
AP_NONE

rtn_ctl
Specifies when the local LU acting on a session request from the local transaction program is to return control to the local transaction program.

AP_IMMEDIATE
AP_WHEN_SESSION_ALLOCATED
AP_WHEN_SESSION_FREE
AP_WHEN_CONV_GROUP_ALLOC
AP_WHEN_CONWINNER_ALLOC
AP_WHEN_CONLOSER_ALLOC

conversation_style
Conversation style of the conversation identified by **conv_id**.

duplex_type
Duplex type of the conversation to be allocated.

AP_HALF_DUPLEX
AP_FULL_DUPLEX

This is only supported for applications residing on the server.

conv_group_id
Conversation group identifier for the session to be allocated. This parameter is only supplied if **rtn_ctl** is set to AP_WHEN_CONV_GROUP_ALLOC.

plu_alias
Alias by which the partner LU is known to the local transaction program. This is an 8-byte string in a locally displayable character set. All 8 bytes are significant and must be set. This name must match the name of a partner LU established during configuration. If this field is set to all zeros, Communications Server uses the **fqplu_name** field to specify the required partner LU.

mode_name
Name of a set of networking characteristics usually defined during configuration. This is an 8-byte alphanumeric type-A EBCDIC string (starting with a letter), padded to the right with EBCDIC spaces.

tp_name
Name of the invoked transaction program. Communications Server does not check the character set of this field. The value of **tp_name** specified by the **ALLOCATE** verb in the invoking transaction program must match the value of **tp_name** specified by the **RECEIVE_ALLOCATE** verb in the invoked transaction program.

security
Specifies the information the partner LU requires in order to validate access to the invoked transaction program.

AP_NONE

The invoked transaction program uses no conversation security.

AP_PGM

The invoked transaction program uses conversation security, which requires a user ID and password.

AP_SAME

The invoked transaction program uses conversation security and is configured to accept an already-verified indicator. The user ID will be sent with an already-verified indicator, informing the invoked transaction program that no password is required.

AP_PGM_STRONG

Same as AP_PGM, but the ALLOCATE will only succeed if the session to the partner LU supports password substitution.

Note: If the [MC_]ALLOCATE specifies a security type of AP_SAME but does not specify a user ID and password, the user ID and password specified on a previous SET_TP_PROPERTIES verb (if any) are used. If the [MC_]ALLOCATE does carry a user ID and password, then these are always used in place of any that may have been specified on the SET_TP_PROPERTIES verb.

pwd
Password associated with **user_id**. This is a 10-byte type-AE EBCDIC character string, padded to the right with EBCDIC spaces. This is required if Security=Program (AP_PGM or AP_PGM_STRONG); otherwise, it is optional.

user_id
User ID required to access the partner transaction program. This is a 10-byte type-AE EBCDIC character string, padded to the right with EBCDIC spaces. This is required if Security=Program (AP_PGM or AP_PGM_STRONG); otherwise, it is optional.

pip_dlen
Length of the program initialization parameters (PIP) to be passed to the partner transaction program. Range: 0–32767

pip_dptr
Address of buffer containing PIP data. Use this parameter only if **pip_dlen** is greater than zero.

fqplu_name
Fully qualified LU name for the partner LU. This name is 17 bytes long and is right-padded with EBCDIC blanks. It is composed of two type-A EBCDIC character strings concatenated by an EBCDIC dot. (Each name can have a maximum length of 8 bytes with no embedded blanks. If the network ID is not present, then omit the dot.) This field is only significant if the **plu_alias** field is set to all zeros.

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameters:

primary_rc
AP_OK

conv_id
Conversation identifier. This value identifies the conversation established between the two transaction programs.

conv_group_id
Conversation group identifier of the session allocated to the conversation.

If the verb is nonblocking and has not completed, Communications Server returns the following parameters:

primary_rc
AP_OPERATION_INCOMPLETE

opext AP_OPERATION_INCOMPLETE_FLAG

If the **rtn_ctl** parameter was set to AP_IMMEDIATE, and no session is available immediately, Communications Server returns the following parameter:

primary_rc
AP_UNSUCCESSFUL

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_CONV_TYPE

AP_BAD_DUPLEX_TYPE

AP_BAD_RETURN_CONTROL

AP_BAD_SECURITY

AP_BAD_SYNC_LEVEL

AP_CONFIRM_INVALID_FOR_FDX

AP_NO_USE_OF_SNASVCMG_CPSVCMG

AP_BAD_TP_ID

AP_PIP_LEN_INCORRECT

AP_UNKNOWN_PARTNER_MODE

sense_data
Provides additional information on the reason the **[MC_]ALLOCATE** failed.

The conditions generating the following possible primary return codes (**primary_rc**) and indented secondary return codes (**secondary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_ALLOCATION_ERROR
- AP_ALLOCATION_FAILURE_NO_RETRY
- AP_ALLOCATION_FAILURE_RETRY
- AP_FDX_NOT_SUPPORTED_BY_LU
- AP_SEC_REQUESTED_NOT_SUPPORTED

AP_TP_BUSY
AP_UNSUCCESSFUL
AP_UNEXPECTED_SYSTEM_ERROR
AP_CANCELLED

If the **primary_rc** is set to AP_ALLOCATION_ERROR, the **sense_data** field carries more information on the failure.

[MC_]CONFIRM

The **CONFIRM** verb sends the contents of the local LUs send buffer and a confirmation request to the partner transaction program. In response to the **CONFIRM** verb, the partner transaction program normally issues the **CONFIRMED** verb to confirm that it has received the data without error. (If the partner transaction program encounters an error, it issues the **SEND_ERROR** verb or abnormally deallocates the conversation.)

The transaction program can issue the **CONFIRM** verb only if the conversation's synchronization level, established by the **ALLOCATE** verb, is AP_CONFIRM_SYNC_LEVEL.

VCB Structure

```
typedef struct confirm
{
 unsigned short    opcode;              /* verb operation code      */
 unsigned char     opext;               /* verb extension code      */
 unsigned char     format;              /* format                   */
 unsigned short    primary_rc;          /* primary return code      */
 unsigned long     secondary_rc;        /* secondary return code    */
 unsigned char     tp_id[8];            /* TP identifier            */
 unsigned long     conv_id;             /* conversation identifier  */
 unsigned char     rts_rcvd;            /* request to send received */
#ifdef WINAPPC_FORMAT_1
 unsigned char     expd_data_rcvd;      /* expedited data received  */
#endif
} CONFIRM;
```

```
typedef struct mc_confirm
{
 unsigned short      opcode;               /* verb operation code      */
 unsigned char       opext;                /* verb extension code      */
 unsigned char       format;               /* format                   */
 unsigned short      primary_rc;           /* primary return code      */
 unsigned long       secondary_rc;         /* secondary return code    */
 unsigned char       tp_id[8];             /* TP identifier            */
 unsigned long       conv_id;              /* conversation identifier  */
 unsigned char       rts_rcvd;             /* request to send received */
#ifdef WINAPPC_FORMAT_1
 unsigned char       expd_data_rcvd;       /* expedited data received  */
#endif
} MC_CONFIRM;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode

AP_B_CONFIRM

AP_M_CONFIRM

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION. For nonblocking operation, this flag can be ORed together with AP_NON_BLOCKING.

format
Identifies the format of the VCB. Set this field to one to specify the version of the VCB listed above.

tp_id
Identifier for the local transaction program. The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

conv_id
Conversation identifier. The value of this parameter was returned by the **ALLOCATE** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameters:

primary_rc
AP_OK

rts_rcvd
Request-to-send-received indicator.

AP_YES
AP_NO

expd_data_rcvd
Expedited-data-received indicator. This indication continues to be set to AP_YES until a RECEIVE_EXPEDITED_DATA is issued.

AP_YES
AP_NO

This field requires the format 1 version of the VCB. See "Full-Duplex VCBs" on page 39 for more details on accessing format 1 VCBs.

If the verb is nonblocking and has not completed, Communications Server returns the following parameters;

primary_rc
AP_OPERATION_INCOMPLETE

opext
If the verb is nonblocking and has not completed, Communications Server returns the following parameters:

AP_OPERATION_INCOMPLETE_FLAG

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_CONV_ID

AP_BAD_TP_ID
AP_CONFIRM_INVALID_FOR_FDX
AP_CONFIRM_ON_SYNC_LEVEL_NONE

If the conversation is in the wrong state when the transaction program issues this verb, Communications Server returns the following parameters:

primary_rc
AP_STATE_CHECK

secondary_rc
AP_CONFIRM_BAD_STATE

AP_CONFIRM_NOT_LL_BDY

The conditions generating the following possible primary return codes (**primary_rc**) and indented secondary return codes (**secondary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_ALLOCATION_ERROR

AP_SECURITY_NOT_VALID
AP_TRANS_PGM_NOT_AVAIL_RETRY
AP_TRANS_PGM_NOT_AVAIL_NO_RTRY
AP_TP_NAME_NOT_RECOGNIZED
AP_PIP_NOT_ALLOWED
AP_PIP_NOT_SPECIFIED_CORRECTLY
AP_CONVERSATION_TYPE_MISMATCH
AP_SYNC_LEVEL_NOT_SUPPORTED
AP_CONV_FAILURE_NO_RETRY
AP_CONV_FAILURE_RETRY
AP_DEALLOC_ABEND

AP_DEALLOC_ABEND_PROG

AP_DEALLOC_ABEND_TIMER

AP_PROG_ERROR_PURGING
AP_SVC_ERROR_PURGING

AP_CONVERSATION_TYPE_MIXED

AP_UNEXPECTED_SYSTEM_ERROR

AP_TP_BUSY
AP_CANCELLED

[MC_]CONFIRMED

The **CONFIRMED** verb replies to a confirmation request from the partner transaction program. It informs the partner transaction program that the local transaction program has not detected an error in the received data.

Because the transaction program issuing the confirmation request waits for a confirmation, the **CONFIRMED** verb synchronizes the processing of the two transaction programs.

VCB Structure

```
typedef struct confirmed
{
 unsigned short    opcode;            /* verb operation code     */
 unsigned char     opext;             /* verb extension code     */
 unsigned char     format;            /* format                  */
 unsigned short    primary_rc;        /* primary return code     */
 unsigned long     secondary_rc;      /* secondary return code   */
 unsigned char     tp_id[8];          /* TP identifier           */
 unsigned long     conv_id;           /* conversation identifier */
} CONFIRMED;
```

```
typedef struct mc_confirmed
{
 unsigned short     opcode;              /* verb operation code     */
 unsigned char      opext;               /* verb extension code     */
 unsigned char      format;              /* format                  */
 unsigned short     primary_rc;          /* primary return code     */
 unsigned long      secondary_rc;        /* secondary return code   */
 unsigned char      tp_id[8];            /* TP identifier           */
 unsigned long      conv_id;             /* conversation identifier */
} MC_CONFIRMED;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode
AP_B_CONFIRMED

AP_M_CONFIRMED

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION. For nonblocking operation, this flag can be ORed together with AP_NON_BLOCKING.

format
Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

tp_id Identifier for the local transaction program. The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

conv_id
Conversation identifier. The value of this parameter was returned by the

ALLOCATE verb in the invoking transaction processor or by **RECEIVE_ALLOCATE** in the invoked transaction processor.

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameter:

primary_rc
AP_OK

If the verb is nonblocking and has not completed, Communications Server returns the following parameters:

primary_rc
AP_OPERATION_INCOMPLETE

opext AP_OPERATION_INCOMPLETE_FLAG

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_CONV_ID

AP_BAD_TP_ID
AP_CONFIRMED_INVALID_FOR_FDX

If the conversation is in the wrong state when the transaction processor issues this verb, Communications Server returns the following parameters:

primary_rc
AP_STATE_CHECK

secondary_rc
AP_CONFIRMED_BAD_STATE

The conditions generating the following possible primary return codes (**primary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_TP_BUSY

AP_UNEXPECTED_SYSTEM_ERROR

AP_CONVERSATION_TYPE_MIXED

[MC_]DEALLOCATE

The **DEALLOCATE** verb deallocates a conversation between two transaction programs. Before deallocating the conversation, this verb performs the equivalent of one of the following verbs:

- The **FLUSH** verb, which sends the contents of the local LU's send buffer to the partner LU (and transaction processor).
- The **CONFIRM** verb, which sends the contents of the local LU's send buffer and a confirmation request to the partner transaction programs.

After this verb has successfully executed, the conversation ID is no longer valid.

For half-duplex conversation:

- Deallocates the specified conversation from the transaction program, it can include the function of the **FLUSH** or **CONFIRM** verb.

For full-duplex conversation

- **DEALLOCATE** with **TYPE(FLUSH)** closes the local program's send queue. Both the local and remote programs must close their send queues independently therefore, two **DEALLOCATE TYPE(FLUSH)** verbs are required to end the conversation. Notification that the partner has closed its send queue is given to the receive queue in the form of a **DEALLOCATE_NORMAL** return code.
- **DEALLOCATE** with **TYPE(ABEND)** is an abrupt termination that will close both sides of the conversation simultaneously. This notification is returned to the remote program's send queue as an **ERROR_INDICATION** return code, and to remote program's receive queue as a **DEALLOCATE_ABEND** return code.

VCB Structure

```
typedef struct deallocate
{
 unsigned short    opcode;             /* verb operation code     */
 unsigned char     opext;              /* verb extension code     */
 unsigned char     format;             /* format                  */
 unsigned short    primary_rc;         /* primary return code     */
 unsigned long     secondary_rc;       /* secondary return code   */
 unsigned char     tp_id[8];           /* TP identifier           */
 unsigned long     conv_id;            /* conversation identifier */
#ifdef WINAPPC_FORMAT_1
 unsigned char     expd_data_rcvd;     /* expedited data received */
 unsigned char     reserv3;            /* reserved                */
#endif
 unsigned char     dealloc_type;       /* deallocate type         */
 unsigned short    log_dlen;           /* log data length         */
 unsigned char    *log_dptr;           /* pointer to log data     */
} DEALLOCATE;
```

```
typedef struct mc_deallocate
{
 unsigned short    opcode;              /* verb operation code     */
 unsigned char     opext;               /* verb extension code     */
 unsigned char     format;              /* format                  */
 unsigned short    primary_rc;          /* primary return code     */
 unsigned long     secondary_rc;        /* secondary return code   */
 unsigned char     tp_id[8];            /* TP identifier           */
 unsigned long     conv_id;             /* conversation identifier */
#ifdef WINAPPC_FORMAT_1
 unsigned char     expd_data_rcvd;      /* expedited data received */
 unsigned char     reserv3;             /* reserved                */
#endif
```

```
     unsigned char     dealloc_type;          /* deallocate type          */
     unsigned char     reserv4[2];            /* reserved                 */
     unsigned char     reserv5[4];            /* reserved                 */
    } MC_DEALLOCATE;
```

Supplied Parameters

The transaction programs supplies the following parameters to Communications Server:

opcode
AP_B_DEALLOCATE

AP_M_DEALLOCATE

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION. For nonblocking operation, this flag can be ORed together with AP_NON_BLOCKING.

On full-duplex conversations, this flag must be ORed together with AP_FULL_DUPLEX_CONVERSATION.

format
Identifies the format of the VCB. Set this field to one to specify the version of the VCB listed above.

tp_id Identifier for the local transaction program. The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction processor or by **RECEIVE_ALLOCATE** in the invoked transaction program.

conv_id
Conversation identifier. The value of this parameter was returned by the **ALLOCATE** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

dealloc_type
Specifies how to perform the deallocation.

AP_ABEND

AP_ABEND_PROG

AP_ABEND_SVC

AP_ABEND_TIMER

AP_FLUSH

AP_SYNC_LEVEL

The following parameters apply to basic only.

AP_TP_NOT_AVAIL_NO_RETRY

AP_TP_NOT_AVAIL_RETRY

AP_TPN_NOT_RECOGNIZED

AP_PIP_DATA_NOT_ALLOWED
AP_PIP_DATA_INCORRECT
AP_RESOURCE_FAILURE_NO_RETRY
AP_CONV_TYPE_MISMATCH
AP_SYNC_LVL_NOT_SUPPORTED
AP_SECURITY_PARAMS_INVALID

log_dlen

Number of bytes of data to be sent to the error log file.

Range: 0–32767

The application can append data to the end of the VCB, in which case this field will be greater than zero and **log_dptr** must be set to NULL. (A length of zero indicates that there is no error log data.)

log_dptr

Address of data buffer containing error information. The application can append data to the end of the VCB, in which case **log_dptr** must be set to NULL.

This data is sent to the local error log and to the partner LU. The transaction processor must format the error data as a General Data Stream (GDS) error log variable. For further information, see *IBM Systems Network Architecture: LU 6.2 Reference: Peer Protocols*

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameter:

primary_rc
AP_OK

expd_data_rcvd
Expedited-data-received indicator. This indication continues to be set to AP_YES until a RECEIVE_EXPEDITED_DATA is issued.

This field requires the format 1 version of the VCB. See “Full-Duplex VCBs” on page 39 for more details on accessing format 1 VCBs.

AP_YES
AP_NO

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_CONV_ID

AP_BAD_TP_ID
AP_DEALLOC_BAD_TYPE

AP_DEALLOC_LOG_LL_WRONG

If the verb does not execute because of a parameter error, Communications Server returns the following parameters (for mapped only):

primary_rc
AP_OPERATION_INCOMPLETE

opext AP_OPERATION_INCOMPLETE_FLAG

If the conversation is in the wrong state when the transaction processor issues this verb, Communications Server returns the following parameters:

primary_rc
AP_STATE_CHECK

secondary_rc
AP_DEALLOC_CONFIRM_BAD_STATE
AP_DEALLOC_FLUSH_BAD_STATE
AP_DEALLOC_NOT_LL_BDY

The conditions generating the following possible primary return codes (**primary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_ALLOCATION_ERROR
- AP_SECURITY_NOT_VALID
- AP_TRANS_PGM_NOT_AVAIL_RETRY
- AP_TRANS_PGM_NOT_AVAIL_NO_RTRY
- AP_TP_NAME_NOT_RECOGNIZED
- AP_PIP_NOT_ALLOWED
- AP_PIP_NOT_SPECIFIED_CORRECTLY
- AP_CONVERSATION_TYPE_MISMATCH
- AP_SYNC_LEVEL_NOT_SUPPORTED
- AP_CONV_FAILURE_NO_RETRY
- AP_CONV_FAILURE_RETRY
- AP_DEALLOC_ABEND
- AP_DEALLOC_ABEND_PROG
- AP_DEALLOC_ABEND_SVC
- AP_DEALLOC_ABEND_TIMER
- AP_PROG_ERROR_PURGING
- AP_SVC_ERROR_PURGING

AP_TP_BUSY
AP_CONVERSATION_TYPE_MIXED
AP_DUPLEX_TYPE_MIXED
AP_UNEXPECTED_SYSTEM_ERROR
AP_CANCELLED
AP_ERROR_INDICATION

  AP_ALLOCATION_ERROR_PENDING

  AP_DEALLOC_ABEND_PROG_PENDING

  AP_DEALLOC_ABEND_SVC_PENDING

  AP_DEALLOC_ABEND_TIMER_PENDING

  AP_UNKNOWN_ERROR_TYPE_PENDING

[MC_]FLUSH

The **FLUSH** verb sends the contents of the local LU's send buffer to the partner LU (and transaction program). If the send buffer is empty, no action takes place.

VCB Structure

```
typedef struct flush
{
 unsigned short    opcode;            /* verb operation code     */
 unsigned char     opext;             /* verb extension code     */
 unsigned char     format;            /* format                  */
 unsigned short    primary_rc;        /* primary return code     */
 unsigned long     secondary_rc;      /* secondary return code   */
 unsigned char     tp_id[8];          /* TP identifier           */
 unsigned long     conv_id;           /* conversation identifier */
} FLUSH;
typedef struct mc_flush
{
 unsigned short      opcode;              /* verb operation code     */
 unsigned char       opext;               /* verb extension code     */
 unsigned char       format;              /* format                  */
 unsigned short      primary_rc;          /* primary return code     */
 unsigned long       secondary_rc;        /* secondary return code   */
 unsigned char       tp_id[8];            /* TP identifier           */
 unsigned long       conv_id;             /* conversation identifier */
} MC_FLUSH;
```

Supplied Parameters

The transaction processor supplies the following parameters to Communications Server: .

opcode
AP_B_FLUSH

AP_M_FLUSH

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION. For nonblocking operation, this flag can be ORed together with AP_NON_BLOCKING.

On full-duplex conversation, this flag must be ORed together with AP_FULL_DUPLEX_CONVERSATION.

format
Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

tp_id Identifier for the local transaction program. The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

conv_id
Conversation identifier. The value of this parameter was returned by the **ALLOCATE** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameter:

primary_rc
AP_OK

If the verb is nonblocking and has not completed, Communications Server returns the following parameters:

primary_rc
AP_OPERATION_INCOMPLETE

opext AP_OPERATION_INCOMPLETE_FLAG

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_CONV_ID

AP_BAD_TP_ID

If the conversation is in the wrong state when the transaction program issues this verb, Communications Server returns the following parameters:

primary_rc
AP_STATE_CHECK

secondary_rc
AP_FLUSH_NOT_SEND_STATE

The conditions generating the following possible primary return codes (**primary_rc**) and indented secondary return codes (**secondary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_TP_BUSY
AP_CONVERSATION_TYPE_MIXED
AP_DUPLEX_TYPE_MIXED
AP_UNEXPECTED_SYSTEM_ERROR
AP_ERROR_INDICATION
- AP_ALLOCATION_ERROR_PENDING
- AP_DEALLOC_ABEND_PROG_PENDING
- AP_DEALLOC_ABEND_SVC_PENDING
- AP_DEALLOC_ABEND_TIMER_PENDING
- AP_UNKNOWN_ERROR_TYPE_PENDING

[MC_]GET_ATTRIBUTES

The **GET_ATTRIBUTES** verb returns the attributes of the conversation.

VCB Structure

```
typedef struct get_attributes
{
 unsigned short    opcode;                 /* verb operation code          */
 unsigned char     opext;                  /* verb extension code          */
 unsigned char     format;                 /* verb format                  */
 unsigned short    primary_rc;             /* primary return code          */
 unsigned long     secondary_rc;           /* secondary return code        */
 unsigned char     tp_id[8];               /* TP identifier                */
 unsigned long     conv_id;                /* conversation identifier      */
 unsigned char     reserv3;                /* reserved                     */
 unsigned char     sync_level;             /* sync_level                   */
 unsigned char     mode_name[8];           /* mode name                    */
 unsigned char     net_name[8];            /* network name of local LU     */
 unsigned char     lu_name[8];             /* local LU name                */
 unsigned char     lu_alias[8];            /* local LU alias               */
 unsigned char     plu_alias[8];           /* partner LU alias             */
 unsigned char     plu_un_name[8];
                                           /* partner LU uninterpreted name */
 unsigned char     reserv4[2];             /* reserved                     */
 unsigned char     fqplu_name[17];
                                           /* fully qualified partner LU   */
                                           /* name                         */
 unsigned char     reserv5;                /* reserved                     */
 unsigned char     user_id[10];            /* user identifier              */
 unsigned long     conv_group_id;          /* conversation group identifier */
 unsigned char     conv_corr_len;          /* conversation correlator      */
                                           /* length                       */
 unsigned char     conv_corr[8];           /* conversation correlator      */
 unsigned char     reserv6[13];            /* reserved                     */
} GET_ATTRIBUTES;
```

```
typedef struct mc_get_attributes
{
 unsigned short    opcode;                  /* verb operation code          */
 unsigned char     opext;                   /* verb extension code          */
 unsigned char     format;                  /* verb format                  */
 unsigned short    primary_rc;              /* primary return code          */
 unsigned long     secondary_rc;            /* secondary return code        */
 unsigned char     tp_id[8];                /* TP identifier                */
 unsigned long     conv_id;                 /* conversation identifier      */
 unsigned char     reserv3;                 /* reserved                     */
 unsigned char     sync_level;              /* sync_level                   */
 unsigned char     mode_name[8];            /* mode name                    */
 unsigned char     net_name[8];             /* network name of local LU     */
 unsigned char     lu_name[8];              /* local LU name                */
 unsigned char     lu_alias[8];             /* local LU alias               */
 unsigned char     plu_alias[8];            /* partner LU alias             */
 unsigned char     plu_un_name[8];          /* partner LU uninterpreted name */
 unsigned char     reserv4[2];              /* reserved                     */
 unsigned char     fqplu_name[17];          /* fully qualified partner LU   */
                                            /* name                         */
 unsigned char     reserv5;                 /* reserved                     */
 unsigned char     user_id[10];             /* user identifier              */
 unsigned long     conv_group_id;           /* conversation group identifier */
 unsigned char     conv_corr_len;           /* conversation correlator      */
                                            /* length                       */
 unsigned char     conv_corr[8];            /* conversation correlator      */
 unsigned char     reserv6[13];             /* reserved                     */
} MC_GET_ATTRIBUTES;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode
AP_B_GET_ATTRIBUTES

AP_M_GET_ATTRIBUTES

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION.

On full-duplex conversations, this flag must be ORed together with AP_FULL_DUPLEX_CONVERSATION.

format
Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

tp_id Identifier for the local transaction program

The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

conv_id
Conversation identifier.

The value of this parameter was returned by the **ALLOCATE** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameters:

primary_rc
AP_OK

sync_level
Synchronization level of the conversation.

AP_CONFIRM_SYNC_LEVEL
AP_NONE

mode_name
Name of the set of networking characteristics associated with the session allocated to the conversation. This is an 8-byte alphanumeric type-A EBCDIC string (starting with a letter), padded to the right with EBCDIC spaces.

net_name
Name of the network containing the local LU. This is an 8-byte alphanumeric type-A EBCDIC string (starting with a letter), padded to the right with EBCDIC spaces.

lu_name
Name of the local LU. This is an 8-byte alphanumeric type-A EBCDIC string (starting with a letter), padded to the right with EBCDIC spaces.

lu_alias
Alias by which the local LU is known to the local transaction program. This is an 8-byte string in a locally displayable character set. All 8 bytes are significant and must be set.

plu_alias
Alias by which the partner LU is known to the local transaction program. This is an 8-byte string in a locally displayable character set. All 8 bytes are significant and must be set.

plu_un_name
Uninterpreted name of partner LU, that is, the name of the partner LU as defined at the system services control point (SSCP). This is an 8-byte type-A EBCDIC character string.

fqplu_name
Fully qualified name of the partner LU. This name is 17 bytes long and is right-padded with EBCDIC blanks. It is composed of two type-A EBCDIC character strings concatenated by an EBCDIC dot. (Each name can have a maximum length of 8 bytes with no embedded blanks. If the network ID is not present, then omit the dot.)

user_id
User ID sent by the invoking transaction program through the **ALLOCATE** verb to access the invoked transaction program (if applicable). This is a 10-byte type-AE EBCDIC character string, padded to the right with EBCDIC spaces.

conv_group_id
The conversation group identifier of the session allocated to the conversation.

conv_corr_len
Always set to 0.

Range: 0–8

conv_corr
Always set to 0.

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_CONV_ID

AP_BAD_TP_ID

The conditions generating the following possible primary return codes (**primary_rc**) and indented secondary return codes (**secondary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_TP_BUSY

AP_CONVERSATION_TYPE_MIXED

AP_DUPLEX_TYPE_MIXED

AP_UNEXPECTED_SYSTEM_ERROR

[MC_]PREPARE_TO_RECEIVE

The **PREPARE_TO_RECEIVE** verb changes the state of the conversation for the local transaction program from SEND or SEND_PENDING to RECEIVE.

Before changing the conversation state, this verb performs the equivalent of one of the following verbs:

- The **FLUSH** verb, which sends the contents of the local LU's send buffer to the partner LU (and transaction program).
- The **CONFIRM** verb, which send the contents of the local LU's send buffer and a confirmation request to the partner transaction program.

After this verb has successfully executed, the local transaction program can receive data.

VCB Structure

```
typedef struct prepare_to_receive
{
 unsigned short    opcode;                /* verb operation code      */
 unsigned char     opext;                 /* verb extension code      */
 unsigned char     format;                /* format                   */
 unsigned short    primary_rc;            /* primary return code      */
 unsigned long     secondary_rc;          /* secondary return code    */
 unsigned char     tp_id[8];              /* TP identifier            */
 unsigned long     conv_id;               /* conversation identifier  */
 unsigned char     ptr_type;              /* prepare to receive type  */
 unsigned char     locks;                 /* prepare to receive locks */
} PREPARE_TO_RECEIVE;
```

```
typedef struct mc_prepare_to_receive
{
 unsigned short     opcode;               /* verb operation code      */
 unsigned char      opext;                /* verb extension code      */
 unsigned char      format;               /* format                   */
 unsigned short     primary_rc;           /* primary return code      */
 unsigned long      secondary_rc;         /* secondary return code    */
 unsigned char      tp_id[8];             /* TP identifier            */
 unsigned long      conv_id;              /* conversation identifier  */
 unsigned char      ptr_type;             /* prepare to receive type  */
 unsigned char      locks;                /* prepare to receive locks */
} MC_PREPARE_TO_RECEIVE;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode
AP_B_PREPARE_TO_RECEIVE

AP_M_PREPARE_TO_RECEIVE

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION. For nonblocking operation, this flag can be ORed together with AP_NON_BLOCKING.

format
Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

tp_id Identifier for the local transaction program. The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

conv_id
Conversation identifier.

The value of this parameter was returned by the **ALLOCATE** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

ptr_type
Specifies how to perform the state change.

AP_FLUSH
AP_SYNC_LEVEL
AP_P_TO_R_CONFIRM

locks Specifies when Communications Server is to return control to the local transaction processor.

AP_LONG

AP_SHORT

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameter:

primary_rc
AP_OK

If the verb is nonblocking and has not completed, Communications Server returns the following parameters;

primary_rc
AP_OPERATION_INCOMPLETE

opext AP_OPERATION_INCOMPLETE_FLAG

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_CONV_ID

AP_BAD_TP_ID
AP_P_TO_R_INVALID_FOR_FDX
AP_P_TO_R_INVALID_TYPE

If the conversation is in the wrong state when the transaction processor issues this verb, Communications Server returns the following parameters:

primary_rc
AP_STATE_CHECK

secondary_rc

AP_TO_R_NOT_LL_BDY

AP_P_TO_R_NOT_SEND_STATE

The conditions generating the following possible primary return codes (**primary_rc**) and indented secondary return codes (**secondary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_ALLOCATION_ERROR
- AP_SECURITY_NOT_VALID
- AP_TRANS_PGM_NOT_AVAIL_RETRY
- AP_TRANS_PGM_NOT_AVAIL_NO_RTRY
- AP_TP_NAME_NOT_RECOGNIZED
- AP_PIP_NOT_ALLOWED
- AP_PIP_NOT_SPECIFIED_CORRECTLY
- AP_CONVERSATION_TYPE_MISMATCH
- AP_SYNC_LEVEL_NOT_SUPPORTED
- AP_CONV_FAILURE_NO_RETRY

AP_CONV_FAILURE_RETRY

AP_DEALLOC_ABEND

AP_DEALLOC_ABEND_PROG

AP_DEALLOC_ABEND_SVC

AP_DEALLOC_ABEND_TIMER

AP_PROG_ERROR_PURGING

AP_SVC_ERROR_PURGING

AP_TP_BUSY

AP_CONVERSATION_TYPE_MIXED

AP_UNEXPECTED_SYSTEM_ERROR

AP_CANCELLED

[MC_]RECEIVE_AND_POST

The **RECEIVE_AND_POST** verb receives application data and status information asynchronously. This enables the transaction program to proceed with processing while data is still arriving at the local LU. This verb can only be issued through the APPC entry point.

Not available on Win 3.1 SNA API clients.

VCB Structure

```
typedef struct receive_and_post
{
 unsigned short    opcode;            /* verb operation code       */
 unsigned char     opext;             /* verb extension code       */
 unsigned char     format;            /* format                    */
 unsigned short    primary_rc;        /* primary return code       */
 unsigned long     secondary_rc;      /* secondary return code     */
 unsigned char     tp_id[8];          /* TP identifier             */
 unsigned long     conv_id;           /* conversation identifier   */
 unsigned short    what_rcvd;         /* what received             */
 unsigned char     rtn_status;        /* return status with data   */
 unsigned char     fill;              /* data fill                 */
 unsigned char     rts_rcvd;          /* request to send received  */
 unsigned char     expd_data_rcvd;    /* expedited data received   */
 unsigned short    max_len;           /* maximum length of received */
                                      /* data                      */
 unsigned short    dlen;              /* actual length of received  */
                                      /* data                      */
 unsigned char    *dptr;              /* pointer to data buffer    */
 unsigned long    *sema;              /* post handle for verb      */
 unsigned char     reserv5;           /* reserved                  */
} RECEIVE_AND_POST;
typedef struct mc_receive_and_post
{
 unsigned short   opcode;            /* verb operation code       */
 unsigned char    opext;             /* verb extension code       */
 unsigned char    format;            /* format                    */
 unsigned short   primary_rc;        /* primary return code       */
 unsigned long    secondary_rc;      /* secondary return code     */
 unsigned char    tp_id[8];          /* TP identifier             */
 unsigned long    conv_id;           /* conversation identifier   */
 unsigned short   what_rcvd;         /* what received             */
 unsigned char    rtn_status;        /* return status with data   */
 unsigned char    reserv4;           /* reserved                  */
 unsigned char    rts_rcvd;          /* request to send received  */
 unsigned char    expd_data_rcvd;    /* expedited data received   */
 unsigned short   max_len;           /* maximum length of received */
                                     /* data                      */
 unsigned short   dlen;              /* actual length of received  */
                                     /* data                      */
 unsigned char   *dptr;              /* pointer to data buffer    */
 unsigned long   *sema;              /* post handle for verb      */
 unsigned char    reserv6;           /* reserved                  */
} MC_RECEIVE_AND_POST;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode
AP_B_RECEIVE_AND_POST

AP_M_RECEIVE_AND_POST

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION.

format
Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

tp_id Identifier for the local transaction program. The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

conv_id
Conversation identifier.

The value of this parameter was returned by the **ALLOCATE** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

rtn_status
Indicates whether status information and data can be returned on the same verb.

AP_YES

AP_NO

fill

Indicates the manner in which the local transaction program receives data.

AP_BUFFER

AP_LL

max_len
Maximum number of bytes of data the local transaction program can receive.

Range: 0–65535

This value must not exceed the length of the buffer to contain the received data.

dptr Address of the buffer to contain the data received by the local LU. The application can append data to the end of the VCB in which case **dptr** must be set to NULL.

sema Handle of the event that the application will wait on. This verb is intended for use with WaitForMultipleObjects in the Win32 API or DosWaitEventSem for OS/2.

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameters:

primary_rc

AP_OK

AP_DEALLOC_NORMAL

what_rcvd

Status information received with the incoming data. If **rtn_status** is set to AP_NO, this field always contains a value from the first part of the following list. If **rtn_status** is set to AP_YES, this field can contain any value from the list.

AP_NONE

AP_CONFIRM_DEALLOCATE

AP_CONFIRM_SEND

AP_CONFIRM_WHAT_RECEIVED

AP_DATA

AP_DATA_COMPLETE

AP_DATA_INCOMPLETE

AP_SEND

AP_USER_CONTROL_DATA_COMPLETE

AP_USER_CONTROL_DATA_INCMP

AP_PS_HEADER_COMPLETE

AP_PS_HEADER_INCOMPLETE

AP_DATA_CONFIRM

AP_DATA_COMPLETE_CONFIRM

AP_DATA_CONFIRM_DEALLOCATE

AP_DATA_COMPLETE_CONFIRM_DEALL

AP_DATA_CONFIRM_SEND

AP_DATA_COMPLETE_CONFIRM_SEND

AP_DATA_SEND

AP_DATA_COMPLETE_SEND

The following parameters are for mapped only:

AP_UC_DATA_COMPLETE_CONFIRM

AP_UC_DATA_COMPLETE_CNFM_DEALL

AP_UC_DATA_COMPLETE_CNFM_SEND

AP_UC_DATA_COMPLETE_SEND

AP_PS_HDR_COMPLETE_CONFIRM

AP_PS_HDR_COMPLETE_CNFM_DEALL

AP_PS_HDR_COMPLETE_CNFM_SEND

AP_PS_HDR_COMPLETE_SEND

rts_rcvd
Request-to-send-received indicator.

AP_YES

AP_NO

expd_data_rcvd
Expedited-data-received indicator. This indication continues to be set to AP_YES until a RECEIVE_EXPEDITED_DATA is issued.

AP_YES

AP_NO

This format field requires the format 1 version of the VCB. See “Full-Duplex VCBs” on page 39 for more details on accessing format 1 VCBs.

dlen Number of bytes of data received (the data is stored in the buffer specified by the **dptr** parameter). A length of zero indicates that no data was received. This parameter is only used if the **what_rcvd** parameter indicates that data was received.

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_CONV_ID

AP_BAD_RETURN_STATUS_WITH_DATA

AP_BAD_TP_ID

AP_RCV_AND_POST_BAD_FILL

If the conversation is in the wrong state when the transaction program issues this verb, Communications Server returns the following parameters:

primary_rc
AP_STATE_CHECK

secondary_rc
AP_RCV_AND_POST_BAD_STATE

AP_RCV_AND_POST_NOT_LL_BDY

If the verb did not execute because it was canceled by another verb issued by the transaction program, Communications Server returns the following parameter:

primary_rc
AP_CANCELLED

The conditions generating the following possible primary return codes (**primary_rc**) and indented secondary return codes (**secondary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_ALLOCATION_ERROR

- AP_SECURITY_NOT_VALID
- AP_TRANS_PGM_NOT_AVAIL_RETRY
- AP_TRANS_PGM_NOT_AVAIL_NO_RTRY
- AP_TP_NAME_NOT_RECOGNIZED
- AP_PIP_NOT_ALLOWED
- AP_PIP_NOT_SPECIFIED_CORRECTLY
- AP_CONVERSATION_TYPE_MISMATCH
- AP_SYNC_LEVEL_NOT_SUPPORTED
- AP_CONV_FAILURE_NO_RETRY
- AP_CONV_FAILURE_RETRY
- AP_DEALLOC_ABEND
- AP_DEALLOC_ABEND_PROG
- AP_DEALLOC_ABEND_SVC
- AP_DEALLOC_ABEND_TIMER
- AP_DEALLOC_NORMAL
- AP_PROG_ERROR_NO_TRUNC
- AP_PROG_ERROR_PURGING
- AP_PROG_ERROR_TRUNC
- AP_SVC_ERROR_NO_TRUNC
- AP_SVC_ERROR_PURGING
- AP_SVC_ERROR_TRUNC
- AP_TP_BUSY
- AP_CONVERSATION_TYPE_MIXED
- AP_UNEXPECTED_SYSTEM_ERROR
- AP_CANCELLED

[MC]RECEIVE_AND_WAIT

The **RECEIVE_AND_WAIT** verb receives any data that is currently available from the partner transaction program. If no data is currently available, the local transaction program waits for data to arrive.

For half-duplex conversations:

- The program can issue this verb when the conversation is in send state. In this case, the LU flushes its send buffer, sending all buffered information and the SEND indication to the remote program. This changes the conversation to receive state. The LU then waits for information to arrive. The remote program can send data to the local program after it receives the SEND indication.

For full-duplex conversations:

- If the send buffer contains the conversation allocation request, it will be flushed; otherwise, this verb will not cause the LU to flush its send buffer. If it is important that the data remaining in the send buffer be transmitted before receiving data, the local program should issue a FLUSH before issuing this verb.

VCB Structure

```
typedef struct receive_and_wait
{
 unsigned short    opcode;             /* verb operation code        */
 unsigned char     opext;              /* verb extension code        */
 unsigned char     format;             /* format                     */
 unsigned short    primary_rc;         /* primary return code        */
 unsigned long     secondary_rc;       /* secondary return code      */
 unsigned char     tp_id[8];           /* TP identifier              */
 unsigned long     conv_id;            /* conversation identifier    */
 unsigned short    what_rcvd;          /* what received              */
 unsigned char     rtn_status;         /* return status with data    */
 unsigned char     fill;               /* data fill                  */
 unsigned char     rts_rcvd;           /* request to send received   */
 unsigned char     expd_data_rcvd;     /* expedited data received    */
 unsigned short    max_len;            /* maximum length of received */
                                       /* data                       */
 unsigned short    dlen;               /* actual length of received  */
                                       /* data                       */
 unsigned char    *dptr;               /* pointer to data buffer     */
 unsigned char     reserv5[5];         /* reserved                   */
} RECEIVE_AND_WAIT;
typedef struct mc_receive_and_wait
{
 unsigned short      opcode;             /* verb operation code        */
 unsigned char       opext;              /* verb extension code        */
 unsigned char       format;             /* format                     */
 unsigned short      primary_rc;         /* primary return code        */
 unsigned long       secondary_rc;       /* secondary return code      */
 unsigned char       tp_id[8];           /* TP identifier              */
 unsigned long       conv_id;            /* conversation identifier    */
 unsigned short      what_rcvd;          /* what received              */
 unsigned char       rtn_status;         /* return status with data    */
 unsigned char       reserv4;            /* reserved                   */
 unsigned char       rts_rcvd;           /* request to send received   */
 unsigned char       expd_data_rcvd;     /* expedited data received    */
 unsigned short      max_len;            /* maximum length of received */
                                         /* data                       */
 unsigned short      dlen;               /* actual length of received  */
```

```
                                            /* data                        */
 unsigned char      *dptr;                  /* pointer to data buffer      */
 unsigned char       reserv6[5];            /* reserved                    */
} MC_RECEIVE_AND_WAIT;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode
AP_B_RECEIVE_AND_WAIT

AP_M_RECEIVE_AND_WAIT

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION. For nonblocking operation, this flag can be ORed together with AP_NON_BLOCKING.

On full-duplex conversations, this flag must be ORed together with AP_FULL_DUPLEX_CONVERSATION.

format
Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

tp_id Identifier for the local transaction program.

The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

conv_id
Conversation identifier.

The value of this parameter was returned by the **ALLOCATE** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

rtn_status
Indicates whether status information and data can be returned on the same verb.

AP_YES

AP_NO

fill

Indicates the manner in which the local transaction program receives data.

AP_BUFFER

AP_LL

max_len
Maximum number of bytes of data the local transaction program can receive.

Range: 0–65535

This value must not exceed the length of the buffer to contain the received data.

dptr Address of the buffer to contain the data received by the local LU. The application can append data to the end of the VCB, in which case **dptr** must be set to NULL.

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameters:

primary_rc
AP_OK

AP_DEALLOC_NORMAL

what_rcvd
Status information received with the incoming data. If **rtn_status** is set to AP_NO, this field always contains a value from the first part of the following list. If **rtn_status** is set to AP_YES, this field can contain any value from the list.

AP_NONE

AP_CONFIRM_DEALLOCATE

AP_CONFIRM_SEND

AP_CONFIRM_WHAT_RECEIVED

AP_DATA

AP_DATA_COMPLETE

AP_DATA_INCOMPLETE

AP_SEND

AP_USER_CONTROL_DATA_COMPLETE

AP_USER_CONTROL_DATA_INCMP

AP_PS_HEADER_COMPLETE

AP_PS_HEADER_INCOMPLETE

AP_DATA_CONFIRM

AP_DATA_COMPLETE_CONFIRM

AP_DATA_CONFIRM_DEALLOCATE

AP_DATA_COMPLETE_CONFIRM_DEALL

AP_DATA_CONFIRM_SEND

AP_DATA_COMPLETE_CONFIRM_SEND

AP_DATA_SEND

AP_DATA_COMPLETE_SEND

The following parameters apply to mapped only:

AP_UC_DATA_COMPLETE_CONFIRM

AP_UC_DATA_COMPLETE_CNFM_DEALL

AP_UC_DATA_COMPLETE_CNFM_SEND

AP_UC_DATA_COMPLETE_SEND

AP_PS_HDR_COMPLETE_CONFIRM

AP_PS_HDR_COMPLETE_CNFM_DEALL

AP_PS_HDR_COMPLETE_CNFM_SEND

AP_PS_HDR_COMPLETE_SEND

rts_rcvd
Request-to-send-received indicator.

AP_YES

AP_NO

This format of the following verb is the format 1 version of the VCB. See "Full-Duplex VCBs" on page 39 for more details on accessing format 1 VCBs.

expd_data_rcvd
Expedited-data-received indicator. This indication continues to be set to AP_YES until a RECEIVE_EXPEDITED_DATA is issued.

AP_YES

AP_NO

dlen This parameter is only used if the **what_rcvd** parameter indicates that data was received. Number of bytes of data received (the data is stored in the buffer specified by the **dptr** parameter). A length of zero indicates that no data was received.

If the verb is nonblocking and has not completed, Communications Server returns the following parameters;

primary_rc
AP_OPERATION_INCOMPLETE

opext AP_OPERATION_INCOMPLETE_FLAG

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_CONV_ID

AP_BAD_RETURN_STATUS_WITH_DATA

AP_BAD_TP_ID

AP_RCV_AND_WAIT_BAD_FILL

If the conversation is in the wrong state when the transaction program issues this verb, Communications Server returns the following parameters:

primary_rc
AP_STATE_CHECK

secondary_rc
AP_RCV_AND_WAIT_BAD_STATE

AP_RCV_AND_WAIT_NOT_LL_BDY

The conditions generating the following possible primary return codes (**primary_rc**) and indented secondary return codes (**secondary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_ALLOCATION_ERROR
- AP_SECURITY_NOT_VALID
- AP_TRANS_PGM_NOT_AVAIL_RETRY
- AP_TRANS_PGM_NOT_AVAIL_NO_RTRY
- AP_TP_NAME_NOT_RECOGNIZED
- AP_PIP_NOT_ALLOWED
- AP_PIP_NOT_SPECIFIED_CORRECTLY
- AP_CONVERSATION_TYPE_MISMATCH
- AP_SYNC_LEVEL_NOT_SUPPORTED
- AP_CONV_FAILURE_NO_RETRY
- AP_CONV_FAILURE_RETRY
- AP_DEALLOC_ABEND
- AP_DEALLOC_ABEND_PROG
- AP_DEALLOC_ABEND_SVC
- AP_DEALLOC_ABEND_TIMER
- AP_DEALLOC_NORMAL
- AP_PROG_ERROR_NO_TRUNC
- AP_PROG_ERROR_PURGING
- AP_PROG_ERROR_TRUNC
- AP_SVC_ERROR_NO_TRUNC
- AP_SVC_ERROR_PURGING

AP_SVC_ERROR_TRUNC

AP_TP_BUSY
AP_CONVERSATION_TYPE_MIXED
AP_DUPLEX_TYPE_MIXED
AP_UNEXPECTED_SYSTEM_ERROR
AP_CANCELLED

[MC_]RECEIVE_EXPEDITED_DATA

This is only supported for applications residing on the server.

The **[MC_]RECEIVE_EXPEDITED_DATA** verb receives any expedited data that is currently available from the partner TP. If expedited data is currently available, the local transaction program receives it without waiting; otherwise, the behavior is governed by the **rtn_ctl** field.

VCB Structure

```
typedef struct receive_expedited_data
{
 unsigned short    opcode;             /* verb operation code        */
 unsigned char     opext;              /* verb extension code        */
 unsigned char     format;             /* format                     */
 unsigned short    primary_rc;         /* primary return code        */
 unsigned long     secondary_rc;       /* secondary return code      */
 unsigned char     tp_id[8];           /* TP identifier              */
 unsigned long     conv_id;            /* conversation identifier    */
 unsigned char     return_control;     /* when to return control     */
 unsigned char     reserv1[3];         /* reserved                   */
 unsigned char     rts_rcvd;           /* request to send received   */
 unsigned char     expd_data_rcvd;     /* expedited data received    */
 unsigned short    max_len;            /* maximum length of received */
                                       /* data                       */
 unsigned short    dlen;               /* actual length of received  */
                                       /* data                       */
 unsigned char    *dptr;               /* pointer to data buffer     */
} RECEIVE_EXPEDITED_DATA
```

```
typedef struct mc_receive_expedited_data
{
 unsigned short    opcode;             /* verb operation code        */
 unsigned char     opext;              /* verb extension code        */
 unsigned char     format;             /* format                     */
 unsigned short    primary_rc;         /* primary return code        */
 unsigned long     secondary_rc;       /* secondary return code      */
 unsigned char     tp_id[8];           /* TP identifier              */
 unsigned long     conv_id;            /* conversation identifier    */
 unsigned char     return_control;     /* when to return control     */
 unsigned char     reserv1[3];         /* reserved                   */
 unsigned char     rts_rcvd;           /* request to send received   */
 unsigned char     expd_data_rcvd;     /* expedited data received    */
 unsigned short    max_len;            /* maximum length of received */
                                       /* data                       */
 unsigned short    dlen;               /* actual length of received  */
                                       /* data                       */
 unsigned char    *dptr;               /* pointer to data buffer     */
} MC_RECEIVE_EXPEDITED_DATA
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode

AP_B_RECEIVE_EXPEDITED_DATA

AP_M_RECEIVE_EXPEDITED_DATA

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION. For nonblocking operation, this flag can be ORed together with AP_NON_BLOCKING.

On full-duplex conversations, this flag must be ORed together with AP_FULL_DUPLEX_CONVERSATION.

format
Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

tp_id Identifier for the local transaction program.

The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

conv_id
Conversation identifier.

The value of this parameter was returned by the **ALLOCATE** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

return_control
Specifies when to return control to the transaction program.

AP_WHEN_EXPD_RECEIVED

AP_IMMEDIATE

max_len
Maximum number of bytes of data the local transaction program can receive.

Range: 0–86

This value must not exceed the length of the buffer to contain the received data.

dptr Address of the buffer to contain the data received by the local LU. The application can append data to the end of the VCB, in which case **dptr** must be set to NULL.

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameters:

primary_rc
AP_OK

rts_rcvd
Request-to-send-received indicator.

AP_YES

AP_NO

expd_data_rcvd
Expedited-data-received indicator. This indication continues to be set to AP_YES until a RECEIVE_EXPEDITED_DATA is issued.

AP_YES
AP_NO

dlen Number of bytes of data received (the data is stored in the buffer specified by the **dptr** parameter). A length of zero indicates that no data was received. Note that any data received is unformatted. No 2-byte length field (LL) is present.

If the verb is nonblocking and has not completed, Communications Server returns the following parameters:

primary_rc
AP_OPERATION_INCOMPLETE

opext AP_OPERATION_INCOMPLETE_FLAG

If the verb does not execute because the remote LU does not support expedited data, Communications Server returns the following parameter:

primary_rc
AP_EXPD_NOT_SUPPORTED_BY_LU

If no data is immediately available from the partner transaction program and the **rtn_ctl** flag is AP_IMMEDIATE, Communications Server returns the following parameter:

primary_rc
AP_UNSUCCESSFUL

If the data buffer provided by the transaction program is not large enough to contain all of the expedited data available at the LU, no data is returned and Communications Server returns the following parameters:

primary_rc
AP_BUFFER_TOO_SMALL

dlen Number of bytes expedited data that the LU has available to receive.

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_CONV_ID

AP_BAD_TP_ID
AP_EXPD_BAD_RETURN_CONTROL
AP_RCV_EXPD_INVALID_LENGTH

If the conversation is in the wrong state when the transaction program issues this verb, Communications Server returns the following parameters:

primary_rc
AP_STATE_CHECK

secondary_rc
AP_EXPD_DATA_BAD_CONV_STATE

The conditions generating the following possible primary return codes (**primary_rc**) and indented secondary return codes (**secondary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_ALLOCATION_ERROR
- AP_SECURITY_NOT_VALID
- AP_TRANS_PGM_NOT_AVAIL_RETRY
- AP_TRANS_PGM_NOT_AVAIL_NO_RTRY
- AP_TP_NAME_NOT_RECOGNIZED
- AP_PIP_NOT_ALLOWED
- AP_PIP_NOT_SPECIFIED_CORRECTLY
- AP_CONVERSATION_TYPE_MISMATCH
- AP_SYNC_LEVEL_NOT_SUPPORTED
- AP_CONV_FAILURE_NO_RETRY
- AP_CONV_FAILURE_RETRY
- AP_DEALLOC_ABEND_PROG
- AP_DEALLOC_ABEND_SVC
- AP_DEALLOC_ABEND_TIMER
- AP_DEALLOC_NORMAL
- AP_TP_BUSY
- AP_CONVERSATION_TYPE_MIXED
- AP_DUPLEX_TYPE_MIXED
- AP_UNEXPECTED_SYSTEM_ERROR
- AP_CANCELLED
- AP_ERROR_INDICATION

[MC_]RECEIVE_IMMEDIATE

The **[MC_]RECEIVE_IMMEDIATE** verb receives any data or status information that is currently available from the partner transaction program. If none is currently available, the local transaction program returns immediately and does not wait.

VCB Structure

```
typedef struct receive_immediate
{
 unsigned short     opcode;           /* verb operation code        */
 unsigned char      opext;            /* verb extension code        */
 unsigned char      format;           /* format                     */
 unsigned short     primary_rc;       /* primary return code        */
 unsigned long      secondary_rc;     /* secondary return code      */
 unsigned char      tp_id[8];         /* TP identifier              */
 unsigned long      conv_id;          /* conversation identifier    */
 unsigned short     what_rcvd;        /* what received              */
 unsigned char      rtn_status;       /* return status with data    */
 unsigned char      fill;             /* data fill                  */
 unsigned char      rts_rcvd;         /* request to send received   */
 unsigned char      expd_data_rcvd;   /* expedited data received    */
 unsigned short     max_len;          /* maximum length of received */
                                      /* data                       */
 unsigned short     dlen;             /* actual length of received  */
                                      /* data                       */
 unsigned char     *dptr;             /* pointer to data buffer     */
 unsigned char      reserv5[5];       /* reserved                   */
} RECEIVE_IMMEDIATE;
```

```
typedef struct mc_receive_immediate
{
 unsigned short    opcode;            /* verb operation code        */
 unsigned char     opext;             /* verb extension code        */
 unsigned char     format;            /* format                     */
 unsigned short    primary_rc;        /* primary return code        */
 unsigned long     secondary_rc;      /* secondary return code      */
 unsigned char     tp_id[8];          /* TP identifier              */
 unsigned long     conv_id;           /* conversation identifier    */
 unsigned short    what_rcvd;         /* what received              */
 unsigned char     rtn_status;        /* return status with data    */
 unsigned char     reserv4;           /* reserved                   */
 unsigned char     rts_rcvd;          /* request to send received   */
 unsigned char     expd_data_rcvd;    /* expedited data received    */
 unsigned short    max_len;           /* maximum length of received */
                                      /* data                       */
 unsigned short    dlen;              /* actual length of received  */
                                      /* data                       */
 unsigned char    *dptr;              /* pointer to data buffer     */
 unsigned char     reserv6[5];        /* reserved                   */
} MC_RECEIVE_IMMEDIATE;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode

AP_B_RECEIVE_IMMEDIATE

AP_M_RECEIVE_IMMEDIATE

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION. For nonblocking operation, this flag can be ORed together with AP_NON_BLOCKING.

On full-duplex conversations, this flag must be ORed together with AP_FULL_DUPLEX_CONVERSATION.

format
Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

tp_id Identifier for the local transaction program.

The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

conv_id
Conversation identifier.

The value of this parameter was returned by the **[MC_]ALLOCATE** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

rtn_status
Indicates whether status information and data can be returned on the same verb.

AP_YES

AP_NO

fill

Indicates the manner in which the local transaction program receives data.

AP_BUFFER

AP_LL

max_len
Maximum number of bytes of data the local transaction program can receive.

Range: 0–65535

This value must not exceed the length of the buffer to contain the received data.

dptr Address of the buffer to contain the data received by the local LU. The application can append data to the end of the VCB, in which case **dptr** must be set to NULL.

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameters:

primary_rc
AP_OK

AP_DEALLOC_NORMAL

what_rcvd
Status information received with the incoming data. If **rtn_status** is set to

AP_NO, this field always contains a value from the first part of the following list. If **rtn_status** is set to AP_YES, this field can contain any value from the list.

AP_NONE
AP_CONFIRM_DEALLOCATE
AP_CONFIRM_SEND
AP_CONFIRM_WHAT_RECEIVED
AP_DATA
AP_DATA_COMPLETE
AP_DATA_INCOMPLETE
AP_SEND
AP_USER_CONTROL_DATA_COMPLETE
AP_USER_CONTROL_DATA_INCMP
AP_PS_HEADER_COMPLETE
AP_PS_HEADER_INCOMPLETE
AP_DATA_CONFIRM
AP_DATA_COMPLETE_CONFIRM
AP_DATA_CONFIRM_DEALLOCATE
AP_DATA_COMPLETE_CONFIRM_DEALL
AP_DATA_CONFIRM_SEND
AP_DATA_COMPLETE_CONFIRM_SEND
AP_DATA_SEND
AP_DATA_COMPLETE_SEND
AP_UC_DATA_COMPLETE_CONFIRM
AP_UC_DATA_COMPLETE_CNFM_DEALL
AP_UC_DATA_COMPLETE_CNFM_SEND
AP_UC_DATA_COMPLETE_SEND
AP_PS_HDR_COMPLETE_CONFIRM
AP_PS_HDR_COMPLETE_CNFM_DEALL

AP_PS_HDR_COMPLETE_CNFM_SEND

AP_PS_HDR_COMPLETE_SEND

expd_data_rcvd
Expedited-data-received indicator.

AP_YES

AP_NO

rts_rcvd
Request-to-send-received indicator.

AP_YES

AP_NO

dlen This parameter is only used if the **what_rcvd** parameter indicates that data was received. Number of bytes of data received (the data is stored in the buffer specified by the **dptr** parameter). A length of zero indicates that no data was received.

If the verb is nonblocking and has not completed, Communications Server returns the following parameter.

primary_rc
AP_OPERATION_INCOMPLETE

opext AP_BASIC_CONVERSION or AP_MAPPED_CONVERSATION ORed together with

AP_NON_BLOCKING ORed together with AP_OPERATION_INCOMPLETE_FLAG

If no data is immediately available from the partner transaction program, Communications Server returns the following parameter.

primary_rc
AP_UNSUCCESSFUL

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_CONV_ID

AP_BAD_RETURN_STATUS_WITH_DATA

AP_BAD_TP_ID

AP_RCV_IMMD_BAD_FILL

If the conversation is in the wrong state when the transaction program issues this verb, Communications Server returns the following parameters:

primary_rc
AP_STATE_CHECK

secondary_rc
AP_RCV_IMMD_BAD_STATE

The conditions generating the following possible primary return codes (**primary_rc**) and indented secondary return codes (**secondary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_ALLOCATION_ERROR
- AP_SECURITY_NOT_VALID
- AP_TRANS_PGM_NOT_AVAIL_RETRY
- AP_TRANS_PGM_NOT_AVAIL_NO_RTRY
- AP_TP_NAME_NOT_RECOGNIZED
- AP_PIP_NOT_ALLOWED
- AP_PIP_NOT_SPECIFIED_CORRECTLY
- AP_CONVERSATION_TYPE_MISMATCH
- AP_SYNC_LEVEL_NOT_SUPPORTED
- AP_CONV_FAILURE_NO_RETRY
- AP_CONV_FAILURE_RETRY
- AP_DEALLOC_ABEND
- AP_DEALLOC_ABEND_PROG
- AP_DEALLOC_ABEND_SVC
- AP_DEALLOC_ABEND_TIMER
- AP_DEALLOC_NORMAL
- AP_PROG_ERROR_NO_TRUNC
- AP_PROG_ERROR_PURGING
- AP_PROG_ERROR_TRUNC
- AP_SVC_ERROR_NO_TRUNC
- AP_SVC_ERROR_PURGING
- AP_SVC_ERROR_TRUNC
- AP_TP_BUSY
- AP_CONVERSATION_TYPE_MIXED
- AP_UNEXPECTED_SYSTEM_ERROR
- AP_DUPLEX_TYPE_MIXED
- AP_CANCELLED

[MC_]REQUEST_TO_SEND

The **[MC_]REQUEST_TO_SEND** verb notifies the partner transaction program that the local transaction program wants to send data.

VCB Structure

```
typedef struct request_to_send
{
 unsigned short    opcode;                /* verb operation code     */
 unsigned char     opext;                 /* verb extension code     */
 unsigned char     format;;               /* format                  */
 unsigned short    primary_rc;            /* primary return code     */
 unsigned long     secondary_rc;          /* secondary return code   */
 unsigned char     tp_id[8];              /* TP identifier           */
 unsigned long     conv_id;               /* conversation identifier */
} REQUEST_TO_SEND;
typedef struct mc_request_to_send
{
 unsigned short      opcode;              /* verb operation code     */
 unsigned char       opext;               /* verb extension code     */
 unsigned char       format;              /* format                  */
 unsigned short      primary_rc;          /* primary return code     */
 unsigned long       secondary_rc;        /* secondary return code   */
 unsigned char       tp_id[8];            /* TP identifier           */
 unsigned long       conv_id;             /* conversation identifier */
} MC_REQUEST_TO_SEND;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode
AP_B_REQUEST_TO_SEND

AP_M_REQUEST_TO_SEND

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION. For nonblocking operation, this flag can be ORed together with AP_NON_BLOCKING.

format
Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

tp_id Identifier for the local transaction program.

The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

conv_id
Conversation identifier.

The value of this parameter was returned by the **[MC_]ALLOCATE** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameter:

primary_rc
AP_OK

If the verb is nonblocking and has not completed, Communications Server returns the following parameters:

primary_rc
AP_OPERATION_INCOMPLETE

opext AP_OPERATION_INCOMPLETE_FLAG

If **[MC_]REQUEST_TO_SEND** is issued in nonblocking mode (see “Queue-Level Nonblocking” on page 39), and the conversation ends while processing a verb on the send/receive queue, Communications Server returns the following parameter:

primary_rc
AP_CONVERSATION_ENDED

The application should not issue any more verbs for this conversation.

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_CONV_ID

AP_BAD_TP_ID
AP_R_T_S_INVALID_FOR_FDX

If the conversation is in the wrong state when the transaction program issues this verb, Communications Server returns the following parameters:

primary_rc
AP_STATE_CHECK

secondary_rc
AP_R_T_S_BAD_STATE

The conditions generating the following possible primary return codes (**primary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_TP_BUSY
AP_CONVERSATION_TYPE_MIXED
AP_UNEXPECTED_SYSTEM_ERROR
AP_CANCELLED

[MC_]SEND_CONVERSATION

The **[MC_]SEND_CONVERSATION** verb allocates a conversation to a session between the local LU and partner LU (causing a transaction program on the partner LU to start), sends a single data record on this conversation, then deallocates the conversation without waiting for confirmation. It is equivalent to an **[MC_]ALLOCATE**, **[MC_]SEND_DATA**, **[MC_]DEALLOCATE** (**FLUSH**) sequence of verbs (commonly termed a "single one-way bracket").

VCB Structure

```
typedef struct send_conversation
{
 unsigned short   opcode;               /* verb operation code          */
 unsigned char    opext;                /* verb extension code          */
 unsigned char    format;               /* format                       */
 unsigned short   primary_rc            /* primary return code          */
 unsigned long    secondary_rc;         /* secondary return code        */
 unsigned char    tp_id[8];             /* TP identifier                */
 unsigned char    reserv3[8];           /* reserved                     */
 unsigned char    rtn_ctl;              /* return control               */
 unsigned char    reserv4;              /* reserved                     */
 unsigned long    conv_group_id;        /* conversation group identifier */
 unsigned long    sense_data;           /* sense data                   */
 unsigned char    plu_alias[8];         /* partner LU alias             */
 unsigned char    mode_name[8];         /* mode name                    */
 unsigned char    tp_name[64];          /* TP name                      */
 unsigned char    security;             /* security                     */
 unsigned char    reserv5[11];          /* reserved                     */
 unsigned char    pwd[10];              /* security password            */
 unsigned char    user_id[10];          /* security user_id             */
 unsigned short   pip_dlen;             /* PIP data length              */
 unsigned char   *pip_dptr;             /* pointer to PIP data          */
 unsigned char    reserv5a;             /* reserved                     */
 unsigned char    fqplu_name[17];       /* fully qualified partner LU   */
                                        /* name                         */
 unsigned char    reserv6[8];           /* reserved                     */
 unsigned short   dlen;                 /* data length                  */
 unsigned char   *dptr;                 /* pointer to data buffer       */
} SEND_CONVERSATION;
```

```
typedef struct mc_send_conversation
{
 unsigned short    opcode;               /* verb operation code          */
 unsigned char     opext;                /* verb extension code          */
 unsigned char     format;               /* format                       */
 unsigned short    primary_rc;           /* primary return code          */
 unsigned long     secondary_rc;         /* secondary return code        */
 unsigned char     tp_id[8];             /* TP identifier                */
 unsigned char     reserv3[8];           /* reserved                     */
 unsigned char     rtn_ctl;              /* return control               */
 unsigned char     reserv4;              /* reserved                     */
 unsigned long     conv_group_id;        /* conversation group identifier */
 unsigned long     sense_data;           /* sense data                   */
 unsigned char     plu_alias[8];         /* partner LU alias             */
 unsigned char     mode_name[8];         /* mode name                    */
 unsigned char     tp_name[64];          /* TP name                      */
 unsigned char     security;             /* security                     */
 unsigned char     reserv6[11];          /* reserved                     */
 unsigned char     pwd[10];              /* security password            */
 unsigned char     user_id[10];          /* security user_id             */
 unsigned short    pip_dlen;             /* PIP data length              */
 unsigned char    *pip_dptr;             /* pointer to PIP data          */
 unsigned char     reserv6a;             /* reserved                     */
 unsigned char     fqplu_name[17];       /* fully qualified partner LU   */
```

```
                                                  /* name                      */
 unsigned char     reserv7[8];                    /* reserved                  */
 unsigned short    dlen;                          /* data length               */
 unsigned char    *dptr;                          /* pointer to data buffer    */
} MC_SEND_CONVERSATION;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode
AP_B_SEND_CONVERSATION

AP_M_SEND_CONVERSATION

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION. For nonblocking operation, this flag can be ORed together with AP_NON_BLOCKING.

format
Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

tp_id Identifier for the local transaction program.

The value of this parameter was returned by the **TP_STARTED** verb for an invoking transaction program, or by the **RECEIVE_ALLOCATE** verb for an invoked transaction program.

rtn_ctl
Specifies when the local LU acting on a session request from the local transaction processor is to return control to the local transaction program.

AP_IMMEDIATE
AP_WHEN_SESSION_ALLOCATED
AP_WHEN_SESSION_FREE
AP_WHEN_CONV_GROUP_ALLOC
AP_WHEN_CONWINNER_ALLOC
AP_WHEN_CONLOSER_ALLOC

conv_group_id
The conversation group identifier for the session to be allocated. This parameter is only supplied if **rtn_ctl** is set to AP_WHEN_CONV_GROUP_ALLOC.

plu_alias
Alias by which the partner LU is known to the local transaction program. This is an 8-byte string in a locally displayable character set. All 8 bytes are significant and must be set. This name must match the name of a partner LU established during configuration.

If this field is set to all zeros, Communications Server uses the **fqplu_name** field to specify the required partner LU.

mode_name
Name of a set of networking characteristics defined during configuration. This is an 8-byte alphanumeric type-A EBCDIC string (starting with a letter), padded to the right with EBCDIC spaces.

tp_name
Name of the invoked transaction program. Communications Server does not check the character set of this field. The value of **tp_name** specified by the **ALLOCATE** verb in the invoking transaction program must match the value of **tp_name** specified by the **RECEIVE_ALLOCATE** verb in the invoked transaction program.

security
Specifies the information the partner LU requires in order to validate access to the invoked transaction program.

AP_NONE

AP_PGM

AP_SAME

AP_PGM_STRONG

pwd Password associated with **user_id**. This is a 10-byte type-AE EBCDIC character string, padded to the right with EBCDIC spaces. This is required if Security=Program (AP_PGM or AP_PGM_STRONG); otherwise, it is optional.

user_id
User ID required to access the partner transaction program. This is a 10-byte type-AE EBCDIC character string, padded to the right with EBCDIC spaces. This is required if Security=Program (AP_PGM or AP_PGM_STRONG); otherwise, it is optional.

pip_dlen
Length of the program initialization parameters (PIP) to be passed to the partner transaction program.

Range: 0–32767

pip_dptr
Address of buffer containing PIP data. Use this parameter only if **pip_dlen** is greater than zero.

fqplu_name
The fully qualified LU name for the partner LU. This name is 17 bytes long and is right-padded with EBCDIC blanks. It is composed of two type-A EBCDIC character strings concatenated by an EBCDIC dot. (Each name can have a maximum length of 8 bytes with no embedded blanks. If the network ID is not present, then omit the dot.) This field is only significant if the **plu_alias** field is set to all zeros.

dlen Number of bytes of data to send.

Range: 0–65535

dptr Address of the buffer containing the data to send. The application can append data to the end of the VCB, in which case **dptr** must be set to NULL.

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameters:

primary_rc
AP_OK

conv_group_id
The conversation group identifier of the session allocated to the conversation.

If the verb is nonblocking and has not completed, Communications Server returns the following parameter:

primary_rc
AP_OPERATION_INCOMPLETE

opext AP_OPERATION_INCOMPLETE_FLAG

If the **rtn_ctl** parameter was set to AP_IMMEDIATE, and no session is available immediately, Communications Server returns the following parameters:

primary_rc
AP_UNSUCCESSFUL

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_TP_ID
AP_BAD_LL
AP_BAD_RETURN_CONTROL
AP_BAD_SECURITY
AP_PIP_LEN_INCORRECT
AP_NO_USE_OF_SNASVCMG
AP_UNKNOWN_PARTNER_MODE

The conditions generating the following possible primary return codes (**primary_rc**) and indented secondary return codes (**secondary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_UNSUCCESSFUL
AP_ALLOCATION_ERROR
AP_ALLOCATION_FAILURE_NO_RETRY
AP_ALLOCATION_FAILURE_RETRY
AP_SEC_REQUESTED_NOT_SUPPORTED
AP_TP_BUSY
AP_CONVERSATION_TYPE_MIXED
AP_UNEXPECTED_SYSTEM_ERROR
AP_CANCELLED

If the **primary_rc** is set to AP_ALLOCATION_ERROR, the **sense_data** field carries more information on the failure.

[MC_]SEND_DATA

The **[MC_]SEND_DATA** verb puts data in the local LU's send buffer for transmission to the partner transaction program.

VCB Structure

```
typedef struct send_data
{
 unsigned short    opcode;                /* verb operation code      */
 unsigned char     opext;                 /* verb extension code      */
 unsigned char     format;                /* format                   */
 unsigned short    primary_rc;            /* primary return code      */
 unsigned long     secondary_rc;          /* secondary return code    */
 unsigned char     tp_id[8];              /* TP identifier            */
 unsigned long     conv_id;               /* conversation identifier  */
 unsigned char     rts_rcvd;              /* request to send received */
 unsigned char     expd_data_rcvd;        /* expedited data received  */
 unsigned short    dlen;                  /* data length              */
 unsigned char    *dptr;                  /* pointer to data          */
 unsigned char     type;                  /* send data type           */
 unsigned char     reserv4;               /* reserved                 */
} SEND_DATA;
```

```
typedef struct mc_send_data
{
 unsigned short    opcode;             /* verb operation code      */
 unsigned char     opext;              /* verb extension code      */
 unsigned char     format;             /* format                   */
 unsigned short    primary_rc;         /* primary return code      */
 unsigned long     secondary_rc;       /* secondary return code    */
 unsigned char     tp_id[8];           /* TP identifier            */
 unsigned long     conv_id;            /* conversation identifier  */
 unsigned char     rts_rcvd;           /* request to send received */
#ifdef WINAPPC_FORMAT_1
 unsigned char     expd_data_rcvd;     /* expedited data received  */
#else
 unsigned char     data_type;          /* data type received       */
#endif
 unsigned short    dlen;               /* data length              */
 unsigned char    *dptr;               /* pointer to data          */
 unsigned char     type;               /* send data type           */
#ifdef WINAPPC_FORMAT_1
 unsigned char     data_type;          /* data type received       */
#else
 unsigned char     reserv4;            /* reserved                 */
#endif
} MC_SEND_DATA;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode

AP_B_SEND_DATA

AP_M_SEND_DATA

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION. For nonblocking operation, this flag can be ORed together with AP_NON_BLOCKING.

On full-duplex conversations, this flag must be ORed together with AP_FULL_DUPLEX_CONVERSATION.

format
Format of the VCB. Set this to one to get the format listed above.

tp_id Identifier for the local transaction program.

The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

conv_id
Conversation identifier.

The value of this parameter was returned by the **[MC_]ALLOCATE** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

dlen Number of bytes of data to be put in the local LU's send buffer.

Range: 0–65535

dptr Address of the buffer containing the data to be put in the local LU's send buffer. The application can append data to the end of the VCB, in which case **dptr** must be set to NULL.

type Specifies whether to perform the function of another verb in addition to **SEND_DATA**.

AP_NONE
AP_SEND_DATA_CONFIRM
AP_SEND_DATA_FLUSH
AP_SEND_DATA_P_TO_R_FLUSH
AP_SEND_DATA_P_TO_R_SYNC_LEVEL
AP_SEND_DATA_P_TO_R_CONFIRM
AP_SEND_DATA_DEALLOC_FLUSH
AP_SEND_DATA_DEALLOC_SYNC_LEVE
AP_SEND_DATA_DEALLOC_CONFIRM
AP_SEND_DATA_DEALLOC_ABEND

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameters:

primary_rc
AP_OK

rts_rcvd
Request-to-send-received indicator.

AP_YES

AP_NO

expd_data_rcvd
Expedited-data-received indicator. This indication continues to be set to AP_YES until a RECEIVE_EXPEDITED_DATA is issued.

AP_YES

AP_NO

If the verb does not execute due to a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

opext AP_OPERATION_INCOMPLETE_FLAG

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_CONV_ID

AP_BAD_TP_ID

AP_BAD_LL

AP_SEND_DATA_INVALID_TYPE

AP_SEND_DATA_CONFIRM_SYNC_NONE

AP_SEND_TYPE_INVALID_FOR_FDX

If the conversation is in the wrong state when the transaction program issues this verb, Communications Server returns the following parameters:

primary_rc
AP_STATE_CHECK

secondary_rc
AP_SEND_DATA_NOT_SEND_STATE

AP_SEND_DATA_NOT_LL_BDY

The conditions generating the following possible primary return codes (**primary_rc**) and indented secondary return codes (**secondary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_ALLOCATION_ERROR

AP_SECURITY_NOT_VALID

AP_TRANS_PGM_NOT_AVAIL_RETRY

AP_TRANS_PGM_NOT_AVAIL_NO_RTRY

AP_TP_NAME_NOT_RECOGNIZED

AP_PIP_NOT_ALLOWED

AP_PIP_NOT_SPECIFIED_CORRECTLY

AP_CONVERSATION_TYPE_MISMATCH

AP_SYNC_LEVEL_NOT_SUPPORTED

AP_CONV_FAILURE_NO_RETRY

AP_CONV_FAILURE_RETRY

AP_DEALLOC_ABEND

AP_DEALLOC_ABEND_PROG

AP_DEALLOC_ABEND_SVC

AP_DEALLOC_ABEND_TIMER

AP_PROG_ERROR_PURGING
AP_SVC_ERROR_PURGING

AP_TP_BUSY
AP_CONVERSATION_TYPE_MIXED
AP_DUPLEX_TYPE_MIXED

AP_UNEXPECTED_SYSTEM_ERROR
AP_CANCELLED
AP_ERROR_INDICATION
AP_ALLOCATION_ERROR_PENDING
AP_DEALLOC_ABEND_PROG_PENDING
AP_DEALLOC_ABEND_SVC_PENDING
AP_DEALLOC_ABEND_TIMER_PENDING
AP_UNKNOWN_ERROR_TYPE_PENDING

[MC_]SEND_ERROR

The **[MC_]SEND_ERROR** verb notifies the partner transaction program that the local transaction program has encountered an application-level error.

VCB Structure

```
typedef struct send_error
{
 unsigned short     opcode;              /* verb operation code     */
 unsigned char      opext;               /* verb extension code     */
 unsigned char      format;              /* format                  */
 unsigned short     primary_rc;          /* primary return code     */
 unsigned long      secondary_rc;        /* secondary return code   */
 unsigned char      tp_id[8];            /* TP identifier           */
 unsigned long      conv_id;             /* conversation identifier */
 unsigned char      rts_rcvd;            /* request to send received */
 unsigned char      err_type;            /* error type              */
 unsigned char      err_dir;             /* error direction         */
 unsigned char      expd_data_rcvd;      /* expedited data received */
 unsigned short     log_dlen;            /* log data length         */
 unsigned char     *log_dptr;            /* pointer to log data     */
} SEND_ERROR;
```

```
typedef struct mc_send_error
{
 unsigned short   opcode;              /* verb operation code     */
 unsigned char    opext;               /* verb extension code     */
 unsigned char    format;              /* format                  */
 unsigned short   primary_rc;          /* primary return code     */
 unsigned long    secondary_rc;        /* secondary return code   */
 unsigned char    tp_id[8];            /* TP identifier           */
 unsigned long    conv_id;             /* conversation identifier */
 unsigned char    rts_rcvd;            /* request to send received */
 unsigned char    err_type;            /* error type              */
 unsigned char    err_dir;             /* error direction         */
 unsigned char    expd_data_rcvd;      /* expedited data received */
 unsigned char    reserv5[2];          /* reserved                */
 unsigned char    reserv6[4];          /* reserved                */
} MC_SEND_ERROR;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode

AP_B_SEND_ERROR

AP_M_SEND_ERROR

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION. For nonblocking operation, this flag can be ORed together with AP_NON_BLOCKING.

On full-duplex conversations, this flag must be ORed together with AP_FULL_DUPLEX_CONVERSATION.

format

Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

tp_id
Identifier for the local transaction program.

The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

conv_id
Conversation identifier.

The value of this parameter was returned by the **[MC_]ALLOCATE** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

err_type

Indicates the type of the error being reported:application program or service program.

AP_PROG
AP_SVC

err_dir
Indicates whether the error being reported is in the data received from the partner transaction program, or in the data the local transaction program was about to send.

This parameter is used only when the **SEND_ERROR** verb is being issued in SEND_PENDING state.

AP_RCV_DIR_ERROR

AP_SEND_DIR_ERROR

log_dlen

Number of bytes of data to be sent to the error log file.

Range: 0–32767

The application can append data to the end of the VCB, in which case this field will be greater than zero and **log_dptr** must be set to NULL. (A length of zero indicates that there is no error log data.)

log_dptr

Address of data buffer containing error information. The application can append data to the end of the VCB, in which case **log_dptr** must be set to NULL.

This data is sent to the local error log and to the partner LU. This parameter is used by the **SEND_ERROR** verb if **log_dlen** is greater than zero.

The transaction program must format the error data as a General Data Stream (GDS) error log variable. For further information, see *IBM Systems Network Architecture: LU 6.2 Reference: Peer Protocols*

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameters:

primary_rc
AP_OK

rts_rcvd
Request-to-send-received indicator.

AP_YES

AP_NO

expd_data_rcvd
Expedited-data-received indicator. This indication continues to be set to AP_YES until a RECEIVE_EXPEDITED_DATA is issued.

AP_YES

AP_NO

If the verb is nonblocking and has not completed, Communications Server returns the following parameters:

primary_rc
AP_OPERATION_INCOMPLETE

opext AP_OPERATION_INCOMPLETE_FLAG

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_CONV_ID

AP_BAD_ERROR_DIRECTION

AP_BAD_TP_ID

AP_SEND_ERROR_BAD_TYPE

AP_SEND_ERROR_LOG_LL_WRONG

If the conversation is in the wrong state when the transaction program issues this verb, Communications Server returns the following parameters:

primary_rc
AP_STATE_CHECK

secondary_rc
AP_SEND_ERROR_BAD_STATE

The conditions generating the following possible primary return codes (**primary_rc**) and indented secondary return codes (**secondary_rc**) are described in Appendix A. APPC Common Return Codes.

Verb Issued in Any Permitted State

The following return codes can be generated when the **[MC_]SEND_ERROR** verb is issued in any permitted state:

AP_CONV_FAILURE_NO_RETRY
AP_CONV_FAILURE_RETRY

AP_TP_BUSY
AP_CONVERSATION_TYPE_MIXED
AP_DUPLEX_TYPE_MIXED
AP_UNEXPECTED_SYSTEM_ERROR
AP_CANCELLED
AP_ERROR_INDICATION

- AP_ALLOCATION_ERROR_PENDING
- AP_DEALLOC_ABEND_PROG_PENDING
- AP_DEALLOC_ABEND_SVC_PENDING
- AP_DEALLOC_ABEND_TIMER_PENDING
- AP_UNKNOWN_ERROR_TYPE_PENDING

Verb Issued in SEND State: The following return codes can be generated only if the **[MC_]SEND_ERROR** verb is issued in SEND state:

AP_ALLOCATION_ERROR

- AP_SECURITY_NOT_VALID
- AP_TRANS_PGM_NOT_AVAIL_RETRY
- AP_TRANS_PGM_NOT_AVAIL_NO_RTRY
- AP_TP_NAME_NOT_RECOGNIZED
- AP_PIP_NOT_ALLOWED
- AP_PIP_NOT_SPECIFIED_CORRECTLY
- AP_CONVERSATION_TYPE_MISMATCH
- AP_SYNC_LEVEL_NOT_SUPPORTED
- AP_DEALLOC_ABEND
- AP_DEALLOC_ABEND_PROG
- AP_DEALLOC_ABEND_SVC
- AP_DEALLOC_ABEND_TIMER
- AP_PROG_ERROR_PURGING
- AP_SVC_ERROR_PURGING

Verb Issued in RECEIVE State: The following return code can be generated only if the verb is issued in RECEIVE state:

AP_DEALLOC_NORMAL

[MC_]SEND_EXPEDITED_DATA

This is only supported for applications residing on the server.

The **[MC_]SEND_EXPEDITED_DATA** verb puts data in the local LU's expedited send buffer for transmission to the partner transaction program. This data can arrive at the partner transaction program before non-expedited data that was sent earlier.

VCB Structure

```
typedef struct send_expedited_data
{
 unsigned short     opcode;                /* verb operation code      */
 unsigned char      opext;                 /* verb extension code      */
 unsigned char      format;                /* format                   */
 unsigned short     primary_rc;            /* primary return code      */
 unsigned long      secondary_rc;          /* secondary return code    */
 unsigned char      tp_id[8];              /* TP identifier            */
 unsigned long      conv_id;               /* conversation identifier  */
 unsigned char      rts_rcvd;              /* request to send received */
 unsigned char      expd_data_rcvd;        /* expedited data received  */
 unsigned short     dlen;                  /* data length              */
 unsigned char     *dptr;                  /* pointer to data          */
 unsigned char      reserve4[2];           /* TP identifier            */
} SEND_EXPEDITED_DATA;
```

```
typedef struct mc_send_expedited_data
{
 unsigned short    opcode;               /* verb operation code       */
 unsigned char     opext;                /* verb extension code       */
 unsigned char     format;               /* format                    */
 unsigned short    primary_rc;           /* primary return code       */
 unsigned long     secondary_rc;         /* secondary return code     */
 unsigned char     tp_id[8];             /* TP identifier             */
 unsigned long     conv_id;              /* conversation identifier   */
 unsigned char     rts_rcvd;             /* request to send received  */
 unsigned char     expd_data_rcvd;       /* expedited data received   */
                                         /* transaction plan          */
                                         /* data                      */
 unsigned short    dlen;                 /* actual length of received */
                                         /* data                      */
 unsigned char    *dptr;                 /* pointer to data buffer    */
 unsigned char     reserv4[2];           /* reserved                  */
} MC_SEND_EXPEDITED_DATA
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode

AP_B_SEND_EXPEDITED_DATA

AP_M_SEND_EXPEDITED_DATA

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION. For nonblocking operation, this flag can be ORed together with AP_NON_BLOCKING.

On full-duplex conversations, this flag must be ORed together with AP_FULL_DUPLEX_CONVERSATION.

format
Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

tp_id Identifier for the local transaction program.

The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

conv_id
Conversation identifier.

The value of this parameter was returned by the **[MC_]ALLOCATE** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

dlen Number of bytes of data to be put in the local LU's send buffer.

Range: 1–86

dptr Address of data buffer containing error information. The application can append data to the end of the VCB, in which case **dptr** must be set to NULL.

Note that the data is unformatted--no 2-byte length field (LL) is present.

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameters:

primary_rc
AP_OK

rts_rcvd
Request-to-send-received indicator.

AP_YES
AP_NO

expd_data_rcvd
Expedited-data-received indicator. This indication continues to be set to AP_YES until a RECEIVE_EXPEDITED_DATA is issued.

AP_YES
AP_NO

If the verb is nonblocking and has not completed, Communications Server returns the following parameters:

primary_rc
AP_OPERATION_INCOMPLETE

opext AP_OPERATION_INCOMPLETE_FLAG

If the verb does not execute because the remote LU does not support expedited data, Communications Server returns the following parameter:

primary_rc
AP_EXPD_NOT_SUPPORTED_BY_LU

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
: AP_PARAMETER_CHECK

secondary_rc
: AP_BAD_CONV_ID
  AP_BAD_TP_ID
  AP_SEND_EXPD_INVALID_LENGTH
  AP_RCV_EXPD_INVALID_LENGTH

If the conversation is in the wrong state when the transaction program issues this verb, Communications Server returns the following parameters:

primary_rc
: AP_STATE_CHECK

secondary_rc
: AP_EXPD_DATA_BAD_CONV_STATE

The conditions generating the following possible primary return codes (**primary_rc**) and indented secondary return codes (**secondary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_ALLOCATION_ERROR
- AP_SECURITY_NOT_VALID
- AP_TRANS_PGM_NOT_AVAIL_RETRY
- AP_TRANS_PGM_NOT_AVAIL_NO_RTRY
- AP_TP_NAME_NOT_RECOGNIZED
- AP_PIP_NOT_ALLOWED
- AP_PIP_NOT_SPECIFIED_CORRECTLY
- AP_CONVERSATION_TYPE_MISMATCH
- AP_SYNC_LEVEL_NOT_SUPPORTED

AP_CONV_FAILURE_NO_RETRY
AP_CONV_FAILURE_RETRY
AP_DEALLOC_ABEND_PROG
AP_DEALLOC_ABEND_SVC
AP_DEALLOC_ABEND_TIMER
AP_TP_BUSY
AP_CONVERSATION_TYPE_MIXED
AP_DUPLEX_TYPE_MIXED
AP_UNEXPECTED_SYSTEM_ERROR
AP_CANCELLED

[MC_]TEST_RTS

The **[MC_]TEST_RTS** verb determines whether a request-to-send notification has been received from the partner transaction program.

VCB Structure

```
typedef struct test_rts
{
 unsigned short     opcode;              /* verb operation code     */
 unsigned char      opext;               /* verb extension code     */
 unsigned char      format;              /* format                  */
 unsigned short     primary_rc;          /* primary return code     */
 unsigned long      secondary_rc;        /* secondary return code   */
 unsigned char      tp_id[8];            /* TP identifier           */
 unsigned long      conv_id;             /* conversation identifier */
 unsigned char      reserv3;             /* reserved                */
} TEST_RTS;
typedef struct mc_test_rts
{
 unsigned short    opcode;             /* verb operation code     */
 unsigned char     opext;              /* verb extension code     */
 unsigned char     format;             /* format                  */
 unsigned short    primary_rc;         /* primary return code     */
 unsigned long     secondary_rc;       /* secondary return code   */
 unsigned char     tp_id[8];           /* TP identifier           */
 unsigned long     conv_id;            /* conversation identifier */
 unsigned char     reserv3;            /* reserved                */
} MC_TEST_RTS;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode
AP_B_TEST_RTS

AP_M_TEST_RTS

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION

format
Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

tp_id Identifier for the local transaction program. The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

conv_id
Conversation identifier.

The value of this parameter was returned by the **[MC_]ALLOCATE** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

Returned Parameters

If the verb executes successfully, Communications Server returns the following parameter:

primary_rc
: Indicates whether a request-to-send notification has been received from the partner transaction program.

  AP_OK

  AP_UNSUCCESSFUL

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
: AP_PARAMETER_CHECK

secondary_rc
: AP_BAD_CONV_ID

  AP_BAD_TP_ID
  AP_TEST_INVALID_FOR_FDX

The conditions generating the following possible primary return codes (**primary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_TP_BUSY
AP_CONVERSATION_TYPE_MIXED
AP_UNEXPECTED_SYSTEM_ERROR

[MC_]TEST_RTS_AND_POST

The **[MC_]TEST_RTS_AND_POST** verb asynchronously determines whether a request-to-send notification has been received from the partner transaction program. A transaction program can issue a **[MC_]TEST_RTS_AND_POST** at any time, even when there is another verb outstanding on the conversation. **[MC_]TEST_RTS_AND_POST** returns when a request-to-send notification is received, or when the conversation ends, or when a conversation failure is detected.

This verb can only be issued through the APPC entry point.

VCB Structure

```
typedef struct test_rts_and_post
{
 unsigned short      opcode;                /* verb operation code     */
 unsigned char       opext;                 /* verb extension code     */
 unsigned char       format;                /* format                  */
 unsigned short      primary_rc;            /* primary return code     */
 unsigned long       secondary_rc;          /* secondary return code   */
 unsigned char       tp_id[8];              /* TP identifier           */
 unsigned long       conv_id;               /* conversation identifier */
 unsigned char       reserv3;               /* reserved                */
 unsigned long       sema;                  /* post handle for verb    */
} TEST_RTS_AND_POST;
```

```
typedef struct mc_test_rts_and_post
{
 unsigned short    opcode;              /* verb operation code     */
 unsigned char     opext;               /* verb extension code     */
 unsigned char     format;              /* format                  */
 unsigned short    primary_rc;          /* primary return code     */
 unsigned long     secondary_rc;        /* secondary return code   */
 unsigned char     tp_id[8];            /* TP identifier           */
 unsigned long     conv_id;             /* conversation identifier */
 unsigned char     reserv3;             /* reserved                */
 unsigned long     sema;                /* post handle for verb    */
} MC_TEST_RTS_AND_POST;
```

Supplied Parameters

The transaction program supplies the following parameters to Communications Server:

opcode

AP_B_TEST_RTS_AND_POST

AP_M_TEST_RTS_AND_POST

opext AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION

format

Identifies the format of the VCB. Set this field to zero to specify the version of the VCB listed above.

tp_id Identifier for the local transaction program.

The value of this parameter was returned by the **TP_STARTED** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

conv_id
Conversation identifier.

The value of this parameter was returned by the **[MC_]ALLOCATE** verb in the invoking transaction program or by **RECEIVE_ALLOCATE** in the invoked transaction program.

sema Handle of the event that the application will wait on. This verb is intended for use with WaitForMultipleObjects in the Win32 API. For more information about this function, see the programming documentation for the Win32 API.

Returned Parameters

If the verb executes successfully (that is, a request-to-send notification is received), Communications Server return the following parameter:

primary_rc
AP_OK

If the verb returns because the conversation has ended or a conversation failure has been detected, Communications Server returns the following parameter:

primary_rc
AP_UNSUCCESSFUL

If the verb does not execute because of a parameter error, Communications Server returns the following parameters:

primary_rc
AP_PARAMETER_CHECK

secondary_rc
AP_BAD_CONV_ID

AP_BAD_TP_ID
AP_TEST_INVALID_FOR_FDX

The conditions generating the following possible primary return codes (**primary_rc**) are described in Appendix A. APPC Common Return Codes.

AP_CONVERSATION_TYPE_MIXED
AP_UNEXPECTED_SYSTEM_ERROR
AP_CANCELLED

Part 2. LUA API

© Copyright IBM Corp. 1994, 1997

Chapter 9. Fundamental Concepts of the IBM Conventional LU Application

This chapter describes the IBM conventional logical unit application (LUA) access method and describes its relationship to Systems Network Architecture (SNA).

Understanding LUA and SNA

The IBM LUA access method provides Communications Server with an application programming interface (API) for secondary dependent logical units (LUs). LUA consists of system software and interfaces that supply input/output (I/O) service routines to support communications using LU types 0, 1, 2, and 3 SNA protocols. Communications Server supports the RUI and SLI interface of LUA.

The Communications Server implementation is designed to be binary compatible with Microsoft** NT SNA Server and similar to the implementation of OS/2 Communications Manager/2 Version 1.0 LUA.

The services that LUA provides to application programs include only those that support data communications; LUA does not provide any device emulation facilities. However, LUA does provide a unique subset of presentation services layer functions.

Communications Server must be installed and configured before an LUA application program can run on the workstation. Refer to *Communications Server: Quick Beginnings* for information about installing and configuring Communications Server.

Connection Capabilities

The main objective of any communications system is to connect with other systems. The goal of SNA is to supply common protocols that give universal connectivity. LUA communication and connectivity requirements include the System/370* (S/370*) connections.

LUA Application Programs

In this book, the term *LUA application program* means an application program, or a portion of an application program, that uses LUA communications functions. Application programs use these functions to communicate with application programs on other systems that support LU types 0, 1, 2, or 3.

As a local LUA application program runs, it exchanges data with a remote host application program. The local and remote application programs are called *partner* application programs.

LUA Verbs

A verb is a formatted request that is processed by LUA. An application program issues a verb to request that LUA take some action. LUA verbs are coded as control blocks. Each verb control block has a precisely defined format. To use the LUA facilities, an application program supplies verb control blocks to the LUA API.

© Copyright IBM Corp. 1994, 1997

An LUA verb always returns immediately to its caller. If the return code is IN_PROGRESS, the application needs to wait for completion of the verb, using the posting method specified in the verb request. See Chapter 12. RUI Entry Points and Chapter 14. SLI Entry Points for a description of LUA verb postings.

Verb control block layouts are available in the Communications Server **INCLUDE** directory. You can use the verb control block layouts and sample programs to help you write LUA application programs.

LUs, Local LUs, and Partner LUs

A logical unit (LU) manages the exchange of data between application programs. Every LUA application program gains access to an SNA network through an LU, which acts as an intermediary between the LUA application program and the SNA network.

In LUA, there is a one-to-many relationship between LUA application program processes and LUs. One LUA application program process can own multiple LUs simultaneously, but a given LU can be owned by only one LUA application program process simultaneously. Before a second application program process can use an LU, the first application program must release the LU.

An LUA application program issues LUA verbs to its local LU. These verbs cause commands and data to flow across the network to the partner LU.

Note: You need to define your local LU only once for each machine, as described in *Quick Beginnings* .

System Services Control Point (SSCP)

The system services control point (SSCP) component in a host system is responsible for starting host applications, for associating host applications with dependent LUs, and for creating and terminating the connections between LUs.

SNA Layers

SNA is a hierarchical structure that consists of seven well-defined layers. Each layer in the architecture performs a specific function. Understanding the layered structure of SNA helps in understanding the various functions that LUA supplies. The following descriptions of the five highest-level SNA layers show the relationship between LUA and SNA.

Data Link Control Layer

The data link control (DLC) layer consists of the elements that provide an interface to the hardware. The DLC elements supply support for various DLC protocols, such as Synchronous Data Link Control (SDLC) and the IBM Token-Ring Network. The DLC layer supplies a common link appearance to the elements in the path control (PC) layer. The DLC layer is common to all Communications Server LU implementations, including LUA.

Path Control Layer

The path control (PC) layer of SNA in a peripheral node supplies basic functions, such as routing to and from multiple half-sessions within its node. SNA permits the PC layer to route to and from only one data link at a time. The PC layer is common to all Communications Server LU implementations, including LUA.

Transmission Control Layer

The transmission control (TC) layer of SNA supplies the connection-point-manager function and the session-control function for each locally supported half-session. The connection-point-manager function controls sequence-number checking, pacing, and other support functions that relate to half-session data flows. The session-control function supplies session-specific support for starting, pacing, enciphering, deciphering, and other support functions that relate to session-related data flows. LUA contains an implementation of the TC layer for LU types 0, 1, 2, and 3 within Communications Server.

Data Flow Control Layer

The data flow control (DFC) layer of SNA controls the flow of function management data (FMD) requests and FMD responses between FMD pairs that are in sessions and between sessions. The data flow control layer supplies various functions, such as request/response formatting, data-chaining protocol, request/response correlation, send- and receive-mode protocols, bracket protocol, error-recovery protocol, stop-bracket-initiation protocol, and queued-response protocol. LUA contains an implementation of the data flow control layer for LU types 0, 1, 2, and 3 within Communications Server.

Presentation Services Layer

The presentation services (PS) layer of SNA contains the function that presents the communications data interface to the user. The presentation services layer is defined in the architecture for all LU types except LU 0. LUA contains a unique subset of the presentation services layer within Communications Server. For more information about the presentation services layer, refer to *Systems Network Architecture Concepts and Products*

The LU services functions are a part of the SNA-session message flow layers. These functions supply support before session establishment, build session structures, and take down session structures. LUA functions interface with common Communications Server support to define LUs and to start and stop SNA sessions.

Using SNA Sessions

Before an LUA application program can communicate with a partner host application program, the respective LUs must be connected in a mutual relationship called a session. An SNA session is a logical connection that enables two network addressable units (NAUs) to communicate with each other; an LU is one kind of NAU. Because the session connects two LUs, it is called an *LU-LU session*. LU-LU sessions enable end users to exchange data.

A session manages how data moves between a pair of LUs in an SNA network. Therefore, sessions are concerned with such things as the quantity of data being transmitted, data security, network routing, data loss, and traffic congestion. Session characteristics are determined by the SNA **BIND** command originating from the primary LU, when the secondary LU accepts the **BIND** command.

Prerequisites to an SNA Session

An LU-LU session consists of communication between a primary logical unit (PLU) and a secondary logical unit (SLU). The SLU is implemented by the LUA application program. Before data can be transferred between a PLU and an SLU on an LU-LU session, the following events must occur:

1. Communications Server activates the data link.
2. When the data link is ready, the system services control point (SSCP) establishes a session between itself and a physical unit (SSCP-PU session) by sending an Activate Physical Unit (**ACTPU**) command and reading a positive response from Communications Server. Communications Server sends a positive response if the PU address from the **ACTPU** command corresponds to the configuration information.
3. The SSCP establishes a session between itself and the logical unit (SSCP-LU session) by sending an Activate Logical Unit (**ACTLU**) command and reading a positive response from Communications Server. Communications Server sends a positive response if the LU address from the **ACTLU** command corresponds to the configuration information.

Starting Sessions

Either the SLU or the PLU can start an LU-LU session.

Starting an LU-LU Session from an SLU

After the SSCP-LU session is established, the SLU program can request an LU-LU session by sending the Initiate Self (**INITSELF**) command to the SSCP. The SSCP receives the **INITSELF** command and checks whether the named host application program is valid. A host application program is valid if it is known and active. If the host application program is valid, the SSCP sends a positive response to the SLU, and the PLU starts the session. If the host application program is not valid, the SSCP sends a negative response to the SLU, and the PLU does not start the session.

If the SSCP sends a positive response to an **INITSELF** command but the session cannot be established, the SSCP sends a Network Services Procedure Error (**NSPE**) command to the SLU to stop the attempt to establish a session. The SLU can reissue the **INITSELF** command after the **NSPE** command.

Starting an LU-LU Session from a PLU

The PLU program can start unsolicited LU-LU sessions. The PLU starts sessions by generating a **BIND** command. A subsequent positive response establishes the agreement to communicate. A data field that is associated with the **BIND** command contains the name of the PLU application program and the session **BIND** parameters. For more information about the format of this data field, refer to *Systems Network Architecture: Formats*.

For nonnegotiable **BIND**s, the SLU returns a positive response if the parameters are acceptable. If the parameters are unacceptable, the SLU returns a negative response with sense data to the PLU.

The negotiable **BIND** command permits the SLU to return a positive response with a minimum of 26 bytes of updated session parameters indicating compatibility with the PLU parameters. If the PLU finds the returned parameters acceptable, it sends a Start Data Traffic (**SDT**) command. If the returned parameters are unacceptable, the PLU sends an **UNBIND** command that indicates unacceptable negotiable **BIND** command parameters from the SLU.

Transferring Data on an LU-LU Session

After the LU-LU session is established and the SLU program responds to the **SDT** command, data transfer can begin. For a data transmission operation, a message moves from end-user storage to Communications Server storage until it is transmitted. For a data-reception operation, Communications Server places a message in its own storage and then moves the message into end-user storage.

Quiesce protocols suspend the transfer of data in an LU-LU session. The PLU or the SLU can send the following Quiesce protocol commands:

- Quiesce at End of Chain (**QEC**). This command requests that the receiver of this command stop sending data after sending the last part in a data chain. A data chain is a series of related messages. For more information about data chaining, see “Using the Data-Chaining Protocol” on page 159.
- Quiesce Complete (**QC**). This command notifies a **QEC** command that data transfer is suspended. When the SLU sends the **QC** command, Communications Server prevents the SLU from sending any normal-flow messages until the Release Quiesce (**RELQ**) command is received.
- Release Quiesce (**RELQ**). This command notifies the receiver that data can again be transferred.

Stopping Sessions

When all data has been transferred and verified, the session can end. An SLU must end one session before it can begin a different session with either the same or another PLU.

Stopping an LU-LU Session by an SLU

An SLU can end an LU-LU session in either of two ways:

- By sending a Terminate-Self (**TERMSELF**) command or an **UNBIND** command. Either command results in an immediate ending.
- By sending a Request Shutdown (**RSHUTD**) command. This command solicits an **UNBIND** from the PLU.

To end a session immediately, the SLU sends the **TERMSELF** command to the SSCP, which checks whether the named LUA application program is the one participating in this session. If it is, the SSCP sends a positive, non-data response. Depending on the host SNA version being used, the SSCP can send a **CLEAR** command, which purges all messages from the LU-LU session, and can then send an **UNBIND** command to end the session. Alternatively, the SLU can send an **UNBIND** command to the PLU.

Stopping an LU-LU Session by a PLU

A PLU can end an LU-LU session in either of two ways:

- By sending a **CLEAR** command and then an **UNBIND** command, or an **UNBIND** command only. Either method results in an immediate ending.
- By sending the Shutdown (**SHUTD**) command. This command results in an orderly session termination. The SLU and PLU have a dialog that tells each to stop sending data and that ensures that data already sent is received.

Ending the LU-LU session has no effect on the SSCP-LU session.

Stopping an SSCP-LU Session and an SSCP-PU Session

The SSCP-LU session ends when the host sends the Deactivate Logical Unit (**DACTLU**) command to the SLU. When the last SSCP-LU session for Communications Server ends, the SSCP can end the SSCP-PU session by sending a Deactivate Physical Unit (**DACTPU**) command.

Disconnecting the Host Link

When the host receives the response to the **DACTPU** command, it returns a command to Communications Server such as the Set Disconnect Response Mode (**SDRM**) command when using SDLC protocol. The SSCP can also disconnect immediately at any time by sending the same command to Communications Server, which ends all sessions. When sessions are ended in this manner, all SLUs that were active earlier receive a loss-of-contact indication.

Message Numbers

All normal-flow messages that are transmitted between the SLU and the PLU during an LU-LU session are numbered in sequence. The SLU maintains a sequence number for normal-flow messages from the SLU to the PLU and another sequence number for normal-flow messages from the PLU to the SLU. Each normal-flow message gets a sequence number one greater than the sequence number of the preceding normal-flow message. There is one pair of sequence numbers for each session that is established between an SLU and a PLU.

For LU-LU expedited-flow messages and for all SSCP-LU and SSCP-PU messages, unsequenced identifiers are used instead of sequence numbers.

When a session is reestablished or a **CLEAR** command is sent, the PLU and the SLU set their sequence numbers to 0. The PLU can change the sequence numbers with the Set and Test Sequence Numbers (**STSN**) command. This enables correct sequence numbering when a session is recovered or restarted.

When the SLU encounters a sequence number error, it sends a negative response to the PLU if a response was requested. When the SLU reads a response, the SLU uses the response sequence number to correlate the response with the original request. When the SLU writes a response, the SLU must supply the sequence number of the original request.

Restarting and Resynchronizing a Session

If the PLU or the SLU encounters an irrecoverable error, such as a line failure, you might need to resynchronize the LU-LU session after restarting it. Resynchronizing the LU-LU session includes reprocessing recoverable messages and (optionally) resetting the message sequence numbers. The application programs can include routines to retransmit lost messages.

When a session is restarted and resynchronized, the PLU sends the **BIND**, the **STSN**, and the **SDT** commands. When the **STSN** command is sent, a dialog can occur to establish the sequence numbers that are acceptable to both the PLU and the SLU. This dialog consists of a series of **STSN** messages and positive responses.

If the SLU determines that resynchronization is required, the SLU can send a Request Recovery (**RQR**) command, a negative response, or an LU-Status command (**LUSTAT**) with a description of the failure in the user sense bytes. If the PLU discovers the failure or receives an **RQR** command from the SLU, the PLU sends a **CLEAR** command to purge all LU-LU messages from the network, an **STSN** command to establish new sequence numbers, and then an **SDT** command.

Using Protocols to Control Requests and Responses

Various protocols can control the sequencing rules for requests and responses. This section describes some of the protocols used for managing the SNA network, transferring data, and synchronizing the states of network components.

Using the Pacing Protocol

To avoid a message-flow rate that is too fast for Communications Server or the host, you can specify pacing in the **BIND** command. Pacing applies to the LU-LU normal flow only. While pacing, Communications Server permits a specified number of messages to flow and waits for a response before permitting additional messages to be sent. You can specify pacing on the Communications Server-to-host flow, the host-to-Communications Server flow, or both. Once the LU-LU session starts, LUA handles all pacing with no participation by the application program.

Receive-Pacing Protocol

Receive-pacing protocol gives the PLU control over the number and the frequency of messages sent from the SLU on an LU-LU session. When the SLU receives pacing values in the **BIND** command, Communications Server automatically enforces pacing for each SLU that communicates with the host.

During a positive response to a negotiable **BIND** command, you can change the pacing values to any number except 0. When the SLU sends the first message of a sequence, Communications Server sets a bit in the request/response header (RH) that indicates a pacing response is to be returned. If the pacing count is exhausted before Communications Server receives a pacing response from the PLU, Communications Server cannot send additional data messages. If the application program issues a write operation and no pacing response is received, Communications Server defers the write operation.

Send-Pacing Protocol

The SLU automatically controls the send-pacing protocol. If the pacing indicator is set on in a message from the PLU to the SLU, the SLU issues a pacing response when the application program reads the message. The message response can contain the pacing indicator or, if no response is required for the received message, the pacing response can be an isolated pacing response (IPR). The PLU can then send another pacing window of messages.

Using the Half-Duplex Contention/Flip-Flop Protocol

The change-direction (CD) indicator is used with both of the following protocols:

- Half-duplex contention protocol, which is a normal-flow send/receive mode in which either half-session can send normal-flow requests at the beginning of the session or after sending or receiving the last request of a chain.
- Half-duplex flip-flop protocol, which is a normal-flow send/receive mode in which one half-session sets the CD indicator in the response header (RH) on an end-of-chain to enable the other half-session to begin sending.

A CD indicator tells the receiver that transmitting can begin.

For example, if the SLU initiates a transaction, the SLU begins by sending the messages that completely describe the transaction. On the last message, the SLU sets the CD indicator to tell the PLU that it can begin transmitting a reply. If the PLU needs additional information to complete the transaction, it sends an inquiry and sets the CD indicator. The dialog proceeds in this half-duplex mode until the transaction is complete. During a half-duplex dialog, the SLU can use the **SIG** command to tell the PLU to stop sending data and to change the direction of the data flow.

Using the Bracket Protocol

Bracket protocol gives the SLU and the PLU context control of the data transmission, indicating that a session concerns a single transaction. Bracket protocol protects a current session from interruption by a concurrent transaction. A bracket encompasses the duration of a transaction.

The first message in the bracket contains a begin-bracket (BB) indicator, and the last message in the bracket contains an end-bracket (EB) indicator. A single message can be a bracket if it contains both indicators.

For a bracket session, the **BIND** command specifies one LU as the first speaker, and the other LU as the bidder. The first speaker can begin a bracket without permission from the other LU. The bidder, however, must request and receive permission from the first speaker to begin a bracket.

A **BID** command is a normal-flow request that is issued by the bidder to request permission to begin a bracket. A positive response to a **BID** command indicates that the first speaker will not begin a bracket but will wait for the bidder to begin a bracket. A negative response to a **BID** command indicates that the first speaker denies permission for the bidder to begin a bracket. The first speaker can send a Ready-to-Receive (**RTR**) command when permission is granted to start a bracket.

The first speaker indicates a negative response to a **BID** command with one of two response codes:

Bracket-Bid-Reject-RTR-Forthcoming
Indicates that an **RTR** command for that **BID** command will be sent later (granting permission to start a bracket). The bidder can wait for the **RTR** command or send the **BID** command again.

Bracket-Bid-Reject-No-RTR-Forthcoming
Indicates that no **RTR** command for that **BID** command will be sent later. The bidder must send the **BID** command again if the bidder still wants to begin a bracket.

Instead of sending a **BID** command followed by a first-in-chain FMD with a BB indicator, the bidder can attempt to initiate a bracket by sending a first-in-chain FMD with a BB indicator. The first speaker can grant the attempt with a positive response or it can refuse the attempt with a negative response that indicates either of the negative response codes. However, if the bidder stops the chain that carries the BB indicator by sending the **CANCEL** command, the bracket is not initiated, regardless of the response. The **RTR** command can be issued by the first speaker either to grant permission to the bidder to begin a bracket or to find out if the bidder wants to begin a bracket.

A positive response to an **RTR** command indicates that the bidder will initiate the next bracket. If the bidder does not want to initiate a bracket, the bidder issues a negative response with the RTR-Not-Required sense code.

Using the Data-Chaining Protocol

Data chaining is an optional protocol for transmitting a group of related messages. To send chained messages from the SLU, the SLU sets to 1 the begin-chain (BC) indicator for the message to indicate the first message in a chain. For all messages between the first and the last in the chain, the SLU sets the BC and the end-chain (EC) indicators to 0. For the last message in the chain, the EC is set to 1 again. When the SLU receives messages, it tests the chaining indicator to determine if the messages are chained.

The data-chaining protocol comprises three types of chains, as follows:

- No-response chain. Each request in the chain is marked *no response*.
- Exception-response chain. Each request in the chain is marked *exception response*.
- Definite-response chain. The last request in the chain is marked *definite response*; all other requests in the chain are marked *exception response*.

When sending a message chain to the PLU, the SLU can send a **CANCEL** command if the SLU or the PLU finds a message error. If the SLU sends a **CANCEL** command to the PLU, the PLU discards all messages in the chain that it has received. If the PLU sends a negative response to any element of a chain, the SLU ends the chain normally or sends a **CANCEL** command.

Data Exchange Control Methods

An SNA session is conducted under rules for orderly exchange of data.

Flow Protocols

At the transport level, data is exchanged through either a half-duplex (HDX) protocol or a full-duplex (FDX) protocol.

When a half-duplex protocol is used, data flows in only one direction at a time, with one LU sending only and the other receiving only. In a half-duplex flip-flop protocol, both LUs recognize which LU has the right to send and which to receive. At specified times the partner LUs agree to change the direction so that the receiver can send and the sender can receive.

When a full-duplex protocol is used, data can flow in either direction at any time. Both LUs can send and receive without constraint.

Response Modes

Each SNA message is either a request or a response. Every request from one LU elicits a matching response from the partner LU. Because the response carries the same transmission sequence number as the request, responses and requests can be matched by their sequence numbers.

When your application has received a request whose RH specifies a mandatory response, your application must generate and send a response message. The response mode rule determines when the response must be sent.

Under immediate response mode, your application must send a response to a request before it sends any request of its own. Under delayed response mode, however, responses can be sent at any time after a request is received.

LUA Correlation Tables

LUA keeps track of the sequence numbers of incoming and outgoing requests until they receive responses-until the application issues a response to an incoming request, or until the PLU responds to an outgoing request. These numbers are recorded in the Communications Server in areas called *correlation tables*.

Under immediate response mode, only a few outstanding requests can be generated in a session, typically one at most. Under delayed response mode, the number can be larger.

The LUA correlation tables are managed dynamically. LUA can record any number of responses. If a very large number of responses accumulate (probably due to a program logic error), the server runs low on memory the Communications Server might shut down.

Exception Response Requests (RQEs)

In most cases, LUA can correlate requests and responses automatically, without any help from your program. LUA observes the request and response RUs as they flow in the session. LUA can tell when a request needs a response, and when the response has been sent. However, there is one case in which LUA cannot tell if a response will be sent, and your program must tell it.

Bit fields in the RH of a request specify whether a response is mandatory, not wanted, or optional. When no response is wanted, LUA need not store the request number in its correlation table. A mandatory response must be sent as the next message on that flow. LUA enters the message in the correlation table, but it will quickly be cleared because the response must come next.

The error response indicator (ERI) in the RH specifies that a response is optional, required only if the receiving LU cannot accept or process the RU. This optional-response RU is called an exception response request (abbreviated RQE). LUA cannot always manage its correlation table automatically in the presence of RQEs. Table 12 summarizes the instances in which LUA can clear a received RQE automatically from its correlation table, and those in which LUA must wait for a signal from the application before clearing a received RQE.

Table 12. Clearing of RQEs

Immediate Response Mode	**Delayed Response Mode**			
Verbs	HDX	FDX	HDX	FDX
RUI_READ SLI_RECEIVE	Automatic	Automatic	Application response	Application response
RUI_WRITE SLI_SEND	Automatic	Application response	Application response	Application response

In immediate response mode on either an HDX or FDX session, LUA can discard the number of an RQE as soon as the application requests input (uses RUI_READ or SLI_RECEIVE), because, in immediate response mode, a response must be sent before another request can be issued. Also, in immediate response mode on an HDX connection, LUA can discard the number of an RQE as soon as the application requests output (uses RUI_WRITE or SLI_SEND)—because the output will either be the RQE response, or no response is going to be sent.

In all other instances, LUA cannot be sure whether a response to the RQE will be produced. The application must format and send a positive response to an RQE, not for the benefit of the PLU (which wants only negative responses) but to inform LUA that the RQE was accepted and will not be generating a negative response.

LUA can then clear the RQE from its table. Because the response is a positive one and the PLU wanted only negative ones, LUA does not transmit the application’s response on the network.

In short, simply to assist LUA, your application must treat received RQE RUs as if they were definite-response RUs.

Session Profiles

The specific SNA protocols and conventions that can be used on a given session, taken together, comprise the “profile” of the session. Two profiles, the transmission services (TS) profile and the function management (FM) profile, can be bound to the session. The choice of profiles is made at BIND time.

TS Profiles

Five TS profiles, numbered 1, 2, 3, 4, and 7 are defined by SNA. However, because TS profile 1 is used only between the SSCP and the PU, only profiles 2, 3, 4, and 7 are applicable to an LUA application. They differ in the SNA commands that can be issued, as shown in Table 13.

Table 13. TS Profile Characteristics

Profile	Pacing Use	CLEAR	CRV	RQR	SDT	STSN
2	Always	Used	Not used	Not used	Not used	Not used
3	Always	Used	Optional	Not used	Used	Not used
4	Always	Used	Optional	Used	Used	Used
7	Optional	Not used	Optional	Not used	Not used	Not used

Note: Permissible LU-LU TS profiles for the RUI are 2, 3, 4, and 7. Permissible TS profiles for SLI are 3 and 4.

FM Profiles

Eight FM profiles, numbered 0, 2, 3, 4, 6, 7, 18, and 19 are defined by SNA. However, because profiles 0 and 6 are used only by the SSCP, and profile 19 is used only with LU type 6.2, five profiles can be applicable to an LUA application. Each profile differs in the SNA facilities that are restricted.

An approximate summary of the FM profiles is shown in Table 14. In the table, a blank cell means that the SNA facility is not restricted in this profile—it can have any use that can be specified in the BIND parameters.

The LUA RUI supports FM profiles 2, 3, 4, 7, and 18. The LUA SLI supports FM profiles 3 and 4.

Table 14. FM Profile Characteristics

SNA Facility	FMP 2	FMP 3	FMP 4	FMP 7	FMP 18
Request mode	SLU uses delayed				
Response mode	SLU uses immediate	Immediate	Immediate	Immediate	Immediate
RU chains	Single RU chains only				
Length-checked compression				LU 0 only	
FMH-1 session control block (SCB) compression	Not allowed				

Table 14. FM Profile Characteristics (continued)

SNA Facility	FMP 2	FMP 3	FMP 4	FMP 7	FMP 18
Data flow control RUs allowed	None	• CANCEL • SIGNAL • LUSTAT (SLU only) • CHASE • SHUTD • SHUTC • RSHUTD • BID, RTR	• CANCEL • SIGNAL • LUSTAT • QEC • QC • RELQ • CHASE • SHUTD • SHUTC • RSHUTD • BID, RTR	• CANCEL • SIGNAL • LUSTAT • RSHUTD	• CANCEL • SIGNAL • LUSTAT • CHASE • BIS, SBI • BID, RTR
FM Headers	Not allowed				
Brackets	Restricted use				
Flow protocol	FDX				
Recovery	By PLU only				

Using LUA Verbs

An application accesses LUA through LUA verbs. Each verb supplies parameters to LUA, which performs the desired function and returns parameters to the application.

RUI Verb Summary

The following is a brief summary of the seven RUI verbs that an application can use. (For a detailed explanation of each verb, see Chapter 13. RUI Verbs.)

RUI_BID
Enables the application to determine when information from the host is available to be read.

RUI_INIT
Sets up the LU-SSCP session for an LUA application.

RUI_PURGE
Cancels an outstanding **RUI_READ** verb.

RUI_READ
Receives data or status information sent from the host to the LUA application’s LU, on either the LU-SSCP session or the LU-LU session.

RUI_TERM
Ends the LU-SSCP session for an LUA application. It also brings down the LU-LU session if it is active.

RUI_WRITE
Sends data to the host on either the LU-SSCP session or the LU-LU session.

In addition, Communications Server can return the RUI_INIT_STATUS indication to the LUA application. It carries information about the state reached in processing an outstanding **RUI_INIT** verb.

RUI Sessions

An RUI session consists of the ownership of an LU for a period of time determined by the application, which can include establishing a session between an SSCP and an LU (called an SSCP-LU session). An RUI session can also include establishing one or more non-overlapped LU-LU sessions. If the SSCP-LU session fails because of a loss-of-contact or another reset condition, the RUI session ends. An RUI session begins with an **RUI_INIT** verb and ends normally with an **RUI_TERM** verb.

Issuing RUI Verbs

Table 15 on page 165 shows the valid conditions under which an RUI application program can issue verbs to the RUI API for a given LU. The entries in the leftmost column represent incoming verbs. The entries in the first row represent verbs that are in progress. If an entry in the table is “OK,” the combination of verbs represents a valid condition. If an entry in the table is “Error,” the combination of verbs represents an incorrect condition and an error code is returned to the LUA application program.

Table 15. RUI Verb Conditions

	In-Progress Commands						
Incoming Commands	**No Current Session**	**RUI_INIT**	**RUI_TERM**	**RUI_WRITE**	**RUI_READ**	**RUI_PURGE**	**RUI_BID**
RUI_INIT	OK	Error	Error	Error	Error	Error	Error
RUI_TERM	Error	OK	Error	OK	OK	OK	OK
RUI_WRITE	Error	Error	Error	OK (See Note 1)	OK	OK	OK
RUI_READ	Error	Error	Error	OK	OK (See Note 2)	OK	OK
RUI_PURGE	Error	Error	Error	OK	OK	Error	OK
RUI_BID	Error	Error	Error	OK	OK	OK	Error

Notes:

1. The RUI permits a maximum of two active **RUI_WRITE** verbs per session at the same time. The active **RUI_WRITE** verbs must be for different session flows. Four session flows are possible:
   - SSCP-LU expedited
   - SSCP-LU normal
   - LU-LU expedited
   - LU-LU normal
2. The RUI permits a maximum of four active **RUI_READ** verbs per session at the same time. The active **RUI_READ** verbs must be for different session flows.

SLI Verb Summary

The following is a brief summary of the six SLI verbs that an application can use. (For a detailed explanation of each verb, see Chapter 15. SLI Verbs).

SLI_OPEN

- Initializes an LUA session by name, establishing the connection to Communications Server, allocating an LU, and setting up the SSCP-LU session.
- Processes BIND, STSN, and SDT messages to set up the LU-LU session.
- Optionally calls application functions to examine BIND and STSN messages.

SLI_CLOSE

- Terminates an LUA session.
- If an LU-LU session is active, exchanges CHASE, CLEAR, RSHUTD, and UNBIND messages with the PLU to end the session.
- Releases the LU and closes the connection to Communications Server, causing it to notify the host that the LU is no longer session capable.

SLI_RECEIVE
Receives a single logical message, either a request or a response, into a buffer.

SLI_SEND
Sends a logical message, either a request or a response, from a buffer.

SLI_PURGE
Cancels an outstanding, incomplete **SLI_RECEIVE** verb.

SLI_BID
Waits for an entire logical message to arrive, then completes to inform the applicaiton that data or status is ready to be received.

Comparing the RUI and SLI

An application must use either the RUI or SLI on any one session. The choice is established when the application starts the session, either with **RUI_INIT** or with **SLI_OPEN**. Once a verb opens a session, only verbs of the same interface can be used with that session.

In general, the SLI is simpler to use. SLI applications are easier to design. The RUI gives you more detailed control over the SNA protocols, when that is required.

Asynchronous Verb Completion

Some LUA verbs complete quickly, after some local processing (for example the **RUI_PURGE** verb); however, most verbs take some time to complete because they require messages to be sent to and received from the host application. Because of this, LUA is implemented as an asynchronous interface; control can be returned to the application while a verb is still in progress, so the application is free to continue with further processing (including issuing other LUA verbs). The way that LUA returns control to the application is by way of a event handle in the verb.

If Communications Server's verb response signal is delayed (for example, because it needs to wait for information from the remote node), then the stub should return the verb asynchronously. The API does this by setting the primary return code to LUA_IN_PROGRESS, and the **lua_flag2** to LUA_ASYNC. The application can now either perform other processing, or wait for notification from the API that the verb has completed. When the verb completes, the application is notified by the setting of the primary return code in the VCB to its final value, and leaving the **lua_flag2** set to LUA_ASYNC.

Sample LUA Communication Sequence

The following is an example of an LUA communication sequence when using the RUI API. It shows the LUA verbs used to start a session, exchange data, and end the session, and the SNA messages sent and received. The arrows indicate the direction in which SNA messages flow.

The following abbreviations are used:

SSCP norm
LU-SSCP session, normal flow

LU norm
LU-LU session, normal flow

LU exp	LU-LU session, expedited flow
+rsp	Positive response to the indicated message
BC	Begin chain
MC	Middle of chain
EC	End chain
CD	Change direction indicator set
RQD	Definite response required

Table 16. LUA Communication Sequence Using the RUI API

Verb issued by LUA application	SNA message	Flow direction Application	Host
RUI_INIT	(ACTLU)	<-----	
	(ACTLU +rsp)	----->	
RUI_WRITE (SSCP norm)	INITSELF	----->	
RUI_READ (SSCP norm)	INITSELF +rsp	<-----	
RUI_READ (LU exp)	BIND	<-----	
RUI_WRITE (LU exp)	BIND +rsp	----->	
RUI_READ (LU exp)	SDT	<-----	
RUI_WRITE (LU exp)	SDT +rsp	----->	
RUI_WRITE (LU norm)	data, BC	----->	
RUI_WRITE (LU norm)	data, MC	----->	
RUI_WRITE (LU norm)	data, EC, CD, RQD	----->	
RUI_READ (LU norm)	data +rsp	<-----	
RUI_READ (LU norm)	data, BC	<-----	
RUI_READ (LU norm)	data, MC	<-----	
RUI_READ (LU norm)	data, EC, RQD	<-----	
RUI_WRITE (LU norm)	data +rsp	----->	
RUI_READ (LU exp)	UNBIND	<-----	
RUI_WRITE (LU exp)	UNBIND +rsp	----->	
RUI_TERM	(NOTIFY)	----->	
	(NOTIFY +rsp)	<-----	

Table 16. LUA Communication Sequence Using the RUI API (continued)

Verb issued by LUA application	SNA message	Flow direction Application	Host

In this example, the application performs the following steps:

1. Issues the **RUI_INIT** verb to establish the LU-SSCP session. (The **RUI_INIT** verb does not complete until Communications Server has received an ACTLU message from the host and sent a positive response; however, these messages are handled by Communications Server and not exposed to the LUA application.)
2. Sends an **INITSELF** message to the SSCP to request a **BIND**, and reads the response.
3. Reads a **BIND** message from the host, and writes the response. This establishes the LU-LU session.
4. Reads an **SDT** message from the host, which indicates that initialization is complete and data transfer can begin.
5. Sends a chain of data consisting of three RUs (the last indicates that a definite response is required), and reads the response.
6. Reads a chain of data consisting of three RUs, and writes the response.
7. Reads an **UNBIND** message from the host, and writes the response. This terminates the LU-LU session.
8. Issues the **RUI_TERM** verb to terminate the LU-SSCP session. (Communications Server sends a NOTIFY message to the host and waits for a positive response; however, these messages are handled by Communications Server and are not exposed to the LUA application.)

BIND Checking

During initialization of the LU-LU session, the host sends a **BIND** message to the Communications Server LUA application that contains information such as RU sizes to be used by the LU-LU session. When using the RUI API, Communications Server returns this message to the LUA application on an **RUI_READ** verb. It is the responsibility of the LUA application to check that the parameters specified on the **BIND** are suitable. The application has the following options:

- Accept the **BIND** as it is, by issuing an **RUI_WRITE** verb containing an OK response to the **BIND**. No data needs to be sent on the response.
- Try to negotiate one or more **BIND** parameters (this is only permitted if the **BIND** is negotiable). To do this, the application issues an **RUI_WRITE** verb containing an OK response, but including the modified **BIND** as data.
- Reject the **BIND** by issuing an **RUI_WRITE** verb containing a negative response, using an appropriate SNA sense code as data.

See Chapter 13. RUI Verbs, for more information on the **RUI_WRITE** verb.

Note: Validation of the **BIND** parameters, and ensuring that all messages sent are consistent with them, is the responsibility of the LUA application. However, the following two restrictions apply:

- Communications Server rejects any **RUI_WRITE** verb that specifies an RU length greater than the size specified on the **BIND**.

- Communications Server requires the **BIND** to specify that the secondary LU is the contention winner, and that error recovery is the responsibility of the contention loser.

Note: To process a negotiable **BIND** command when using the SLI API, the application must supply a special routine. For more information, see “SLI_BIND_ROUTINE” on page 259.

Negative Responses and SNA Sense Codes

SNA sense codes may be returned to an LUA application in the following cases:

- When the host sends a negative response to a request from the LUA application, this includes an SNA sense code indicating the reason for the negative response. This is reported to the application on a subsequent **SLI_RECEIVE** or **RUI_READ** verb, as follows:
  - The primary return code is LUA_OK.
  - The Request/Response Indicator, Response Type Indicator, and Sense Data Included Indicator are all set to 1, indicating a negative response which includes sense data.
  - The data returned by the **RUI_READ** or **SLI_RECEIVE** verb is the SNA sense code.
- When Communications Server receives incorrect data from the host, it generally sends a negative response to the host and does not pass the incorrect data to the LUA application. This is reported to the application on a subsequent **RUI_READ** or **RUI_BID** /**SLI_RECEIVE** or **SLI_BID** verb, as follows:
  - The primary return code is LUA_NEGATIVE_RSP.
  - The secondary return code is the SNA sense code sent to the host.
- In some cases, Communications Server detects that data supplied by the host is not valid, but cannot determine the correct sense code to send. In this case, it passes the incorrect data in an Exception Request (EXR) to the LUA application on an **RUI_READ** or **SLI_RECEIVE** verb in the following way:
  - The Request/Response Indicator is set to zero, indicating a request.
  - The Sense Data Included Indicator is set to one, indicating that sense data is included (this indicator is normally used only for a response).
  - The message data is replaced by a suggested SNA sense code.

  The application must then send a negative response to the message; it may use the sense code suggested by Communications Server, or may alter it.
- Communications Server may send a sense code to the application to indicate that data supplied by the application was not valid. This is reported to the application on the **RUI_WRITE** or **SLI_SEND** verb that supplied the data, as follows:
  - The primary return code is LUA_UNSUCCESSFUL.
  - The secondary return code is the SNA sense code.

Distinguishing SNA Sense Codes from Other Secondary Return Codes

For a secondary return code which is not a sense code, the first two bytes of this value are always zero. For an SNA sense code, the first two bytes are non-zero;

the first byte gives the sense code category, and the second identifies a particular sense code within that category. (The third and fourth bytes may contain additional information, or may be zero.)

Information on SNA Sense Codes

If you need information on a returned sense code, refer to *IBM Systems Network Architecture: Formats*. The sense codes are listed in numeric order by category.

Segmentation

Segmentation of RUs is handled by LUA. LUA always passes complete RUs to the application, and the application should pass complete RUs to LUA.

Courtesy Acknowledgments

Communications Server keeps a record of requests received from the host in order to correlate any response sent by the application with the appropriate request. When the application sends a response, Communications Server correlates this with the data from the original request, and can then free the storage associated with it.

If the host specifies exception response only (a negative response may be sent, but a positive response should not be sent), Communications Server must still keep a record of the request in case the application subsequently sends a negative response. If the application does not send a response, the storage associated with this request cannot be freed.

Because of this, Communications Server enables the LUA application to issue a positive response to an exception-response-only request from the host (this is known as a courtesy acknowledgment). The response is not sent to the host, but is used by Communications Server to clear the storage associated with the request.

Purging Data to End of Chain

When the host sends a chain of request units to an LUA application, the application may wait until the last RU in the chain is received before sending a response, or it may send a negative response to an RU which is not the last in the chain. If a negative response is sent mid-chain, Communications Server purges all subsequent RUs from this chain, and does not send them to the application.

When Communications Server receives the last RU in the chain, it indicates this to the application by setting the primary return code of an **RUI_READ** or **RUI_BID** verb to LUA_NEGATIVE_RSP with a zero secondary return code.

Note: The host may terminate the chain by sending a message such as CANCEL while in mid-chain. In this case, the CANCEL message is returned to the application on an **RUI_READ** verb, and the LUA_NEGATIVE_RSP return code is not used.

Configuration

Each LU used by an LUA application must be configured using Communications Server NOF verbs or through the SNA Node Configuration program. (Refer to *Communications Server System Management Programming* for more information.) In addition, the Communications Server configuration may include LUA LU pools. A pool is a group of LUs with similar characteristics, such that an application can use any free LU from the group. This can be used to allocate LUs on a first-come, first-served basis when there are more applications than LUs available, or to provide a choice of LUs on different links.

LUA LU Pool (Optional)

If required, you can configure more than one LUA LU for use by the application, and group the LUs into a pool. This means that an application can specify the pool rather than a specific LU when attempting to start a session, and will be assigned the first available LU from the pool.

An LUA application indicates to Communications Server that it wants to start a session by issuing an **RUI_INIT** or **SLI_OPEN** verb with an LU name. This name must match the name of an LUA LU or LU pool that has previously been defined in *Communications Server System Management Programming* Communications Server uses this name as follows:

- If the name supplied is the name of an LU that is not in a pool, a session will be assigned using that LU if it is available (that is, if it is not already in use by an LUA application).
- If the name supplied is the name of an LU pool, or the name of a specific LU within the pool that is already in use, then a session will be assigned using the first available LU in the pool (if one is available).

  Note: This may not be the LU whose name was specified on the **RUI_INIT** or **SLI_OPEN** verb.

SNA API Client Considerations

If your LUA application resides on a client workstation, an LUA session should also be defined on the local workstation. This LUA session name can contain multiple communication servers and LUA definitions, thus allowing the SNA client code to roll over to new servers when connections become unavailable.

Chapter 10. Features of the LUA Verbs

This chapter covers the following special cases and usage tips for the LUA verbs.

- Handling exception requests—requests from LUA for your program to issue a negative response
- Minimizing LAN traffic through program design
- Dealing with indefinite suspensions of LUA verbs
- Supporting data compression for RUI and SLI APIs
- Recovering from session failure

Handling Exception Requests

Both the RUI and SLI monitor the state of several protocols and as validates the format of RUs. When the interface detects an improper RU coming from the PLU, it must issue a negative response. LUA notifies your application of this detected error by formatting the incoming RU as an exception request (EXR). An EXR is delivered to your program on either a bid verb (**RUI_BID** or **SLI_BID**) or on an input verb (**RUI_READ** or **SLI_RECEIVE**). An EXR is indicated by the following conditions in the request header (RH):

- *lua_rh.rri* set to 0 (RU is a request unit)
- *lua_rh.sdi* set to 1 (sense data included)

This is an abnormal combination of RH bits. Sense data is normally the contents of a response RU, not a request RU. LUA uses this abnormal combination to alert your program to the abnormal fact that the PLU has apparently made an error. A 4-byte sense code is part of the EXR; it specifies the error detected. In addition to the sense data, LUA returns up to three bytes of the original RU.

Changing the Verb Record

Your application must format the EXR as a negative response and send it to the PLU using either **RUI_WRITE** or **RUI_SEND**, depending on the API in use. To convert an EXR input to a response output, make the following changes in the verb record:

- Set *lua_rh.rri* to 1 to show this is a response.
- Set *lua_rh.ri* to 1, indicating a negative response.
- Set the appropriate data-flow flag in *lua_flag1* based on the values in *lua_flag2*.
- Set *lua_message_type* to LUA_MESSAGE_TYPE_RSP (for SLI) or to 0 (for RUI).
- Set *lua_opcode* to LUA_OPCODE _RUI_WRITE or LUA_OPCODE_SLI_SEND, depending on the API in use.
- Set *lua_data_length* to 4, the length of the sense data.
- Set *lua_data_ptr* to the address of the sense data, whose location depends on the verb that detected the EXR-if the verb was **RUI_BID** or **SLI_BID**, the sense data is in the ″peek buffer″ in the verb record; if the verb was **RUI_READ** or **SLI_RECEIVE** , the sense data is in the input buffer.
- Set *lua_max_length* to 0.

Your program can now use the verb record and buffer for the EXR to initiate the **SLI_SEND** or **RUI_WRITE** to send the negative response.

© Copyright IBM Corp. 1994, 1997

Handling Bracket Bid Reject

In all but one case, the sense code provided by LUA in an EXR is the only appropriate one to return to the PLU. When bracketing is in use, however, and the PLU asks to become speaker, your application has a choice of sense codes:

- LUA can reject a BID command from the PLU. To reject the BID, LUA formats an EXR containing the sense code LUA_BB_REJECT_NO_RTR, stating that the bracket bid is rejected and no RTR command will be issued later. The numeric value of this sense code is 0x00001308L (in Intel**, or byte-swapped, form, as you would code it in a C program).
- Your application can accept the BID command if it supports bracketing and can issue an RTR command later. To notify the PLU that its BID can now be accepted, you can change the sense code to LUA_BB_REJECT_RTR (value 0x00001408L), the sense code that states an RTR will be forthcoming. At some later time your application must format and send an RTR message.

Minimizing LAN Traffic

If your application must run on a client workstation, you can design it to minimize the overhead of the LAN traffic by reducing the use of "bid logic."

Reducing RUI_BID Usage

The verbs **RUI_BID** and **SLI_BID** are similiar; they wait until a data unit (or SLI status) is available at the server and then they complete. The completion of **RUI_BID** or **SLI_BID** notifies your program that data is ready, on a particular flow, and has a particular length. Your program can then allocate a buffer and issue an **RUI_READ** or **SLI_RECEIVE** verb for the data.

When you issue a bid verb followed by an input verb, the following four LAN messages are generated:

- A message to initiate the **RUI_BID** or **SLI_BID**
- A message to notify the workstation the bid is complete
- A message to initiate the **RUI_READ** or **SLI_RECEIVE**
- A message returning data to the workstation

However, **RUI_READ** or **SLI_RECEIVE** can do the same job in one step. If you simply initiate the **RUI_READ** or **SLI_RECEIVE** verb and wait for it to complete, two LAN messages are eliminated.

The only benefit of "bid logic" is that you find out the size of a message before you receive it. This allows you to defer allocating a data buffer until you know how large a buffer you need. When you use only input verbs, you must know the maximum buffer size in advance, rather than allocating a buffer after the bid completes.

Dealing with Suspensions

The completion of an SLI or RUI verb depends on the actions of the PLU application, the host system, the network, and Communications Server. If any one of these responds slowly or fails to respond, a verb can be suspended indefinitely. When designing your program, you can anticipate suspensions by giving the user or the program a way of terminating suspended verbs.

Canceling SLI_OPEN and RUI_INIT

The **SLI_OPEN** and **RUI_INIT** verbs suspend until the host activates the assigned LU. Normally the host will have sent an ACTLU command before the application starts up, but it is not required to do so. When the application starts up, the mainframe might be down or still initializing.

If your program needs to cancel an **SLI_OPEN** that has benn suspended, it can issue an **SLI_CLOSE** verb. If the program needs to cancel a suspended **RUI_INIT**, it can issue an **RUI_TERM** verb.

Canceling SLI_SEND and RUI_WRITE

When pacing is in use, output can be suspended. If the host temporarily stops reading data or fails to transmit a pacing response, either **SLI_SEND** or **RUI_WRITE** can be suspended waiting for the pacing window to open.

If your program needs to cancel a suspended **SLI_SEND**, it must close the session with **SLI_CLOSE**. To cancel a suspended **RUI_WRITE**, it must close the session with **RUI_TERM** .

Canceling SLI_RECEIVE and RUI_READ

An input verb is normally suspended until input arrives on the flow that the verb specified. Your program can cancel a pending **SLI_RECEIVE** using **SLI_PURGE**. It can cancel a pending **RUI_READ** using **RUI_PURGE**. Closing the session also cancels pending input verbs.

Ensuring Verb Completion

Your program can create the appearance of an indefinite wait if it mishandles verb completion. If the program starts a verb, fails to note that the verb completed synchronously, and waits for asynchronous completion, the program will wait forever.

The LUA entry points return as their explicit result the primary return code of the verb that was executed. The simplest way to tell whether a verb will be completed asynchronously is to test this explicit return for LUA_IN_PROGRESS, as shown in Figure 8.

```
unsigned short rc;
rc = RUI(ptrToTheVerb);
if (LUA_IN_PROGRESS == rc)
   // verb will complete later; the callback function will be entered
else
   // verb is finished now; the callback function will never be entered
```

Figure 8. Testing Verb Completion

Compressing Data

Data compression is supported for both the RUI and SLI API interfaces. The use of data compression is negoiated per session by the BIND and BIND response. If compression is negotiated for use on the session, then LZ9 or run-length encoding (RLE) compression algorithms are accepted inbound from the Primary LU (PLU) and RLE will be used for sending data to the PLU.

For both the RUI and SLI APIs, data compression can be handled by either of the following:

- The application compresses and decompresses data
- Communications Server delivers and accepts uncompressed data to and from the application.

Rules for Negotiating Data Compression Per Session

Following are rules for negotiating data compression for both RUI and SLI APIs per session.

RUI Rules

1. To allow the RUI application to handle the compression and decompression of data:
   - RUI receives the BIND request that has bits 6 and 7 of Byte 25 set to indicate compression is offered or requested.
   - The RUI application should return the positive BIND response with bits 6 and 7 of Byte 25 set to indicate ″offered or mandated compression accepted″.
2. To allow Communications Server to handle compression on behalf of the RUI application:
   - Use the Communications Server SNA Node Configuration utility to set the configuration node panel to indicate that the node supports compression.
   - RUI receives the BIND response with bits 6 and 7 of Byte 25 set to indicate compression is offered or requested.
   - RUI returns the positive BIND response with bits 6 and 7 of Byte 25 set to indicate ″no compression″, then Communications Server compresses and decompresses the data to the host.

SLI Rules

1. To allow the SLI application to handle the compression and decompression of data:
   - SLI must supply a BIND Callback routine when it issues the **SLI_OPEN** verb.
   - When SLI’s BIND callback routine is started, SLI receives the BIND request that has bits 6 and 7 of Byte 25 set to indicate compression is offered or requested.
   - SLI should return the BIND response with bits 6 and 7 of Byte 25 set to indicate ″offered or mandated compression accepted″.
2. To allow Communications Server to handle compression on behalf of SLI:
   - Use the Communications Server SNA Node Configuration utility to set the configuration node panel to indicate that the node supports compression.
   - If the application did not supply a BIND callback routine on the **SLI_OPEN** verb, SLI will by default set the BIND response to indicate that Communications Server will compress and decompress the data for SLI.

- If the application did supply a BIND callback rountine:
  - When the BIND callback routine is started, it receives the BIND request that has bits 6 and 7 of Byte 25 set to indicate compression is offered or requested.
  - SLI returns the BIND response with bits 6 and 7 of Byte 25 set to indicate ″no compression″, then Communications Server compresses and decompresses the data to the host.

Recovering from Session Failure

When an LUA communication verb completes with the primary return code LUA_SESSION_FAILURE, the LUA session has been closed due to an error. The specific error is documented by the secondary return code.

The session can often be reconstructed. LUA will attempt recovery if your program requests it.

When your program receives LUA_SESSION_FAILURE as the primary return code of any operation, it should do the following if it wants to recover:

- Avoid closing the session-the session is already closed.
- Reopen the session using the verb originally used to open the session (**RUI_INIT** or **SLI_OPEN**). If this verb completes with a nonzero primary return code, the session cannot be restarted at this time.
- Notify the interactive user when recovery is underway, because the recovery might take some time. The state of the user’s work will depend on the design of the PLU application.

Chapter 11. Implementing LUA Programs

This chapter describes some of the aspects of implementing and writing LUA programs. It includes the following topics:

- Calling and sequencing LUA services
- Writing LUA programs
- Using the asynchronous completion and callback functions
- Compiling and linking on different platforms

The Communications Server implementation of LUA is designed to be binary compatible with Microsoft** NT SNA Server and similar to the implementation of the RUI interface of OS/2 Communications Manager/2 Version 1.0 LUA.

Writing LUA Programs

The LUA contains one main DLL, for the RUI verbs and one for SLI verbs. An LUA application program calls these DLLs to issue verbs.

The LUA application program sets selected fields in a verb control block and calls the RUI or SLI, passing a pointer to the verb control block. The fields in the verb control block define the requested action to the LUA. The LUA modifies fields in the verb control block to indicate the results of the action before the LUA returns control to the application program. The application program can then use the returned parameters from the verb control block in subsequent processing.

The table below shows source module usage of supplied header files and libraries needed to compile and link RUI programs.

Table 17. Header Files and Libraries for RUI APIs

Operating System	Header File	Library	DLL Name
WINNT & WIN95	WINLUA.H	WINRUI32.LIB	WINRUI32.DLL
WIN3.1	WINRUI.H	WINRUI.LIB	WINRUI.DLL
OS/2	LUA_C.H	ACSRUI.LIB	ACSRUI.DLL

Table 18. Header Files and Libraries for SLI APIs

Operating System	Header File	Library	DLL Name
WINNT	WINLUA.H	WINSLI32.LIB	WINSLI32.DLL

Note: SLI API is supported on the server and is not supported by the Communications Server clients.

Calling LUA Services

Your program invokes LUA services by calling a designated entry point and passing a single parameter — the address of a data structure called a *verb record*. The record contains the input parameters for a particular function. LUA updates the record with the output parameters resulting from the operation.

Understanding Verb Record Contents

Although structured differently, the three types of verb records all provide fields for the following parameters:

Operation
A number specifying the particular operation to be done. .

Verb record length
The size of the verb record, which can vary from operation to operation, and which LUA needs in order to process the record.

Session identifier
In communication and service verbs, a number to identify the session or the name of the session.

Primary return code
A number returned by LUA to indicate general success or failure.

Secondary return code
A number returned by LUA on a failure to indicate the specific problem.

Correlator
A long integer that your application can use to relate the verb record to other data, or to identify the verb record during an asynchronous completion.

Post handle
The handle of an event to be posted when the verb completes asynchronously.

For Windows NT and Windows 95, this should be an event handle. In Windows 3.1, this should be a window handle. For OS/2, this should be a semaphore handle.

Most of these fields have the identical data type and are at the identical offset in every verb record in which they appear. The operation code and verb-length fields have different characteristics, however.

Multiple Processes

An LUA application program is restricted to a single process. Different instances of the same LUA application program can start in different processes, but each application program must use a different LUA LU.

In addition, a single process can comprise multiple LUAapplication programs, each with its own LUA LU.

Multiple Threads

A single LUA application program can use multiple threads to issue verbs. This lets you issue multiple verbs simultaneously from a single LUA application program. Different instances of the same LUA application program can start in different threads, but each application program can use a different LUA LU.

Note: After an LUA application program issues a verb, it should not change any part of the verb control block until the verb is complete. The LUA uses only the application copy of the verb control block. See “LUA Verb Postings” on page 181 for additional information.

LUA Verb Postings

LUA verbs complete synchronously or asynchronously. Synchronous verb completion means that when LUA returns to the LUA application program after a call to LUA, all processing for that verb is finished and the asynchronous posting method is not used. Because of timing conditions, a verb can complete asynchronously, but all processing is completed by the time LUA returns to the LUA application program. Asynchronous verb completion means that LUA uses the posting method to notify the application program when processing completes, either successfully or unsuccessfully.

An LUA application program can be notified in one of the following ways when a verb completes asynchronously:

- The application specifies an event in the **lua_post_handle** parameter. This is set when the verb is complete.
- The application uses the WinRUI or WinSLI entry points for asynchronous message notification.

The LUA application program uses the **lua_flag2.async** and **lua_prim_rc** parameters to determine the verb processing state.

Converting to EBCDIC from ASCII

Typically, all messages sent to the host are in EBCDIC, and the PLU assumes that the messages are in EBCDIC. For example, a PLU name that appears in a **BIND** should be an EBCDIC string. An LUA application program that stores strings in ASCII should convert the strings to EBCDIC before the strings are sent in SNA messages.

Whether an LUA application program needs to convert application data depends on a private agreement between the partner application programs. If your LUA application program communicates with a node that normally uses EBCDIC, you should convert your ASCII data to EBCDIC where appropriate.

Conversion of ASCII to EBCDIC (or vice versa) can be done by the convert verbs described in “Chapter 17. Common Services Verbs (CSV)” on page 279.

Chapter 12. RUI Entry Points

This chapter describes the procedure entry points for LUA.

The RUI DLL defines the following procedure entry points:

RUI()

Provides event notification for all **RUI** verbs.

Syntax

```
void WINAPI RUI (LUA_VERB_RECORD* vcb);
```

Parameter	**Description**
vcb	Supplied parameter; specifies the address of the verb control block.

Returned Values

The value returned in *lua_prim_rc* indicates whether asynchronous notification will occur. If the field is set to LUA_IN_PROGRESS, asynchronous notification will occur through event signaling. If the flag is not LUA_IN_PROGRESS, the request completed synchronously. Examine the primary return code and secondary return code for any errors.

Usage Notes

The application must provide a handle to an event in the *lua_post_handle* parameter of the verb control block. The event must be in the not-signaled state.

When the asynchronous operation is complete, the application is notified by the signaling of the event. Upon signaling of the event, examine the primary return code and secondary return code for any error conditions. **See also:** “WinRUI” on page 185 .

This is the only entry point supported under OS/2.

WinRUI

Provides asynchronous message notification for all RUI verbs.

Syntax

```
int WINAPI WinRUI (HWND hWnd,
                   LUA_VERB_RECORD* vcb);
```

Parameter	Description
hwnd	Window handle to receive completion message.
vcb	Pointer to verb control block.

Returned Values

The function returns a value indicating whether the request was accepted by the RUI for processing. A returned value of 0 indicates that the request was accepted and will be processed. A value other than 0 indicates an error. Possible error codes are as follows:

WLUAINVALIDHANDLE
The window handle provided is not valid.

The value returned in **lua_flag2.async** indicates whether asynchronous notification will occur. If the flag is set (nonzero), asynchronous notification will occur through a message posted to the application's message queue. If the flag is not set, the request completed synchronously. Examine the primary return code and secondary return code for any error conditions.

Usage Notes

Upon completion of the verb, the application's window *hWind* receives the message returned by **RegisterWindowMessage** with **WinRUI** as the input string. The **lParam** argument contains the address of the VCB being posted as complete. The *wParam* argument is undefined. It is possible for the request to be accepted for processing (the function call returns 0), but rejected later with a primary return code and secondary return code set in the VCB. Examine the primary return code and secondary return code for any error conditions.

If the application calls **WinRUI** without first initializing the session using **WinRUIStartup**, an error is returned.

See also:: "RUI()" on page 184.

WinRUICleanup()

Terminates and deregisters an application from the RUI API.

Syntax

```
BOOL WINAPI WinRUICleanup (void);
```

Returned Values

The return value indicates success or failure of the deregistration. If the value is not `0`, the application was successfully deregistered. If the value is `0`, the application was not deregistered.

Usage Notes

Use **WinRUICleanup** to deregister the RUI API, for example, to free up resources allocated to the specific application.

If **WinRUICleanup** is called while LUs are in session (**RUI_TERM** not issued), the cleanup code issues **RUI_TERM** close type ABEND for the application for all open sessions. **See also:** “WinRUIStartup()” on page 190.

WinRUIGetLastInitStatus()

This is the only supported for applications residing on the server.

This function provides a way for an application to determine the status of an **RUI_INIT** so that the application can determine whether the **RUI_INIT** should be timed out. Use this call to initiate the reporting of status, terminate the reporting of status, or find the current status. For details, see the Usage Notes section.

Syntax

```
int WINAPI WinRUIGetLastInitStatus (DWORD dwSid,
                                    HANDLE hStatusHandle,
                                    DWORD dwNotifyType,
                                    BOOL bClearPrevious);
```

Parameter
Description

dwSid Session identifier of the session for which status will be determined. If the value is `0`, *hStatusHandle* is used to report status on all sessions. The *lua_sid* parameter in the **RUI_INIT** VCB is valid as soon as the call to **RUI**() or **WinRUI**() for the **RUI_INIT** returns.

hStatusHandle
A handle used for signaling the application that the status for the session has changed. Can be a window handle, an event handle, or NULL; *dwNotifyType* must be set accordingly:

- If *hStatusHandle* is a window handle, status is sent to the application through a window message. The program obtains the message from **RegisterWindowMessage** using the string **WinRUI**. The parameter wParam contains the session status (see Return Values). Depending on the value of *dwNotifyType*, **lParam** contains either the RUI session ID of the session, or the value of **lua_correlator** from the **RUI_INIT** verb.
- If *hStatusHandle* is an event handle, when the status for the session specified by *dwSid* changes, the event is put into the signaled state. The application must then make a further call to **WinRUIGetLastInitStatus()** to find out the new status. The event should not be the same as one used for signaling completion of any **RUI** verb.
- If *hStatusHandle* is NULL, the status of the session specified by *dwSid* is returned in the return code. In this case, *dwSid* must not be 0 unless *bClearPrevious* is TRUE. If *hStatusHandle* is NULL, *dwNotifyType* is ignored.

dwNotifyType
The type of indication required. This determines the contents of the **lParam** of the window message and how **WinRUIGetLastInitStatus()** interprets *hStatusHandle*. Permitted values are:

WLUA_NTFY_EVENT
The *hStatusHandle* parameter contains an event handle.

WLUA_NTFY_MSG_CORRELATOR
The *hStatusHandle* parameter contains a window handle and the **lParam** of the returned window message should contain the LUA correlator and RUI.

WLUA_NTFY_MSG_SID
The *hStatusHandle* parameter contains a window handle and the **lParam** of the returned window message should contain the LUA session identifier.

bClearPrevious
If TRUE, status messages are no longer sent for the session identified by *dwSid*. If *dwSid* is `0`, status messages are no longer sent for any session. If *bClearPrevious* is TRUE, *hStatusHandle* and *dwNotifyType* are ignored.

Returned Values

WLUASYSNOTREADY
Communications Server is not running.

WLUANTFYINVALID
The *dwNotifyType* parameter is not valid.

WLUAINVALIDHANDLE
The *hStatusHandle* parameter does not contain a valid handle.

WLUALINKINACTIVE
The link to the host is not yet active.

WLUAPUINACTIVE
The link to the host is active, but no **ACTPU** has yet been received.

WLUAPUACTIVE
An **ACTPU** has been received.

WLUAPUREACTIVATED
The PU has been reactivated.

WLUAUINACTIVE
The link to the host is active, and an **ACTPU** has been received, but no **ACTLU** has been received.

WLUALUACTIVE
The LU is active.

WLUAUNKNOWN
The session is in an unknown status. (This is an internal error.)

WLUASIDINVALID
The SID specified does not match any known by the **RUI**.

WLUASIDZERO
The *hStatusHandle* parameter is NULL and *bClearPrevious* is FALSE, but *dwSid* is `0`.

WLUAGLOBALHANDLER
The *dwSid* parameter is `0`, and messages from all sessions will be notified. (This is a normal return code, not an error.)

Usage Notes

This function is intended to be used either with a window handle or an event handle to enable asynchronous notification of status changes, but it can also be used on its own to find out the current status of a session.

To use this function with a window handle, you can implement it in one of two ways as follows:

```
WinRUIGetLastInitStatus(Sid,Handle,WLUA_NTFY_MSG_CORRELATOR,FALSE);
```

or

```
WinRUIGetLastInitStatus(Sid,Handle,WLUA_NTFY_MSG_SID,FALSE);
```

With this implementation, changes in status are reported by a window message sent to the window handle specified. If WLUA_NTFY_MSG_CORRELATOR is specified, the **lParam** field in the window message contains the **lua_correlator** field for the session. If WLUA_NFTY_MSG_SID is specified, the **lParam** field in the window message contains the LUA session identifier for the session.

When the function has been used with a window handle, use the following command to cancel the reporting of status:

```
WinRUIGetLastInitStatus(Sid,NULL,0,TRUE);
```

For this implementation, note that if *Sid* is nonzero, status is only reported for that session. If *Sid* is `0`, status is reported for all sessions.

To use this function with an event handle, implement it as follows:

```
WinRUIGetLastInitStatus(Sid,Handle,WLUA_NOTIFY_EVENT,FALSE);
```

The event whose handle is given will be signaled when a change in state occurs. Because no information is returned when an event is signaled, the following call must be issued to find out the status:

```
Statu = WinRUIGetLastInitStatus(Sid,NULL,0,0,FALSE);
```

In this case, a *Sid* must be specified.

When the function has been used with an event handle, use the following command to cancel the reporting of status:

```
WinRUIGetLastInitStatus(Sid,NULL,0,TRUE);
```

To use this function to query the current status of a session, it is not necessary to use an event or window handle. Instead, use the following command:

```
Status = WinRUIGetLastInitStatus(Sid,NULL,0,0,FALSE);
```

WinRUIStartup()

Enables an application to specify the required version of the RUI API and to retrieve details of the API.

Syntax

```
int WINAPI WinRUIStartup (WORD wVersionRequired,
                          LPWLUADATA* luadata);
```

Parameter
: **Description**

wVersionRequired
: Specifies the version of RUI API support required. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.

luadata
: Returns version of RUI implementation.

Returned Values

The return value specifies whether the application was registered successfully and whether the RUI API can support the specified version number. If the value is `0`, it was registered successfully and the specified version can be supported. Otherwise, the return value is one of the following values:

WLUAVERNOTSUPPORTED
: The version of RUI API support requested is not provided by this particular RUI API.

WLUAINVALID
: The version requested could not be determined.

Usage Notes

This call is intended to aid in compatibility with future versions of the API. The current version is 1.0. **See also:** “WinRUICleanup()” on page 186.

GetLuaReturnCode()

Converts the primary and secondary return codes in the VCB to a printable string. This function provides a standard set of error strings for use by LUA applications.

Syntax

```
int WINAPI GetLuaReturnCode (lua_common* vcb,
                             UINT buffer_length,
                             unsigned char* buffer_addr);
```

Parameter
Description

vcb Supplied parameter; specifies the address of the verb control block.

buffer_length
Supplied parameter; specifies the length (in bytes) of the buffer pointed to by buffer_addr. The recommended length is 256.

buffer_addr
Supplied/returned parameter; specifies the address of the buffer that will hold the formatted, null-terminated string; the length of the string in the specified buffer.

Returned Values

0x20000001
The parameters are not valid; the function could not read from the specified verb control block or could not write to the specified buffer.

0x20000002
The specified buffer is too small.

Usage Notes

The descriptive error string returned in *buffer_addr* does not terminate with a new line character (**/n**).

Call Example

The following example shows how to call **WINRUI32.DLL**. The header file for this DLL is **WINLUA.H**. This example calls the **RUI DLL** to issue an **RUI** verb from a program:

```
#include "WINLUA.H"                         /* LUA C include file for
                                               the LUA Application. */
. . .

. . .

example()
{
LUA_VERB_RECORD    VerbRecord;              /* Declare VerbRecord as a verb
                                               control block using the
                                               TYPEDEF in WINLUA.H */
. . .

WINRUI((LUA_VERB_RECORD *) &VerbRecord);    /* Call the RUI API */
. . .
}
```

Chapter 13. RUI Verbs

This chapter contains the following information:

- Details of the LUA common control block structure
- A description of each LUA verb for all LUA verbs and the LUA verb records

The following information is provided for each LUA verb:

- The purpose of the verb.
- Parameters supplied to and returned by LUA. The description of each parameter includes information on the valid values for that parameter, and any additional information necessary.
- Interactions with other verbs.
- Additional information describing the use of the verb.

Note: Parameters marked as *reserved* should always be set to zero.

LUA Verb Control Block Format

The verb control block consists of:

- **lua_common**, used for all verbs and described in "Common Verb Header".
- **specific**, used only for the **RUI_BID** verb (described in "RUI_BID Data Structure" on page 197).

The structure is defined as follows:

```
typedef struct lua_verb_record
{
 LUA_COMMON        common;                 /* The common verb header     */
 union
 {
   unsigned char   lua_peek_data[12];      /* field specific to RUI_BID  */
 }
} LUA_VERB_RECORD;
```

Common Verb Header

The Communications Server LUA uses a generic *common verb header* to transport all incoming and outgoing data. The fields in the verb control block are defined as follows:

```
typedef struct lua_common
{
 unsigned short    lua_verb;           /* LUA_VERB_RUI               */
 unsigned short    lua_verb_length;    /* VCB length                 */
 unsigned short    lua_prim_rc;        /* primary return code        */
 unsigned long     lua_sec_rc;         /* secondary return code      */
 unsigned short    lua_opcode;         /* verb opcode                */
 unsigned long     lua_correlator;     /* verb correlator            */
 unsigned char     lua_luname[8];      /* local LU name              */
 unsigned short    lua_extension_list_offset;
                                       /* reserved                   */
 unsigned short    lua_cobol_offset;   /* reserved                   */
 unsigned long     lua_sid;            /* session ID                 */
 unsigned short    lua_max_length;     /* max buffer length          */
 unsigned short    lua_data_length;    /* actual data length         */
 unsigned char    *lua_data_ptr;       /* data pointer               */
 unsigned long     lua_post_handle;    /* post handle                */
 LUA_TH          lua_th;             /* TH structure               */
 unsigned char     lua_rh;             /* message RH                 */
 unsigned char     lua_flag1;          /* application message flag   */
 unsigned char     lua_message_type;   /* SNA message type           */
 unsigned char     lua_flag2;          /* LUA message flag           */
 unsigned char     lua_resv56[7];      /* reserved                   */
 unsigned char     lua_encr_decr_option;/* cryptography              */
} LUA_COMMON;
typedef struct lua_th
{
 unsigned char     reserv1;            /* reserved                   */
 unsigned char     daf;                /* DAF                        */
 unsigned char     oaf;                /* OAF                        */
 unsigned char     snf[2];             /* SNF                        */
} LUA_TH;
```

lua_verb
Identifies this as an **LUA** verb: always set to **LUA_VERB_RUI**.

lua_verb_length
Length of the verb control block.

lua_prim_rc
Primary return code set by LUA.

lua_sec_rc
Secondary return code set by LUA.

lua_opcode
Verb operation code, which identifies the **LUA** verb being issued.

lua_correlator
A 4-byte correlator, which you can use to correlate this verb with other processing in your application. LUA does not use this parameter.

lua_luname
The local LU name used by the LUA session (in ASCII). This can be an LU name or an LU pool name; see “RUI_INIT” on page 203 for more information.

lua_sid
The session ID of the LUA session on which this verb is issued.

lua_max_length
The length of the buffer used to receive data.

lua_data_length
The length of the data to be sent, or the actual length of data received.

lua_data_ptr
A pointer to the data to be sent, or the data buffer to receive data.

lua_th.flags
Specifies the flags set in the transmission header. (Refer to *Systems Network Architecture: Formats* for more information.) It can be one or more of the following values ORed together:

LUA_FID
Format identification type 2

LUA_MPF
Segmenting mapping field

LUA_BBIU
Begin BIU

LUA_EBIU
End BIU

LUA_ODAI
OAF-DAF assignor Indicator

LUA_EFI
Expedited Flow Indicator

lua_th.daf
DAF (Destination address field).

lua_th.oaf
OAF (Originating address field).

lua_th.snf
Sequence number field.

lua_rh Specifies the request/response header (RH) of the message sent or received. (Refer to *Systems Network Architecture: Formats* for more information.) This can be one or more of the following values ORed together:

LUA_RRI
Request-response indicator

LUA_RH_FMD
RU category: FMI data segment

LUA_RH_NC
RU category: Network control

LUA_RH_DFC
RU category: Data flow control

LUA_RH_SC
RU category: Session control

LUA_FI
Format indicator

LUA_SDI
Sense data included indicator

LUA_BCI
Begin chain indicator

LUA_ECI
End chain indicator

LUA_DR1I
Definite Response 1 indicator

LUA_DR2I
Definite Response 2 indicator

LUA_RI
Exception response indicator (for a request), or response type indicator (for a response)

LUA_QRI
Queued Response indicator

LUA_PI
Pacing indicator

LUA_BBI
Begin Bracket indicator

LUA_EBI
End Bracket indicator

LUA_CDI
Change Direction indicator

LUA_CSI
Code Selection indicator

LUA_EDI
Enciphered Data indicator

LUA_PDI
Padded Data indicator

lua_flag1
Specifies flags for messages supplied by the application. (Refer to *Systems Network Architecture: Formats* for more information.) The flags can be one or more of the following values ORed together:

LUA_BID_ENABLE
Bid Enable indicator

LUA_NOWAIT
No Wait for Data flag

LUA_SSCP_EXP
SSCP expedited flow

LUA_SSCP_NORM
SSCP normal flow

LUA_LU_EXP
LU expedited flow

LUA_LU_NORM
LU normal flow

LUA_CLOSE_ABEND

LUA_RESERVE1

lua_message_type
The type of SNA message received by an **RUI_READ** verb (or indicated to an **RUI_BID** verb). This can be one the following values:

LUA_MESSAGE_TYPE_LU_DATA
LUA_MESSAGE_TYPE_SSCP_DATA
LUA_MESSAGE_TYPE_RSP
LUA_MESSAGE_TYPE_BID
LUA_MESSAGE_TYPE_BIND
LUA_MESSAGE_TYPE_BIS
LUA_MESSAGE_TYPE_CANCEL
LUA_MESSAGE_TYPE_CHASE
LUA_MESSAGE_TYPE_CLEAR
LUA_MESSAGE_TYPE_CRV
LUA_MESSAGE_TYPE_LUSTAT_LU
LUA_MESSAGE_TYPE_LUSTAT_SSCP

LUA_MESSAGE_TYPE_QC
LUA_MESSAGE_TYPE_QEC
LUA_MESSAGE_TYPE_RELQ
LUA_MESSAGE_TYPE_RQR
LUA_MESSAGE_TYPE_RTR
LUA_MESSAGE_TYPE_SBI
LUA_MESSAGE_TYPE_SHUTD
LUA_MESSAGE_TYPE_SIGNAL
LUA_MESSAGE_TYPE_SDT
LUA_MESSAGE_TYPE_STSN
LUA_MESSAGE_TYPE_UNBIND

lua_flag2
Specifies flags for messages returned by LUA. (Refer to *Systems Network Architecture: Formats* for more information.) The flags can be one or more of the following values ORed together:

LUA_BID_ENABLE
Bid Enable indicator

LUA_ASYNC
Asynchronous verb completion flag

LUA_SSCP_EXP
SSCP expedited flow

LUA_SSCP_NORM
SSCP normal flow

LUA_LU_EXP
LU expedited flow

LUA_LU_NORM
LU normal flow

lua_encr_decr_option
Cryptography option.

RUI_BID Data Structure

The following parameter is specific to and only supplied on the **RUI_BID** verb:

lua_peek_data
Up to 12 bytes of data waiting to be read.

RUI_BID

The **RUI_BID** verb is used by the application to indicate when a received message is waiting to be read. This enables the application to determine what data, if any, is available before issuing the **RUI_READ** verb. When a message is available, the **RUI_BID** verb returns with details of the message flow on which it was received, the message type, the TH and RH of the message, and up to 12 bytes of message data. The main difference between **RUI_BID** and **RUI_READ** is that **RUI_BID** enables the application to check the data without removing it from the incoming message queue, so it can be left and accessed at a later stage. **RUI_READ** removes the message from the queue, so once the application has read the data it must process it.

Supplied Parameters

The application supplies the following parameters:

lua_verb
LUA_VERB_RUI

lua_verb_length
The length in bytes of the LUA verb record. Set this to

```
sizeof(struct LUA_COMMON) + 12.
```

lua_opcode
LUA_OPCODE_RUI_BID

lua_correlator
Optional. A 4-byte value, which you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
The name in ASCII of the local LU used by the session. This must match the LU name of an active LUA session.

This parameter is required only if the **lua_sid** parameter is zero. If a session ID is supplied in **lua_sid**, LUA does not use this parameter.

This parameter must be 8 bytes long; pad on the right with spaces, 0x20, if the name is shorter than 8 characters.

lua_sid
The session ID of the session. This must match a session ID returned on a previous **RUI_INIT** verb. This parameter is optional; if you do not specify the session ID, you must specify the LU name for the session in the **lua_luname** parameter.

lua_post_handle
This is a 4-byte handle that is used to post the completion of asynchronous verbs.

Returned Parameters

The following parameter will always be returned:

lua_flag2
This is only set to LUA_ASYNC if the verb completed asynchronously.

Other returned parameters depend on whether the verb completed successfully; see the following sections.

If the verb completed successfully, the following parameters are returned:

lua_prim_rc
: LUA_OK

lua_sid
: If the application specified the **lua_luname** parameter when issuing this verb, rather than specifying the session ID, LUA supplies the session ID.

lua_max_length
: The number of bytes of data in the received message.

lua_data_length
: The number of bytes of data returned in the **lua_peek_data** parameter; from 0 to 12.

lua_th Information from the transmission header (TH) of the received message.

lua_rh Information from the request/response header (RH) of the received message.

lua_message_type
: Message type of the received message, which will be one of the following values:

  LUA_MESSAGE_TYPE_LU_DATA

  LUA_MESSAGE_TYPE_SSCP_DATA

  LUA_MESSAGE_TYPE_RSP

  LUA_MESSAGE_TYPE_BID

  LUA_MESSAGE_TYPE_BIND

  LUA_MESSAGE_TYPE_BIS

  LUA_MESSAGE_TYPE_CANCEL

  LUA_MESSAGE_TYPE_CHASE

  LUA_MESSAGE_TYPE_CLEAR

  LUA_MESSAGE_TYPE_CRV

  LUA_MESSAGE_TYPE_LUSTAT_LU

  LUA_MESSAGE_TYPE_LUSTAT_SSCP

  LUA_MESSAGE_TYPE_QC

  LUA_MESSAGE_TYPE_QEC

  LUA_MESSAGE_TYPE_RELQ

  LUA_MESSAGE_TYPE_RTR

  LUA_MESSAGE_TYPE_SBI

  LUA_MESSAGE_TYPE_SHUTD

  LUA_MESSAGE_TYPE_SIGNAL

  LUA_MESSAGE_TYPE_SDT

  LUA_MESSAGE_TYPE_STSN

  LUA_MESSAGE_TYPE_UNBIND

lua_flag2
: One of the following flags will be set to indicate which message flow the data was received on:

LUA_SSCP_EXP
SSCP expedited flow

LUA_LU_EXP
LU expedited flow

LUA_SSCP_NORM
SSCP normal flow

LUA_LU_NORM
LU normal flow

lua_peek_data
The first 12 bytes of the message data (or all of the message data if it is shorter than 12 bytes).

The following return codes indicate that the verb did not complete successfully because it was canceled by another verb:

lua_prim_rc
LUA_CANCELLED

lua_sec_rc
LUA_TERMINATED

An **RUI_TERM** verb was issued while this verb was pending.

The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
LUA_PARAMETER_CHECK

lua_sec_rc
Possible values:

LUA_BID_ALREADY_ENABLED
The **RUI_BID** verb was rejected because a previous **RUI_BID** verb was already outstanding. Only one **RUI_BID** can be outstanding at a time.

LUA_RESERVED_FIELD_NOT_ZERO
A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID
The value of the **lua_verb_length** parameter was less than the length of the verb record required for this verb.

The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
LUA_STATE_CHECK

lua_sec_rc
LUA_NO_RUI_SESSION

An **RUI_INIT** verb has not yet completed successfully for this session, or a session outage has occurred.

The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
LUA_UNSUCCESSFUL

lua_sec_rc
LUA_INVALID_PROCESS

The application instance that issued this verb was not the same as the one that issued the **RUI_INIT** verb for this session.

The following return code indicates that Communications Server detected an error in the data received from the host. Instead of passing the received message to the application on an **RUI_READ** verb, Communications Server discards the message (and the rest of the chain if it is in a chain), and sends a negative response to the host. LUA informs the application on a subsequent **RUI_READ** or **RUI_BID** verb that a negative response was sent.

lua_prim_rc
LUA_NEGATIVE_RSP

lua_sec_rc
The secondary return code contains the sense code sent to the host on the negative response. See "SNA Layers" on page 152 for information on interpreting the sense code values that can be returned.

A zero secondary return code indicates that, following a previous **RUI_WRITE** of a negative response to a message in the middle of a chain, Communications Server has now received and discarded all messages from this chain.

The following primary and secondary return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc
LUA_SESSION_FAILURE

The session has been brought down.

lua_sec_rc
Possible values:

LUA_LU_COMPONENT_DISCONNECTED
The LUA session has failed because of a problem with the communications link or with the host LU.

LUA_RUI_LOGIC_ERROR
This return code indicates one of the following things:

- The host system has violated SNA protocols.
- An internal error was detected within LUA.

Attempt to reproduce the problem with tracing active, and check that the host is sending correct data.

lua_prim_rc
LUA_INVALID_VERB

Either the **lua_verb** parameter or the **lua_opcode** parameter was not valid. The verb did not execute.

lua_prim_rc
LUA_UNEXPECTED_DOS_ERROR

An operating system error occurred, such as resource shortage.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

Comments

The **RUI_INIT** verb must complete successfully before this verb can be issued.

Only one **RUI_BID** can be outstanding at any one time. After the **RUI_BID** verb has completed successfully, it can be reissued by setting the **lua_flag1** to LUA_BID_ENABLE on a subsequent **RUI_READ** verb. If the verb is to be reissued in this way, the application program must not free or modify the storage associated with the **RUI_BID** verb record.

If a message arrives from the host when an **RUI_READ** and an **RUI_BID** are both outstanding, the **RUI_READ** completes and the **RUI_BID** is left in progress.

Usage Notes

Each message that arrives will only be bid once. Once an **RUI_BID** verb has indicated that data is waiting on a particular session flow, the application should issue the **RUI_READ** verb to receive the data. Any subsequent **RUI_BID** will not report data arriving on that session flow until the message that was bid has been accepted by issuing an **RUI_READ** verb.

In general, the **lua_data_length** parameter returned on this verb indicates only the length of data in **lua_peek_data**, not the total length of data on the waiting message (except when a value of less than 12 is returned). The **lua_max_length** parameter returns the number of bytes in the received message. The application should ensure that the data length on the **RUI_READ** verb that accepts the data is sufficient to contain the message.

RUI_INIT

The **RUI_INIT** verb establishes the SSCP-LU session for a given LUA LU.

Supplied Parameters

The application supplies the following parameters:

lua_verb
LUA_VERB_RUI

lua_verb_length
The length in bytes of the LUA verb record. Set this to sizeof(struct LU_COMMON).

lua_opcode
LUA_OPCODE_RUI_INIT

lua_correlator
Optional. A 4-byte value, which you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
The name in ASCII of the local LU or LU pool that you want to start the session. This must match a configured LUA LU name or LU pool name. For applications on the Communications Server, the name is used as follows:

If the name is the name of an LU that is not in a pool, Communications Server attempts to start the session using this LU.

If the name is the name of an LU pool, or the name of an LU within a pool, Communications Server attempts to start the session using the first avaliable LU from the pool. This field is an 8-byte ASCII string, padded with trailing space (0x20) characters if necessary.

For applications on a SNA API client, the name should match a configured LUA Session Name.

lua_post_handle
This is a 4-byte handle that is used to post the completion of asynchronous verbs.

lua_flag1
The application should set this to LUA_ASYNC_STATUS, to receive RUI_INIT_STATUS indications from Communications Server when processing the **RUI_INIT** verb. (RUI_INIT_STATUS messages are described in "RUI_INIT_STATUS" on page 210.)

lua_encr_decr_option
Session-level cryptography option. Communications Server accepts the following two values:

0	Session-level cryptography is not used.
128	Encryption and decryption are performed by the application program.

Any other value will result in the return code LUA_ENCR_DECR_LOAD_ERROR. (Values in the range 1 to 127, indicating user-defined encryption and decryption routines, are supported by OS/2 Communications Manager/2's LUA implementation but not by Communications Server.)

Returned Parameters

The following parameter will always be returned:

lua_flag2
This is only set to LUA_ASYNC if the verb completed asynchronously.

Note: **RUI_INIT** will always complete asynchronously, unless it returns an error such as LUA_PARAMETER_CHECK.

Other returned parameters depend on whether the verb completed successfully; see the following sections.

If the verb executes successfully, LUA returns the following parameters:

lua_prim_rc
LUA_OK

lua_sid
A session ID for the new session. This can be used by subsequent verbs to identify this session.

lua_luname
The name of the local LU used by the session. This is required if the application specified an LU pool and needs to know which LU in the pool has been used.

The following return codes indicate that the verb did not complete successfully because it was canceled by another verb:

lua_prim_rc
LUA_CANCELLED

lua_sec_rc
LUA_TERMINATED

An **RUI_TERM** verb was issued before the **RUI_INIT** had completed.

The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
LUA_PARAMETER_CHECK

lua_sec_rc
Possible values:

LUA_INVALID_LUNAME
The **lua_luname** parameter could not be found. Check that the LU name or LU pool name was defined in the *Communications Server System Management Programming* API.

LUA_RESERVED_FIELD_NOT_ZERO
A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID
The value of the **lua_verb_length** parameter was less than the length of the verb record required for this verb.

The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
LUA_STATE_CHECK

lua_sec_rc
LUA_DUPLICATE_RUI_INIT

The **lua_luname** parameter specified an LU name or LU pool name that is already in use by this application (or for which this application already has an **RUI_INIT** verb in progress).

The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
LUA_UNSUCCESSFUL

lua_sec_rc
Possible values:

LUA_COMMAND_COUNT_ERROR
The verb specified the name of an LU pool, or the name of an LU in a pool, but all LUs in the pool are in use.

LUA_ENCR_DECR_LOAD_ERROR
The verb specified a value for **lua_encr_decr_option** other than 0 or 128.

LUA_INVALID_PROCESS
The LU specified by the **lua_luname** parameter is in use by another process.

LUA_LINK_NOT_STARTED
The link to the host has not been started.

The following values for **lua_sec_rc** are Communications Server sense codes, and can be returned if **lua_prim_rc** is LUA_UNSUCCESSFUL (these values reflect the state of the LU):

`X10020000`
ACTPU has not been received. **RUI_INIT** will not activate the PU.

`X10100000`
ACTPU has not been received. **RUI_INIT** will activate the PU.

`X10110000`
ACTPU has been received. **ACTLU** has not been received. SSCP does not support self-defining dependent LU (SSDLU). **RUI_INIT** will activate the LU.

`X10120000`
ACTPU has been received. **ACTLU** has not been received. SSCP does support SSDLU. **RUI_INIT** will activate the LU.

The following primary and secondary return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc
LUA_SESSION_FAILURE

The session has been brought down.

lua_sec_rc
LUA_LU_COMPONENT_DISCONNECTED

The LUA session has failed because of a problem with the communications link or with the host LU.

lua_prim_rc
LUA_INVALID_VERB

Either the **lua_verb** parameter or the **lua_opcode** parameter was not valid. The verb did not execute.

lua_prim_rc
LUA_UNEXPECTED_DOS_ERROR

An operating system error occurred, such as resource shortage.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

Comments

This verb must be the first LUA verb issued for the session. Until this verb has completed successfully, the only other LUA verb that can be issued for this session is **RUI_TERM** (which will terminate a pending **RUI_INIT**). All other verbs issued on this session must identify the session using one of the following parameters from this verb.

- The session ID is returned to the application in the **lua_sid** parameter.
- The LU name is supplied by the application in the **lua_luname** parameter.

Usage Notes

The **RUI_INIT** verb completes after an **ACTLU** is received from the host. If necessary, the verb waits indefinitely. If an **ACTLU** has already been received prior to the **RUI_INIT** verb, LUA sends a **NOTIFY** to the host to inform it that the LU is ready for use.

Note: Neither the **ACTLU** nor **NOTIFY** is visible to the LUA application.

Once the **RUI_INIT** verb has completed successfully, this session uses the LU for which the session was started. No other LUA session (from this or any other application) can use the LU until the **RUI_TERM** verb is issued.

RUI_PURGE

The **RUI_PURGE** verb cancels a previous **RUI_READ**. An **RUI_READ** can wait indefinitely if it is sent without setting **lua_flag1** to LUA_NO WAIT (the immediate return option), and no data is available on the specified flow; **RUI_PURGE** forces the waiting verb to return (with the primary return code CANCELLED).

Supplied Parameters

The application supplies the following parameters:

lua_verb
LUA_VERB_RUI

lua_verb_length
The length in bytes of the LUA verb record. Set this to sizeof(struct LUA_COMMON).

lua_opcode
LUA_OPCODE_RUI_PURGE

lua_correlator
Optional. A 4-byte value, which you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
The name in ASCII of the local LU used by the session. This must match the LU name of an active LUA session.

This parameter is required only if the **lua_sid** parameter is zero. If a session ID is supplied in **lua_sid**, LUA does not use this parameter.

This parameter must be 8 bytes long; pad on the right with spaces, 0x20, if the name is shorter than 8 characters.

lua_sid
The session ID of the session. This must match a session ID returned on a previous **RUI_INIT** verb.

This parameter is optional; if you do not specify the session ID, you must specify the LU name for the session in the **lua_luname** parameter.

lua_data_ptr
A pointer to the **RUI_READ** LUA_VERB_RECORD that is to be purged.

lua_post_handle
This is a 4-byte handle that is used to post the completion of asynchronous verbs.

Returned Parameters

The following parameter will always be returned:

lua_flag2
This is only set to LUA_ASYNC if the verb completed asynchronously.

Other returned parameters depend on whether the verb completed successfully; see the following sections.

If the verb completed successfully, the following parameters are returned:

lua_prim_rc
LUA_OK

lua_sid
If the application specified the **lua_luname** parameter when issuing this verb, rather than specifying the session ID, LUA supplies the session ID.

The following return codes indicate that the verb did not complete successfully because it was canceled by another verb:

lua_prim_rc
LUA_CANCELLED

lua_sec_rc
LUA_TERMINATED

An **RUI_TERM** verb was issued while this verb was pending.

The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
LUA_PARAMETER_CHECK

lua_sec_rc
Possible values:

LUA_BAD_DATA_PTR
The **lua_data_ptr** parameter was set to zero.

LUA_RESERVED_FIELD_NOT_ZERO
A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID
The value of the **lua_verb_length** parameter was less than the length of the verb record required for this verb.

The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
LUA_STATE_CHECK

lua_sec_rc
Possible values:

LUA_SEC_RC_OK
A previous **RUI_PURGE** verb is still in progress on this session.

LUA_NO_RUI_SESSION
An **RUI_INIT** verb has not yet completed successfully for this session, or a session outage has occurred.

The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
LUA_UNSUCCESSFUL

lua_sec_rc
Possible values:

LUA_INVALID_PROCESS
The application instance that issued this verb was not the same as the one that issued the **RUI_INIT** verb for this session.

LUA_NO_READ_TO_PURGE
Either the **lua_data_ptr** parameter did not contain a pointer to an **RUI_READ** LUA_VERB_RECORD or the **RUI_READ** verb completed before the **RUI_PURGE** verb was issued.

The following primary and secondary return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc
LUA_SESSION_FAILURE

The session has been brought down.

lua_sec_rc
Possible values:

LUA_LU_COMPONENT_DISCONNECTED
The LUA session has failed because of a problem with the communications link or with the host LU.

LUA_RUI_LOGIC_ERROR
This return code indicates one of the following things:

- The host system has violated SNA protocols.
- An internal error was detected within LUA.

Attempt to reproduce the problem with tracing active, and check that the host is sending correct data.

lua_prim_rc
LUA_INVALID_VERB

Either the **lua_verb** parameter or the **lua_opcode** parameter was not valid. The verb did not execute.

lua_prim_rc
LUA_UNEXPECTED_DOS_ERROR

An operating system error occurred, such as resource shortage.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

Comments

This verb can only be used when an **RUI_READ** has been issued and is pending completion (that is, the primary return code is IN_PROGRESS). This verb should not be issued while another **RUI_PURGE** is in progress on this session.

RUI_INIT_STATUS

The application cannot issue **RUI_INIT_STATUS** indications. Communications Server sends this indication to the application during **RUI_INIT** processing to provide information about the state of the LU-SSCP session. **RUI_INIT_STATUS** indications are only sent if the application requests status information when issuing **RUI_INIT** (see “RUI_INIT” on page 203).

Supplied Parameters

The following parameters are set on the **RUI_INIT_STATUS** indication:

lua_verb
LUA_VERB_RUI

lua_verb_length
The length in bytes of the LUA verb record (set by Communications Server to sizeof(LUA_COMMON)).

lua_opcode
LUA_OPCODE_RUI_INIT_STATUS

lua_primary_rc
Contains the information about the state of the LU-SSCP session. Possible values are:

LUA_LINK_INACTIVE
The link to the host is not yet active.

LUA_PU_INACTIVE
ACTPU has not yet been received, or **DACTPU** has been received.

LUA_PU_ACTIVE
ACTPU has been received from the SSCP.

LUA_PU_REACTIVATED
ACTPU(COLD) has been received while the PU was active.

LUA_LU_INACTIVE
ACTLU has been rejected, or **DACTLU** has been received.

LUA_UNKNOWN
The LU-SSCP session is not active due to a Data Link Control link error.

lua_correlator
Communications Server uses the value specified on the **RUI_INIT** verb for this parameter.

lua_post_handle
This is a 4-byte handle that is used to post the completion of asynchronous verbs. Communications Server uses the value specified on the **RUI_INIT** verb for this parameter.

RUI_READ

The **RUI_READ** verb receives data or status information sent from the host to the application's LU. You can specify a particular message flow (LU normal, LU expedited, SSCP normal, or SSCP expedited) from which to read data, or you can specify more than one message flow. You can have multiple **RUI_READ** verbs outstanding, provided that no two of them specify the same flow.

Supplied Parameters

The application supplies the following parameters:

lua_verb
LUA_VERB_RUI

lua_verb_length
The length in bytes of the LUA verb record. Set this to sizeof(struct LUA_COMMON).

lua_opcode
LUA_OPCODE_RUI_READ

lua_correlator
Optional. A 4-byte value, which you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
The name in ASCII of the local LU used by the session. This must match the LU name of an active LUA session.

This parameter is required only if the **lua_sid** parameter is zero. If a session ID is supplied in **lua_sid**, LUA does not use this parameter.

This parameter must be 8 bytes long; pad on the right with spaces, 0x20, if the name is shorter than 8 characters.

lua_sid
The session ID of the session. This must match a session ID returned on a previous **RUI_INIT** verb.

This parameter is optional; if you do not specify the session ID, you must specify the LU name for the session in the **lua_luname** parameter.

lua_max_length
The length of the buffer supplied to receive the data (see **lua_data_ptr**).

lua_data_ptr
A pointer to the buffer supplied to receive the data.

lua_post_handle
This is a 4-byte handle that is used to post the completion of asynchronous verbs.

lua_flag1
The flags can be one or more of the following values ORed together:

- Set LUA_NOWAIT if you want the **RUI_READ** verb to return immediately whether or not data is available to be read, or do not set it if you want the verb to wait for data before returning.

- Set LUA_BID_ENABLE to reenable the most recent **RUI_BID** verb (equivalent to issuing **RUI_BID** again with exactly the same parameters as before), or do not set it if you do not want to reenable **RUI_BID**.

  Note: Re-enabling the previous **RUI_BID** reuses the LUA_VERB_RECORD originally allocated and does not permit the LUA_VERB_RECORD to be freed or modified.
- Set one or more of the following flags to indicate which message flow to read data from:

  LUA_SSCP_EXP
  SSCP expedited flow

  LUA_LU_EXP
  LU expedited flow

  LUA_SSCP_NORM
  SSCP normal flow

  LUA_LU_NORM
  LU normal flow

  If more than one flag is set, the highest-priority data available will be returned. The order of priorities (highest to lowest) is as follows:
  1. SSCP expedited
  2. LU expedited
  3. SSCP normal
  4. LU normal

  The equivalent flag will be set in **lua_flag2** to indicate which flow the data was read from (see “Returned Parameters”).

Returned Parameters

The following parameters will always be returned:

lua_flag2
LUA_ASYNC is set if the verb completes asynchronously (and not set if the verb completes synchronously).

LUA_BID_ENABLE is set if an **RUI_BID** was successfully reenabled (and not set if it was not reenabled).

Other returned parameters depend on whether the verb completed successfully; see the following sections.

If the verb executes successfully, LUA also returns the following parameters:

lua_prim_rc
LUA_OK

The following parameters are returned if the verb completes successfully. They are also returned if the verb returns with truncated data because the **lua_data_length** parameter supplied was too small.

lua_sid
If the application specified the **lua_luname** parameter when issuing this verb, rather than specifying the session ID, LUA supplies the session ID.

lua_data_length
The length of the data received. LUA places the data in the buffer specified by **lua_data_ptr**.

lua_th Information from the transmission header (TH) of the received message.

lua_rh Information from the request/response header (RH) of the received message.

lua_message_type
Message type of the received message, which will be one of the following values:

LUA_MESSAGE_TYPE_LU_DATA
LUA_MESSAGE_TYPE_SSCP_DATA
LUA_MESSAGE_TYPE_RSP
LUA_MESSAGE_TYPE_BID
LUA_MESSAGE_TYPE_BIND
LUA_MESSAGE_TYPE_BIS
LUA_MESSAGE_TYPE_CANCEL
LUA_MESSAGE_TYPE_CHASE
LUA_MESSAGE_TYPE_CLEAR
LUA_MESSAGE_TYPE_CRV
LUA_MESSAGE_TYPE_LUSTAT_LU
LUA_MESSAGE_TYPE_LUSTAT_SSCP
LUA_MESSAGE_TYPE_QC
LUA_MESSAGE_TYPE_QEC
LUA_MESSAGE_TYPE_RELQ
LUA_MESSAGE_TYPE_RTR
LUA_MESSAGE_TYPE_SBI
LUA_MESSAGE_TYPE_SHUTD
LUA_MESSAGE_TYPE_SIGNAL
LUA_MESSAGE_TYPE_SDT
LUA_MESSAGE_TYPE_STSN
LUA_MESSAGE_TYPE_UNBIND

lua_flag2 parameters
This will be set to one of the following values, to indicate which message flow the data was received on:

LUA_SSCP_EXP
SSCP expedited flow

LUA_LU_EXP
LU expedited flow

LUA_SSCP_NORM
SSCP normal flow

LUA_LU_NORM
LU normal flow

The following return codes indicate that the verb did not complete successfully because it was canceled by another verb or by an internal error:

lua_prim_rc
LUA_CANCELLED

lua_sec_rc
Possible values:

LUA_PURGED
This **RUI_READ** verb has been canceled by an **RUI_PURGE** verb.

LUA_TERMINATED
An **RUI_TERM** verb was issued while this verb was pending.

The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
LUA_PARAMETER_CHECK

lua_sec_rc
Possible values:

LUA_BAD_DATA_PTR
The **lua_data_ptr** parameter contained an incorrect value.

LUA_BID_ALREADY_ENABLED
The **lua_flag1** was set to LUA_BID_ENABLE to reenable an **RUI_BID** verb, but the previous **RUI_BID** verb was still in progress.

LUA_DUPLICATE_READ_FLOW
The flow flags on **lua_flag1** specified one or more session flows for which an **RUI_READ** verb was already outstanding. Only one **RUI_READ** at a time can be waiting on each session flow.

LUA_INVALID_FLOW
None of the **lua_flag1** flow flags was set. At least one of these flags must be set to indicate which flow or flows to read from.

LUA_NO_PREVIOUS_BID_ENABLED
The **lua_flag1** was set to LUA_BID_ENABLE, to reenable an **RUI_BID** verb, but there was no previous **RUI_BID** verb that could be enabled. (See “Comments” on page 216 for more information.)

LUA_RESERVED_FIELD_NOT_ZERO
A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID
The value of the **lua_verb_length** parameter was less than the length of the verb record required for this verb.

The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
LUA_STATE_CHECK

lua_sec_rc
LUA_NO_RUI_SESSION

An **RUI_INIT** verb has not yet completed successfully for this session, or a session outage has occurred.

The following primary return code indicates one of the following two cases, which can be distinguished by the secondary return code:

- Communications Server detected an error in the data received from the host. Instead of passing the received message to the application on an **RUI_READ** verb, Communications Server discards the message (and the rest of the chain if it is in a chain), and sends a negative response to the host. LUA informs the application on a subsequent **RUI_READ** or **RUI_BID** verb that a negative response was sent.
- The LUA application previously sent a negative response to a message in the middle of a chain. Communications Server has purged subsequent messages in this chain, and is now reporting to the application that all messages from the chain have been received and purged.

lua_prim_rc
LUA_NEGATIVE_RSP

lua_sec_rc
A nonzero secondary return code contains the sense code sent to the host on the negative response. This indicates that Communications Server detected an error in the host data, and sent a negative response to the host. See “SNA Layers” on page 152 for information on interpreting the sense code values that can be returned.

A zero secondary return code indicates that, following a previous **RUI_WRITE** of a negative response to a message in the middle of a chain, Communications Server has now received and discarded all messages from this chain.

The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
LUA_UNSUCCESSFUL

lua_sec_rc
Possible values:

LUA_DATA_TRUNCATED
The **lua_data_length** parameter was smaller than the actual length of data received on the message. Only **lua_data_length** bytes of data were returned to the verb; the remaining data was discarded. Additional parameters are also returned if this secondary return code is obtained.

LUA_NO_DATA
The **lua_flag1** was set to LUA_NOWAIT to indicate immediate return without waiting for data, and no data was currently available on the specified session flow or flows.

LUA_INVALID_PROCESS
The application instance that issued this verb was not the same as the one that issued the **RUI_INIT** verb for this session.

The following primary and secondary return codes indicate that the verb did not complete successfully for other reasons.

lua_prim_rc
LUA_SESSION_FAILURE

The session has been brought down.

lua_sec_rc
Possible values:

LUA_LU_COMPONENT_DISCONNECTED
The LUA session has failed because of a problem with the communications link or with the host LU.

LUA_RUI_LOGIC_ERROR
This return code indicates one of the following things:

- The host system has violated SNA protocols.
- An internal error was detected within LUA.

Try to reproduce the problem with tracing active, and check that the host is sending correct data.

lua_prim_rc
LUA_INVALID_VERB

Either the **lua_verb** parameter or the **lua_opcode** parameter was not valid. The verb did not execute.

lua_prim_rc
LUA_UNEXPECTED_DOS_ERROR

An operating system error occurred, such as resource shortage.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

Comments

The **RUI_INIT** verb must have completed successfully before this verb can be issued. While an existing **RUI_READ** is pending, you can issue another **RUI_READ** only if it specifies a different session flow or flows from pending **RUI_READs**; that is, you cannot have more than one **RUI_READ** outstanding for the same session flow.

The **lua_flag1** can only be set to LUA_BID_ENABLE if all of the following things are true:

- An **RUI_BID** has already been issued successfully and has completed.
- The storage allocated for the **RUI_BID** verb has not been freed or modified.
- No other **RUI_BID** is pending.

Usage Notes

If the data received is longer than the **lua_max_length** parameter, it will be truncated; only **lua_max_length** bytes of data will be returned. The primary and secondary return codes LUA_UNSUCCESSFUL and LUA_DATA_TRUNCATED will also be returned.

Once a message has been read using the **RUI_READ** verb, it is removed from the incoming message queue and cannot be accessed again.

Note: The **RUI_BID** verb can be used as a nondestructive read; that is, the application can use it to check the type of data available, but the data remains on the incoming queue and need not be used immediately.

Pacing can be used on the primary-to-secondary half-session (this is specified in the host configuration) to protect the Communications Server node from being flooded with messages. If the LUA application is slow to read messages, Communications Server delays the sending of pacing responses to the host in order to slow it down.

RUI_TERM

The **RUI_TERM** verb ends both the LU-LU session and the LU-SSCP session for a given LUA LU.

Supplied Parameters

The application supplies the following parameters:

lua_verb
LUA_VERB_RUI

lua_verb_length
The length in bytes of the LUA verb record. Set this to size of (struct LUA_COMMON).

lua_opcode
LUA_OPCODE_RUI_TERM

lua_correlator
Optional. A 4-byte value, which you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
The name in ASCII of the local LU used by the session. This must match the LU name of an active LUA session (or the LU name specified on an outstanding **RUI_INIT** verb).

This parameter is required only if the **lua_sid** parameter is zero. If a session ID is supplied in **lua_sid**, LUA does not use this parameter.

This parameter must be 8 bytes long; pad on the right with spaces, 0x20, if the name is shorter than 8 characters.

lua_sid
The session ID of the session. This must match a session ID returned on a previous **RUI_INIT** verb.

This parameter is optional; if you do not specify the session ID, you must specify the LU name for the session in the **lua_luname** parameter.

lua_post_handle
This is a 4-byte handle that is used to post the completion of asynchronous verbs.

Returned Parameters

The following parameter will always be returned:

lua_flag2
This is only set to LUA_ASYNC if the verb completed asynchronously.

Other returned parameters depend on whether the verb completed successfully; see the following sections.

If the verb executes successfully, LUA also returns the following parameter:

lua_prim_rc
LUA_OK

The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
LUA_PARAMETER_CHECK

lua_sec_rc
Possible values:

LUA_RESERVED_FIELD_NOT_ZERO
A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID
The value of the **lua_verb_length** parameter was less than the length of the verb record required for this verb.

The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
LUA_STATE_CHECK

lua_sec_rc
LUA_NO_RUI_SESSION

An **RUI_INIT** verb has not yet completed successfully for this session, or a session outage has occurred.

The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
LUA_UNSUCCESSFUL

lua_sec_rc
Possible values:

LUA_COMMAND_COUNT_ERROR
An **RUI_TERM** was already pending when the verb was issued.

LUA_INVALID_PROCESS
The application instance that issued this verb was not the same as the one that issued the **RUI_INIT** verb for this session.

The following primary and secondary return codes indicate that the verb did not complete successfully for other reasons.

lua_prim_rc
LUA_SESSION_FAILURE

The session has been brought down.

lua_sec_rc
Possible values:

LUA_LU_COMPONENT_DISCONNECTED
The LUA session has failed because of a problem with the communications link or with the host LU.

LUA_RUI_LOGIC_ERROR
This return code indicates one of the following things:

- The host system has violated SNA protocols.
- An internal error was detected within LUA.

Try to reproduce the problem with tracing active, and check that the host is sending correct data.

lua_prim_rc
LUA_INVALID_VERB

Either the **lua_verb** parameter or the **lua_opcode** parameter was not valid. The verb did not execute.

lua_prim_rc
LUA_UNEXPECTED_DOS_ERROR

An operating system error occurred, such as resource shortage.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

Comments

This verb can be issued at any time after the **RUI_INIT** verb has been issued (whether or not it has completed). If any other LUA verb is pending when **RUI_TERM** is issued, no further processing on the pending verb will take place, and it will return with a primary return code of LUA_CANCELLED.

After this verb has completed, no other LUA verb can be issued for this session.

RUI_WRITE

The **RUI_WRITE** verb sends an SNA request or response unit from the LUA application to the host, over either the LU-LU session or the LU-SSCP session.

Supplied Parameters

The application supplies the following parameters:

lua_verb
LUA_VERB_RUI

lua_verb_length
The length in bytes of the LUA verb record. Set this to sizeof(struct LUA_COMMON).

lua_opcode
LUA_OPCODE_RUI_WRITE

lua_correlator
Optional. A 4-byte value, which you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
The name in ASCII of the local LU used by the session. This must match the LU name of an active LUA session.

This parameter is required only if the **lua_sid** parameter is zero. If a session ID is supplied in **lua_sid**, LUA does not use this parameter.

This parameter must be 8 bytes long; pad on the right with spaces, 0x20, if the name is shorter than 8 characters.

lua_sid
The session ID of the session. This must match a session ID returned on a previous **RUI_INIT** verb.

This parameter is optional; if you do not specify the session ID, you must specify the LU name for the session in the **lua_luname** parameter.

lua_data_length
The length of the supplied data (see **lua_data_ptr**). When sending data on the LU normal flow, the maximum length is as specified in the **BIND** received from the host; for all other flows the maximum length is 256 bytes.

When sending a positive response, this parameter is normally set to zero. LUA will complete the response based on the supplied sequence number (see **lua_th.snf**). In the case of a positive response to a **BIND** or **STSN**, an extended response is permitted, so a nonzero value can be used.

When sending a negative response, set this parameter to the length of the SNA sense code (4 bytes), which is supplied in the data buffer (see **lua_data_ptr**).

lua_data_ptr
A pointer to the buffer containing the supplied data.

For a request, or a positive response that requires data, the buffer should contain the entire RU. The length of the RU must be specified in **data_length**.

For a negative response, the buffer should contain the SNA sense code.

lua_post_handle
This is a 4-byte handle that is used to post the completion of asynchronous verbs.

lua_th.snf
Required only when sending a response. The sequence number of the request to which this is the response.

lua_rh When sending a request, most of the **lua_rh** flags must be set to correspond to the RH (request header) of the message to be sent. Do not set LUA_PI and LUA_QRI; these will be set by LUA.

When sending a response, only the following two **lua_rh** flags are set:

LUA_RRI
Is set to indicate a response.

LUA_RI
Is not set for a positive response, or set for a negative response.

lua_flag1
Set one of the following flags to indicate which message flow the data is to be sent on:

LUA_LU_EXP
LU expedited flow

LUA_SSCP_NORM
SSCP normal flow

LUA_LU_NORM
LU normal flow

One and only one of the flags must be set.

Note: Communications Server does not permit applications to send data on the SSCP expedited flow (LUA_SSCP_EXP).

Returned Parameters

The following parameter will always be returned:

lua_flag2
This is only set to LUA_ASYNC if the verb completed asynchronously.

Other returned parameters depend on whether the verb completed successfully; see the following sections.

If the verb executes successfully, LUA also returns the following parameters:

lua_prim_rc
LUA_OK

lua_sid
If the application specified the **lua_luname** parameter when issuing this verb, rather than specifying the session ID, LUA supplies the session ID.

lua_th The completed TH of the message written, including the fields filled in by LUA. You might need to save the value of **lua_th.snf** (the sequence number) for correlation with responses from the host.

lua_rh The completed RH of the message written, including the fields filled in by LUA.

lua_flag2
This will be set to one of the following values to indicate which message flow the data was received on:

LUA_SSCP_EXP
SSCP expedited flow

LUA_LU_EXP
LU expedited flow

LUA_SSCP_NORM
SSCP normal flow

LUA_LU_NORM
LU normal flow

The following return codes indicate that the verb did not complete successfully because it was canceled by another verb:

lua_prim_rc
LUA_CANCELLED

lua_sec_rc
LUA_TERMINATED

The verb was canceled because an **RUI_TERM** verb was issued for this session.

The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
LUA_PARAMETER_CHECK

lua_sec_rc
Possible values:

LUA_BAD_DATA_PTR
The **lua_data_ptr** parameter contained an incorrect value.

LUA_DUPLICATE_WRITE_FLOW
An **RUI_WRITE** was already outstanding for the session flow specified on this verb (the session flow is specified by setting one of the **lua_flag1** flow flags). Only one **RUI_WRITE** at a time can be outstanding on each session flow.

LUA_INVALID_FLOW
lua_flag1 was set to LUA_SSCP_EXP, indicating that the message should be sent on the SSCP expedited flow. Communications Server does not permit applications to send data on this flow.

LUA_MULTIPLE_WRITE_FLOWS
More than one of the **lua_flag1** flow flags was set. One and only one of these flags must be set to indicate which session flow the data is to be sent on.

LUA_REQUIRED_FIELD_MISSING
This return code indicates one of the following cases:

- None of the **lua_flag1** flow flags was set. One and only one of these flags must be set.
- The **RUI_WRITE** verb was used to send a response, and the response required more data than was supplied.

LUA_RESERVED_FIELD_NOT_ZERO
A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID
The value of the **lua_verb_length** parameter was less than the length of the verb record required for this verb.

The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
LUA_STATE_CHECK

lua_sec_rc
Possible values:

LUA_MODE_INCONSISTENCY
The SNA message sent on the **RUI_WRITE** was not valid at this time. This is caused by trying to send data on the LU-LU session before the session is bound. Check the sequence of SNA messages sent.

LUA_NO_RUI_SESSION
An **RUI_INIT** verb has not yet completed successfully for this session, or a session outage has occurred.

The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
LUA_UNSUCCESSFUL

lua_sec_rc
Possible values:

LUA_FUNCTION_NOT_SUPPORTED
This return code indicates one of the following cases:

- **lua_rh** was set to LUA_FI (Format Indicator), but the first byte of the supplied RU was not a recognized request code.
- **lua_rh** was set to LUA_RH_NC (RU category specified the Network Control (NC) category); Communications Server does not permit applications to send requests in this category.

LUA_INVALID_PROCESS
The application instance that issued this verb was not the same as the one that issued the **RUI_INIT** verb for this session.

LUA_INVALID_SESSION_PARAMETERS
The application used **RUI_WRITE** to send a positive response to a **BIND** message received from the host. However, the Communications Server node cannot accept the **BIND** parameters as specified, and has sent a negative response to the host. See "SNA Layers" on page 152 for more information on the **BIND** profiles accepted by Communications Server.

LUA_RSP_CORRELATION_ERROR
When using **RUI_WRITE** to send a response, the **lua_th.snf** parameter (which indicates the sequence number of the received message being responded to) did not contain a valid value.

LUA_RU_LENGTH_ERROR
The **lua_data_length** parameter contained an incorrect value. When sending data on the LU normal flow, the maximum length is as specified in the **BIND** received from the host; for all other flows the maximum length is 256 bytes.

(any other value)
Any other secondary return code here is an SNA sense code indicating that the supplied SNA data was not valid or could not be sent. See “SNA Layers” on page 152 for information on interpreting the SNA sense codes that can be returned.

The following primary and secondary return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc
LUA_SESSION_FAILURE

The session has been brought down.

lua_sec_rc
Possible values:

LUA_LU_COMPONENT_DISCONNECTED
The LUA session has failed because of a problem with the communications link or with the host LU.

LUA_RUI_LOGIC_ERROR
This return code indicates one of the following things: The host system has violated SNA protocols. An internal error was detected within LUA.

Attempt to reproduce the problem with tracing active, and check that the host is sending correct data.

lua_prim_rc
LUA_INVALID_VERB

Either the **lua_verb** parameter or the **lua_opcode** parameter was not valid.
The verb did not execute.

lua_prim_rc
LUA_UNEXPECTED_DOS_ERROR

An operating system error occurred, such as resource shortage.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

Comments

The **RUI_INIT** verb must be issued successfully before this verb can be issued. While an existing **RUI_WRITE** is pending, you can issue a second **RUI_WRITE** only if it specifies a different session flow from the pending **RUI_WRITE**; that is, you cannot have more than one **RUI_WRITE** outstanding for the same session flow.

The **RUI_WRITE** verb can be issued on the SSCP normal flow at any time after a successful **RUI_INIT** verb. **RUI_WRITE** verbs on the LU expedited or LU normal flows are permitted only after a **BIND** has been received, and must abide by the protocols specified on the **BIND**.

Usage Notes

The successful completion of **RUI_WRITE** indicates that the message was queued successfully to the data link; it does not necessarily indicate that the message was sent successfully, or that the host accepted it. Pacing can be used on the secondary-to-primary half-session (this is specified on the **BIND**) to prevent the LUA application from sending more data than the local or remote LU can handle. If this is the case, an **RUI_WRITE** on the LU normal flow can be delayed by LUA and can take some time to complete.

Note: Communications Server does not permit applications to send data on the SSCP expedited flow (LUA_SSCP_EXP).

Chapter 14. SLI Entry Points

This chapter describes the procedure entry points for SLI.

The SLI DLL defines the following procedure entry points:

© Copyright IBM Corp. 1994, 1997

SLI()

Provides event notification for all **SLI** verbs.

Syntax

```
void WINAPI SLI (LUA_VERB_RECORD* vcb);
```

Parameter	**Description**
vcb	Supplied parameter; specifies the address of the verb control block.

Returned Values

The value returned in **lua_flag2.async** indicates whether asynchronous notification will occur. If the flag is set (nonzero), asynchronous notification will occur through event signallng. If the flag is not set, the request completed synchronously. Examine the primary return code and secondary return code for any error conditions.

Usage Notes

The application must provide a handle to an event in the *lua_post_handle* parameter of the verb control block. The event must be in the not-signaled state.

When the asynchronous operation is complete, the application is notified by the signaling of the event. Upon signaling of the event, examine the primary return code and secondary return code for any error conditions. **See also:** “WinSLI” on page 229 .

WinSLI

Provides asynchronous message notification for all SLI verbs.

Syntax

```
int WINAPI WinSLI (HWND hWnd,
                   LUA_VERB_RECORD* vcb);
```

Parameter	**Description**
hwnd	Window handle to receive completion message.
vcb	Pointer to verb control block.

Returned Values

The function returns a value indicating whether the request was accepted by the SLI for processing. A returned value of 0 indicates that the request was accepted and will be processed. A value other than 0 indicates an error. Possible error codes are as follows:

WLUAINVALIDHANDLE
The window handle provided is not valid.

The value returned in **lua_flag2.async** indicates whether asynchronous notification will occur. If the flag is set (nonzero), asynchronous notification will occur through a message posted to the application's message queue. If the flag is not set, the request completed synchronously. Examine the primary return code and secondary return code for any error conditions.

Usage Notes

Upon completion of the verb, the application's window *hWind* receives the message returned by **RegisterWindowMessage** with **WinSLI** as the input string. The **lParam** argument contains the address of the VCB being posted as complete. The *wParam* argument is undefined. It is possible for the request to be accepted for processing (the function call returns 0), but rejected later with a primary return code and secondary return code set in the VCB. Examine the primary return code and secondary return code for any error conditions.

See also:: "SLI()" on page 228.

WinSLICleanup()

Terminates and deregisters an application from the SLI API.

Syntax

```
BOOL WINAPI WinSLICleanup (void);
```

Returned Values

The return value indicates success or failure of the deregistration. If the value is not `0`, the application was successfully deregistered. If the value is `0`, the application was not deregistered.

Usage Notes

Use **WinSLICleanup** to deregister the SLI API, for example, to free up resources allocated to the specific application.

Using **WinSLICleanup** is not required.

WinSLIStartup()

Enables an application to specify the required version of the SLI API and to retrieve details of the API.

Syntax

```
int WINAPI WinSLIStartup (WORD wVersionRequired,
                          LUADATA* luadata);
```

Parameter
Description

wVersionRequired
Specifies the version of SLI API support required. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.

luadata
Returns version of SLI implementation.

Returned Values

The return value specifies whether the application was registered successfully and whether the SLI API can support the specified version number. If the value is `0`, it was registered successfully and the specified version can be supported. Otherwise, the return value is one of the following values:

WLUAVERNOTSUPPORTED
The version of SLI API support requested is not provided by this particular SLI API.

WLUAINVALID
The version requested could not be determined.

Usage Notes

Using **WinSLIStartup** is not required.

Chapter 15. SLI Verbs

This chapter contains the following information for each SLI verb:

- The purpose of the verb.
- Parameters supplied to and returned by SLI. The description of each parameter includes information on the valid values for that parameter, and any additional information necessary.
- Interactions with other verbs.
- Additional information describing the use of the verb.

Note: Parameters marked as *reserved* should always be set to zero.

SLI_BID

This verb tells an SLI application program that a message is pending to be read by SLI_RECEIVE or that status is presented. SLI_BID is used to preview the pending data so the application can formulate a strategy for receiving the data. When data or status arrives for the SLI application program, SLI_BID is posted if an eligible SLI_RECEIVE is not active. The application program issues an SLI_BID verb after the session opens successfully (or during the SLI_OPEN if the initiation type is primary with SSCP access) to indicate that the application program will use the bid mechanism.

Supplied Parameters

The application supplies the following parameters:

lua_verb
LUA_VERB_SLI

The verb-code indicator for the LUA verbs.

lua_verb_length
The length of the verb control block. This number must equal the length expected by the SLI for the SLI_BID verb.

lua_opcode
LUA_OPCODE_SLI_BID

The operation code for the verb.

lua_correlator
A value that links the verb with other user-supplied information. This parameter is not used by the LUA interface.

lua_luname
The local LU name in ASCII. If the name contains fewer than 8 characters, you must pad it with blanks. LUA examines this parameter only if **lua_sid** is 0. Using the **lua_luname** parameter on all verbs helps make debugging easier, especially when multiple LUs are configured.

lua_sid
The session ID returned by SLI_OPEN that identifies the session to be used. If this parameter is 0, the **lua_luname** parameter is used for identification.

lua_post_handle
This is a 4-byte handle that is used to post the completion of asynchronous verbs.

Returned Parameters

If the verb completed successfully, the following parameters are returned:

lua_prim_rc
The primary return code, set by the verb function.

lua_sec_rc
The secondary return code, set by the verb function.

lua_data_length
The length of the peek data received.

lua_peek_data
This parameter contains up to the first 12 bytes of RU data to be read. The length of the data returned in this parameter is in the **lua_data_length** parameter.

lua_th A 6-byte parameter that contains the SNA transmission header (TH) for the message.

lua_rh A 3-byte parameter that contains the SNA request/response header (RH) for the message.

lua_message_type
The type of SNA data and commands. The valid message types follow:

LUA_MESSAGE_TYPE_LU_DATA

LUA_MESSAGE_TYPE_SSCP_DATA

LUA_MESSAGE_TYPE_RSP

LUA_MESSAGE_TYPE_BID

LUA_MESSAGE_TYPE_BIND

LUA_MESSAGE_TYPE_BIS

LUA_MESSAGE_TYPE_CANCEL

LUA_MESSAGE_TYPE_CHASE

LUA_MESSAGE_TYPE_LUSTAT_LU

LUA_MESSAGE_TYPE_LUSTAT_SSCP

LUA_MESSAGE_TYPE_QC

LUA_MESSAGE_TYPE_QEC

LUA_MESSAGE_TYPE_RELQ

LUA_MESSAGE_TYPE_RTR

LUA_MESSAGE_TYPE_SBI

LUA_MESSAGE_TYPE_SIGNAL

LUA_MESSAGE_TYPE_STSN

The SLI receives and responds to the BIND and STSN requests through the LUA interface extension routines.

LU_DATA, LUSTAT_LU, LUSTAT_SSCP, and SSCP_DATA are not SNA commands.

lua_flag2
A 1-byte flag that contains bits used as output parameters. At verb completion, all bits that are not described by value are reserved and must be set to 0. The flag in the high-order half-byte follows:

lua_flag2.async
A flag that indicates that this verb completes asynchronously

The low-order half-byte contains flags that describe the message session and flow. One of the following flags is returned:

lua_flag2.sscp_exp
Specifies SSCP-expedited flow

lua_flag2.sscp_norm
Specifies SSCP-normal flow

lua_flag2.lu_exp
Specifies LU-expedited flow

lua_flag2.lu_norm
Specifies LU-normal flow

lua_prim_rc
The primary return code, set by the verb function. For details, see “Appendix B. LUA Verb Return Codes” on page 319.

lua_sec_rc
The secondary return code, set by the verb function. For details, see “Appendix B. LUA Verb Return Codes” on page 319.

Usage Notes

Only one **SLI_BID** can be active for each session. The application program can be bid once for each flow if the **SLI_BID** is reactivated, even if the data is not read. If the application program does not read the bid data, it is not bid again for that specific flow.

Issuing the **SLI_BID** verb initially enables the bid function. After the **SLI_BID** verb posts complete, the bid function is disabled. The bid function can be reenabled in one of two ways:

- By calling the SLI again with the address of an **SLI_BID** verb control block.
- By issuing an **SLI_RECEIVE** with the **lua_flag1.bid_enable** parameter set to 1. If **SLI_RECEIVE** with **lua_flag1.bid_enable** is issued, the SLI uses the address of the last-accepted **SLI_BID** verb control block as the active bid.

Note:

1. If multiple flows have data available when the **SLI_BID** is issued, the data returned by the **SLI_BID** is from the highest priority flow that has data. From highest to lowest, the priorities are:
   - SSCP–expedited
   - LU–expedited
   - SSCP–normal
   - LU–normal
2. If, following **SLI_BID** completion, the LUA application issues an **SLI_RECEIVE** with multiple **lua_flag1** flow flags set, the data read could be for a different flow from the data returned by the **SLI_BID**. This could happen if higher priority data arrived from the host between the time that the **SLI_BID** completed and the **SLI_RECEIVE** was issued.

   The LUA application can, however, guarantee that an **SLI_RECEIVE** reads the data for which it was just bid. It does so by setting only one of the **lua_flag1** flow flags in the control block for the **SLI_RECEIVE** verb, specifying the same flow as that returned in the **lua_flag2** field of the completed **SLI_BID**.

The **SLI_BID** completes as soon as an RU arrives. This RU could be the only RU in a chain, or it could be the first RU in a multiple-RU chain. At **SLI_BID** completion, a single element chain is the only time a complete chainis bid to the application.

If the **SLI_BID** completes with the first RU of a multiple-RU chain and the subsequent **SLI_RECEIVE** specifies the **lua_flag1.nowait** option, the **lua_flag1.nowait** option is ignored. The **SLI_RECEIVE** verb returns in progress and will complete asynchronously after all RUs in the chain arrive.

If status is available, the application must read it. Until the application reads the status by issuing an **SLI_BID** or **SLI_RECEIVE**, all other operations are rejected, except for:

- **SLI_SEND** verbs on the SSCP flow
- **SLI_CLOSE**

When the primary return code is STATUS, the only **SLI_BID** parameters returned are **lua_prim_rc, lua_sec_rc,** and **lua_sid**. If **SLI_BID** and **SLI_RECEIVE** are both active when status becomes available, only the **SLI_BID** is posted with the status. When the application program is bid for status, all information is presented and no **SLI_RECEIVE** is required.

When the value of the primary return code is STATUS, the possible values for the secondary return code are:

- READY

  Indicates the SLI session is now ready for processing all additional commands. The READY status is issued after a prior NOT_READY status was received.
- NOT_READY

  Indicates that a CLEAR command or an UNBIND command with a type value of X'02' or X'01' was received from the host. The SLI session is suspended.
  - When a CLEAR arrives, the session is suspended until an SDT command is received.
  - When an SNA UNBIND type X'02' (UNBIND with BIND forthcoming) arrives, the session is suspended until BIND, optional CRV and STSN, and SDT commands are received. Any user extension routines must be reentrant.
  - When an UNBIND type X'01' (UNBIND normal) arrives and the **SLI_OPEN** verb for this session specified an **lua_session_type** of LUA_SESSION_TYPE_DEDICATED, the session is suspended until BIND, optional CRV and STSN, and SDT commands are received. User extension routines provided to process these commands must be reentrant.

    After the CLEAR, UNBIND type X'02', or UNBIND type X'01' arrives, the application can send SSCP data before reading the NOT_READY status, and can both send and receive SSCP data after reading the NOT_READY status.
- SESSION_END_REQUESTED

  Indicates that a SHUTD command was received from the host. The host is requesting that the SLI application end the session as soon as convenient.

  When the application is ready to end the session, it should issue an **SLI_OPEN**.
- INIT_COMPLETE

  Indicates that an **RUI_INIT** verb completed during **SLI_OPEN** processing. This statusis returned only when the **SLI_OPEN lua_init_type** parameter is LUA_INIT_TYPE_PRIM_SSCP.

  After this status is received, the application can send and receive data on the SSCP-normal flow.

In addition to the return codes, additional SNA sense data can be returned if a request unit sent by the host application has been converted into an exception request (EXR). An EXR is indicated by having the **SLI_BID** complete with the following returned verb parameters values:

Parameter
  Set to

lua_prim_rc
OK (X'0000')

lua_sec_rc
OK (X'00000000')

lua_rh.rri
bit off (request unit)

lua_rh.sdi
bit on (sense data included)

Under these conditions, the request has been converted into an EXR and up to 7 bytes of information is returned in the **lua_peek_data** verb parameter. The format of the information in the **lua_peek_data** parameter is as follows:

- Bytes 0—3 contain sense data defining the error detected. If LUA converted the request into an EXR, the sense data is one of the following values:

Sense Data	**Value of byes 0 - 3**
LUA_MODE_INCONSISTENCY	X'08090000'
LUA_BRACKET_RACE_ERROR	X'080B0000'
LUA_BB_REJECT_NO_RTR	X'08130000'
LUA_RECEIVER_IN_TRANSMIT_MODE	X'081B0000'
LUA_CRYPTOGRAPHY_FUNCTION_INOP	X'08480000'
LUA_SYNC_EVENT_RESPONSE	X'10010000'
LUA_RU_DATA_ERROR	X'10020000'
LUA_RU_LENGTH_ERROR	X'10020000'
LUA_INCORRECT_SEQUENCE_NUMBER	X'20010000'

The information returned to bytes 4 through 6 in **lua_peek_data** contain up to the first 3 bytes of the original request unit.

SLI_CLOSE

This verb closes the SNA session. SLI_CLOSE terminates the

connection with the host application program and frees the resources that were used. The posting of SLI_CLOSE signifies that the LU-LU and the SSCP-LU communications have ended.

Supplied Parameters

The application supplies the following parameters:

lua_verb
LUA_VERB_SLI

The verb-code indicator for the LUA verbs.

lua_verb_length
The length of the verb control block. This number must equal the length expected by the SLI for the SLI_CLOSE verb.

lua_opcode
LUA_OPCODE_SLI_CLOSE

The operation code for this verb. For SLI_CLOSE.

lua_correlator
A value that an LUA application program can supply to help correlate this verb with other information that the program supplies. This parameter is not used by the LUA interface.

lua_luname
The local LU name in ASCII. If the name contains fewer than 8 characters, you must pad it with blanks. LUA examines this parameter only if **lua_sid** is 0. Using the **lua_luname** parameter on all verbs helps make debugging easier, especially when multiple LUs are configured.

lua_sid
The session ID returned by a successfully completed SLI verb that identifies the session to be used. If this parameter is 0, the **lua_luname** parameter is used for identification.

lua_post_handle
This is a 4-byte handle that is used to post the completion of asynchronous verbs.

lua_flag1.close_abend
Specifies whether the close is a close immediate (on) or a normal close (off).

Returned Parameters

If the verb completed successfully, the following parameters are returned:

lua_flag2.async
A flag that indicates that this verb completes asynchronously.

lua_prim_rc
The primary return code, set by the verb function. For details, see "Appendix B. LUA Verb Return Codes" on page 319.

lua_sec_rc
The secondary return code, set by the verb function. For details, see "Appendix B. LUA Verb Return Codes" on page 319.

Usage Notes

There are two types of **SLI_CLOSE**: close normal and close abend.

- Close Normal

  The close normal is identified when the **lua_flag1.close_abend** parmeter is set to 0. The close sequence can be secondary initiated or primary initiated. The close normal uses a SHUTD command for a primary initiated or primary initiated. The close normal uses a SHUTD command for a primary initiated close and sends an RSHUTD command for a secondary initiated close.

  If the host sends an UNBIND type X'02' (UNBIND with BIND forthcoming) during a primary or secondary initiated **SLI_CLOSE** normal, the session is not closed. The **SLI_CLOSE** verb completes with the CANCELED primary return code, RECEIVED_UNBIND_HOLD secondary return code. The application program should issue an **SLI_BID** or **SLI_RECEIVE** verb to return STATUS.

  If the host sends UNBIND type X'01' (normal UNBIND) during a primary or secondary initiated **SLI_CLOSE** Normal and the **SLI_OPEN** verb for this session specified and **lua_session_type** of LUA_SESSION_TYPE_DEDICATED, the session is not closed. The **SLI_CLOSE** verb completes with the CANCELED primary return code and the RECEIVED_UNBIND_NORMAL secondary return code. The application program should issue **SLI_BID** or **SLI_RECEIVE** to return STATUS.
- Close Abend

  The close abend is identified when the **lua_flag.close_abend** parameter is set to 1. The CLOSE_ABEND option tells the SLI to end the session immediately.

The following SNA commands can flow during the different types of close processing:

- **SLI_CLOSE** Normal
  - Secondary Initiated Close

    After the SLI application program issues an **SLI_CLOSE** verb with **lua_flag.close_abend** set to 0, the SLI performs the following processing:

    ```
    Writes the RSHUTD command
    Reads and processes the RSHUTD command response
    Reads and processes the CLEAR command (if required)
    Writes the CLEAR command response (if required)
    Reads and processes the UNBIND command
    Writes the UNBIND command response
    Stops the RUI session
    ```
  - Primary Initiated Close

    Reads the SHUTD comand and gives the application SESSION_END_REQUESTED status.

    After the SLI application program issues **SLI_CLOSE** with **lua_flag.close_abend** set to 0, the SLI performs the following processing:

    ```
    Writes the CHASE command
    Reads and processes the CHASE command response
    Writes the Shutdown Complete (SHUTC) command
    ```

Reads and processes the SHUTC command response
Reads and processes the CLEAR command (if required)
Writes the CLEAR command response (if required)
Reads and processes the UNBIND command
Writes the UNBIND command response
Stops the RUI session

– **SLI_CLOSE** Abend
  - After the SLI application program issues an **SLI_CLOSE** verb with **lua_flag1.close_abend** set to 1, the SLI stops the RUI session.

The completion of the **SLI_CLOSE** verb implies that the LU-LU session is unbound and that the SSCP was notified of no-session capability for the LU. After the **SLI_CLOSE** verb completes successfully, no other SLI command can be issued for the session except another **SLI_OPEN**. All pending commands are terminated when the **SLI_CLOSE** verb is received.

Notes:

1. Do not use this function to close sessions that are established using the RUI.
2. Before you issue and **SLI_CLOSE** Normal, be certain that all owed responses have been sent to the host. The SLI automatically changes the CLOSE type to ABEND if responses are owed.

   The CLOSE type might be automatically changed to ABEND if the LUA application program ignores data. It is good programming practice to use the **SLI_RECEIVE** verb to receive all data from the host. Otherwise, the SLI might assume that a response is owed, even if the data was an exception request, and change the CLOSE type to ABEND.

SLI_OPEN

This verb opens an SNA session for an application program that is requesting session-level communications on the link. The session-level function issues SNA commands on behalf of the application program to open the session. The LUA application program is simplified because SLI functions perform multiple RUI functions to establish the LU-LU session.

Supplied Parameters

The application supplies the following parameters:

lua_verb
LUA_VERB_SLI

The verb-code indicator for the LUA verbs.

lua_verb_length
The length of the verb control block. This number must equal the length expected by the SLI for the SLI_OPEN verb.

lua_opcode
LUA_OPCODE_SLI_OPEN

lua_correlator
A value that an LUA application program can supply to help correlate this verb with other information that the program supplies. This parameter is not used by the Windows LUA interface.

lua_luname
The local LU name in ASCII. If the name contains fewer than 8 characters, you must pad it with blanks.

This parameter is required by **SLI_OPEN**. Other verbs require this parameter only if the **lua_sid** parameter is zero; however, using the **lua_luname** parameter on all verbs helps make debugging easier, especially when multiple LUs are configured.

lua_data_length
The length of the unformatted LOGON or INITSELF data being sent.

lua_data_ptr
A pointer to the data buffer of the application. Because this buffer is used for data and SNA commands, the contents of the buffer are usually in EBCDIC.

This data buffer contains one of the following things:

- The user's SNA INITSELF request unit (RU) with all of the required application program data filled in if the lua_init_type parameter specifies secondary initiated with INITSELF. The INITSELF contains user information, such as the mode name and the PLU name. For more information, refer to *Systems Network Architecture Network Product Formats*.
- The LOGON message that is sent on the SSCP-normal flow when the **lua_init_type** parameter specifies secondary initiated with an unformatted LOGON message.
- If the session is primary initiated, this buffer is not used and the **lua_data_ptr** parameter must be 0.

lua_post_handle
If asynchronous notification is to be accomplished by events, **lua_post_handle** contains the handle of the event to be signaled.

lua_encr_decr_option
Cryptography is not supported.

lua_init_type
Defines how the LU-LU session is initialized by the Windows LUA interface. Valid values are:

LUA_INIT_TYPE_SEC_IS
Secondary-initiated; send the INITSELF command that is supplied in the data buffer of the OPEN

LUA_INIT_TYPE_SEC_LOG
Secondary-initiated with an unformatted LOGON message specified in the data buffer of the OPEN

LUA_INIT_TYPE_PRIM
Primary-initiated; wait on BIND

LUA_INIT_TYPE_PRIM_SSCP
Primary-initiated with SSCP access

lua_session_type
A value that defines how the SLI processes UNBIND type X'01', UNBIND normal. The valid values follow:

LUA_SESSION_TYPE_NORMAL
When an UNBIND normal is received from the primary logical unit, the SLI sends a positive response and issues **RUI_TERM** which causes a NOTIFY disabled to flow to the SSCP. The SSCP-LU flow is disabled. This is the default value for this parameter.

LUA_SESSION_TYPE_DEDICATED
When an UNBIND normal is received from the primary logical unit, the SLI sends a positive response and the SLI session is suspended until a new BIND, optional CRV and STSN, and SDT commands are received. In this case, the SLI does not issue **RUI_TERM** and NOTIFY disabled does not flow to the SSCP.

lua_wait
The number of seconds (up to a maximum of 65 535) for the SLI to wait before automatically retrying the transmission of the INITSELF or the LOGON message after the host sends any one of these messages:

- A negative response to the INITSELF or LOGON message and the secondary return code is one of the following values:
  - RESOURCE_NOT_AVAILABLE (X'08010000')
  - SESSION_LIMIT_EXCEEDED (X'08050000')
  - SSCP_LU_SESS_NOT_ACTIVE (X'0857nnnn' where *nnnn* is X'0002')
  - SESSION_SERVICE_PATH_ERROR (X'087Dnnnn' where *nnnn* is X'0000')
- A Network Services Procedure Error (NSPE) message
- A NOTIFY command, which indicates a procedure error

If the value of **lua_wait** is 0, no retries occur. This parameter applies only to sessions initiated by the SLU. If the PLU initiates the session, **lua_wait** is ignored.

lua_extension_list_offset
Specifies the offset from the start of the verb control block to the extension list of user-supplied DLLs. The value must be the beginning of a word boundary. If there is no extension list, the value must be set to zero.

lua_routine_type
The type of routine of the following module and procedure name. The valid entries follow:

lua_routine_type_bind
Bind routine

lua_routine_type_crv
Cryptography vector routine

Note: Encryption is not currently supported.

lua_routine_type_sdt
Start data traffic (SDT) routine

lua_routine_type_stsn
Set and test sequence numbers (STSN) routine

lua_routine_type_end
Ending delimiter for list of routines.

lua_module_name
Provides the user-supplied ASCII module name. The parameter can be up to eight characters in length, with the remaining bytes set to X'00'.

lua_procedure_name
Provides the user-supplied DLL procedure name, in ASCII. The parameter can be up to 32 characters in length, with the remaining bytes set to X'00'.

Returned Parameters

If the verb completed successfully, the following parameters are returned:

lua_flag2.async
A flag that indicates that this verb completes asynchronously.

lua_sid
The session ID that subsequent verbs use to identify the session to be used. The value of this parameter is valid only if the primary return code is OK or IN_PROGRESS. If the **SLI_OPEN** fails after having returned IN_PROGRESS, the session ID is no longer valid.

lua_prim_rc
The primary return code, set by the verb function. For details, see "Appendix B. LUA Verb Return Codes" on page 319.

lua_sec_rc
The secondary return code, set by the verb function. For details, see "Appendix B. LUA Verb Return Codes" on page 319.

Usage Notes

The SLI can perform the following session initialization tasks:

- Start the RUI session
- Write an INITSELF or an unformatted logon message (secondary initialization only).

- Read and process an INITSELF response or the response to the logon message (secondary initialization only).
- Read and verify a BIND command from the host.
- Write a BIND response.
- Read and process an UNBIND type X'02' or an UNBIND type X'01' if one is sent by the host.
- Write the UNBIND response and prepare to receive the subsequent BIND.
- Read and process the STSN command (if required).
- Write the STSN response (if required).
- Read and process the SDT command.
- Write the SDT response.
- Go to user routines to process BIND, STSN, and SDT commands when they are specified by the application program in the **SLI_OPEN** verb.

The **SLI_OPEN** verb handles all SNA message traffic through the response to the SDT command.

An application program issues an **SLI_OPEN** verb to select a particular defined LUA LU in the **lua_luname** parameter. This field is an ASCII string that should be padded with blanks.

The **lua_init_type** parameter tells the SLI how to establish the LU session. The following list describes the initialization options:

- Secondary Initialization with INITSELF

  Set the **lua_init_type** parameter to LUA_INIT_TYPE_SEC_IS for this option. With this option, the application program must supply the INITSELF command used in the **SLI_OPEN** verb because the INITSELF contains all of the session-specific information needed by the host, such as the mode name and the PLU name. The **lua_data_ptr** parameter gives the address of the INITSELF, and the **lua_data_length** parameter gives its length.
- Secondary Initialization with an Unformatted LOGON Message

  Set the **lua_init_type** parameter to LUA_INIT_TYPE_SEC_LOG for this option. In secondary initialization with an unformatted LOGON message, the **lua_data_ptr** parameter contains the address of the user's EBCDIC LOGON message of the length that is specified in the **lua_data_length** parameter.
- Primary Initialization

  Set the **lua_init_type** parameter to LUA_INIT_TYPE_PRIM for this option. In primary initialization, the SLU does nothing to start the session with the host. The **SLI_OPEN** remains IN_PROGRESS until the host starts the session with a BIND command and a subsequent SDT command.
- Primary Initialization with SSCP Access

  Set the **lua_init_type** parameter to LUA_INIT_TYPE_PRIM_SSCP for this option. In primary initialization with SSCP access, the SLI does not send commands to the host to start the session. Instead, the SLI allows the application program to issue **SLI_SEND** and **SLI_RECEIVE** verbs for SSCP-normal flow data to send INITSELF commands or LOGON messages and to receive their responses. With this option, the application program is not limited to one INITSELF or LOGON message as it is for the secondary initialization types. This is the only **SLI_OPEN** type that allows the application program to issue SLI verbs before the **SLI_OPEN** completes. After the **SLI_OPEN** verb is issued, the application program can issue an **SLI_BID** or an **SLI_RECEIVE** to get

INIT_COMPLETE status. This status tells the application program that it can begin to issue the **SLI_SEND** and **SLI_RECEIVE** verbs for SSCP-normal flow data.

The optional **lu_session_type** parameter tells the SLI how to process UNBIND type X’01’, UNBIND normal. This parameter takes effect after the **SLI_OPEN** verb passes initial parameter checking and stays in effect until **SLI_CLOSE** abend is issued or until the SLI issues **RUI_TERM**. The following list describes standard UNBIND and dedicated UNBIND processing:

- Standard UNBIND Normal Processing **SLI_CLOSE** Normal

  Set the **lua_session_type** parameter to LUA_SESSION_TYPE_NORMAL for this option. This is the default value. With this option, the SLI sends a positive response to an UNBIND Normal sent by the primary LU and issues **RUI_TERM**, which causes a NOTIFY Disabled to flow to the SSCP. These actions do the following things:

  – End the LU-LU session.

  – Indicate to the SSCP and the PLU that the SLU is unable to process new BINDs. New BINDs that are received are rejected.

  – Prevent data from flowing on the SSCP-LU session.

    The SLI will issue **RUI_TERM** when it receives any UNBIND except type X'02' (UNBIND with BIND forthcoming).

  – Dedicated UNBIND Normal Processing

    Set the **lua_session_type** parameter to LUA_SESSION_TYPE_DEDICATED for this option. With this option, the SLI sends a positive response to an UNBIND normal sent by the primary logical unit. However, the SLI does not issue **RUI_TERM**. The status of the SSCP-LU session is not changed (enabled). The SLI session is suspended until BIND, optional CRV and STSN, and SDT commands are received. An SLI session that is waiting for a new BIND can be terminated by issuing an **SLI_CLOSE** Abend.

    The SLI issues **RUI_TERM** when it receives any UNBIND except type X'02' or type X'01'.

    This option is useful when the primary LU is unable to send an UNBIND with BIND forthcoming, but expects this type of behavior when UNBIND normal is sent.

Application-Supplied BIND, SDT, or STSN Routines

- If the application program supplies BIND, SDT, or STSN routines, the DLL module names and procedure entry points are passed in the **SLI_OPEN** extension routine list. If the corresponding SNA request is received, these routines are called during the **SLI_OPEN**. If no BIND routine is supplied, the SLI does a limited amount of BIND checking and responds as needed. If an STSN routine is not supplied and an STSN request is received, the SLI issues a positive response to indicate that no information is available. If an SDT routine is not supplied and an SDT request is received, the SLI issues a positive response.

Posting

- The posting of the **SLI_OPEN** with OK in the **lua_prim_rc** parameter means that the **SLI_OPEN** completed successfully and that an LU-LU data flow session was established. After the session is opened successfully, the application program ca issue **SLI_SEND**, **SLI_RECEIVE**, **SLI_PURGE**, **SLI_BID**, or **SLI_CLOSE** verbs.

Session Recovery

- The SLI supplies limited session recovery for the application program. When any SLI verb completes with SESSION_FAILURE in the **lua_prim_rc** parameter, the application program must reissue the **SLI_OPEN**. In this situation, the program does not have to issue an **SLI_CLOSE** verb before it issues a new **SLI_OPEN** verb.

Terminating a Pending SLI_OPEN

- To terminate a pending **SLI_OPEN**, issue an **SLI_CLOSE** with **lua_flag1.close_abend** parameter set to 1.

SLI_PURGE

This verb purges an outstanding **SLI_RECEIVE**. **SLI_PURGE** might be needed by an application program that uses an **SLI_RECEIVE** verb with the WAIT option. For example, if the **SLI_RECEIVE** verb does not complete in a specified interval of time, the application program can issue **SLI_PURGE**. The application program supplies the address of the **SLI_RECEIVE** verb control block in the **lua_data_ptr** parameter to specify which **SLI_RECEIVE** to purge.

Supplied Parameters

The application supplies the following parameters:

lua_verb
LUA_VERB_SLI

The verb-code indicator for the LUA verbs.

lua_verb_length
The length of the verb control block. This number must equal the length expected by the SLI for the **SLI_PURGE** verb.

lua_opcode
LUA_OPCODE_SLI_PURGE

The operation code for the verb.

lua_correlator
A value that an LUA application program can supply to help correlate this verb with other information that the program supplies. This parameter is ignored by the LUA interface.

lua_luname
The local LU name in ASCII. If the name contains fewer than 8 characters, you must pad it with blanks. LUA examines this parameter only if **lua_sid** is 0. Using the **lua_luname** parameter on all verbs helps make debugging easier, especially when multiple LUs are configured.

lua_sid
The session ID, returned by **SLI_OPEN**, that identifies the session to be used. If this parameter is 0, the **lua_luname** parameter is used for identification.

lua_data_ptr

A pointer to the application program **SLI_RECEIVE** verb control block to be purged.

lua_post_handle
If asynchronous notification is to be accomplished by events, **lua_post_handle** contains the handle of the event to be signaled.

Returned Parameters

If the verb completes successfully, the following parameters are returned:

lua_flag2.async
A flag that indicates that this verb completes asynchronously.

lua_prim_rc
: The primary return code, set by the verb function. For details, see "Appendix B. LUA Verb Return Codes" on page 319.

lua_sec_rc
: The secondary return code, set by the verb function. For details, see "Appendix B. LUA Verb Return Codes" on page 319.

Usage Notes

If **SLI_RECEIVE** is purged successfully, **SLI_RECEIVE** ends with the CANCELED primary return code and the **SLI_PURGE** completes with the OK primary return code.

SLI_RECEIVE

This verb transfers data or a status code to the application program. SLI_RECEIVE also provides the current status of the session to the Windows LUA application.

An **SLI_RECEIVE** verb for an LU-LU session flow can only be issued on an opened session. If the **SLI_OPEN** initiation type is primary with SSCP access, the application program can issue an **SLI_RECEIVE** verb for SSCP-LU normal flow data even when an **SLI_OPEN** verb is pending.

Supplied Parameters

The application supplies the following parameters:

lua_verb
LUA_VERB_SLI

The verb-code indicator for the LUA verbs.

lua_verb_length
The length of the verb control block. This number must equal the length expected by the SLI for the SLI_RECEIVE verb.

lua_opcode
LUA_OPCODE_SLI_RECEIVE

lua_correlator
A value that an LUA application program can supply to help correlate this verb with other information that the program supplies. This parameter is ignored by the LUA interface.

lua_luname
The local LU name in ASCII. If the name contains fewer than 8 characters, you must pad it with blanks. LUA examines this parameter only if **lua_sid** is 0. Using the **lua_luname** parameter on all verbs helps make debugging easier, especially when multiple LUs are configured.

lua_sid
The session ID returned by SLI_OPEN that identifies the session to be used. If this parameter is 0, the **lua_luname** parameter is used for identification.

lua_max_length
The length of the buffer used to receive data.

lua_data_ptr
A pointer to the buffer where the SLI places data received from the host application. Because this buffer is used for data and SNA commands, the contents of the buffer are usually in EBCDIC.

lua_post_handle
For Windows NT, if asynchronous notification is to be accomplished by events, **lua_post_handle** contains the handle of the vent to be signaled.

lua_flag1.bid_enable
A flag that specifies whether the LUA should reuse the SLI_BID verb control block on behalf of the LUA application program.

lua_flag1.nowait
A flag that tells the SLI to post the SLI_RECEIVE verb with the return code NO_DATA when there is no data to be read. If the first RU of a multiple-RU

chain arrives and the **lua_flag1.nowait** option has been selected, the **lua_flag1.nowait** option is ignored. The SLI_RECEIVE verb returns IN_PROGRESS and completes asynchronously after all RUs of the chain arrive. If chaining is allowed, the **lua_flag1.nowait** option should not be used.

The lower-order half-byte of **lua_flag1** contains flags that describe the message session and flow. The flow flags describe the flow or flows on which the LUA application program can accept a message. At least one of the following flags must be set, but the set flags must not overlap flags that are set in another active **SLI_RECEIVE** verb.

lua_flag1.sscp_exp
A flag that specifies SSCP-expedited flow.

lua_flag1.sscp_norm
A flag that specifies SSCP-normal flow.

lua_flag1.lu_exp
A flag that specifies LU-expedited flow

lua_flag1.lu_norm
A flag that specifies LU-normal flow.

Returned Parameters

If the verb completed successfully, the following parameters are returned:

lua_data_length
The length of the data being received.

lua_th A 6-byte parameter that contains the SNA transmission header (TH) for the message.

lua_rh A 3–byte parameter that contains the SNA request/response header (RH) for the message.

lua_message_type
The type of SNA data and commands. When the SLI application program wants to send data, the application program must set this parameter. The valid message types follow:

LUA_MESSAGE_TYPE_LU_DATA
LUA_MESSAGE_TYPE_SSCP_DATA
LUA_MESSAGE_TYPE_RSP
LUA_MESSAGE_TYPE_BID
LUA_MESSAGE_TYPE_BIS
LUA_MESSAGE_TYPE_CANCEL
LUA_MESSAGE_TYPE_CHASE
LUA_MESSAGE_TYPE_LUSTAT_LU
LUA_MESSAGE_TYPE_LUSTAT_SSCP
LUA_MESSAGE_TYPE_QC
LUA_MESSAGE_TYPE_QEC
LUA_MESSAGE_TYPE_RELQ
LUA_MESSAGE_TYPE_RTR
LUA_MESSAGE_TYPE_SBI
LUA_MESSAGE_TYPE_SIGNAL

LU_DATA, LUSTAT_LU, LUSTAT_SSCP, and SSCP_DATA are not SNA commands.

lua_flag2.async
A flag that specifies that this verb completes asynchronously.

lua_flag2.sscp_exp
A flat that specifies SSCP-expedited flow.

lua_flag2.sscp_norm
A flag that specifies SSCP-normal flow.

lua_flag2.lu_exp
A flag that specifies LU-expedited flow.

lua_flag2.lu_norm
A flag that specifies LU-normal flow.

lua_prim_rc
The primary return code, set by the verb function. For details, see "Appendix B. LUA Verb Return Codes" on page 319.

lua_sec_rc
The secondary return code, set by the verb function. For details, see "Appendix B. LUA Verb Return Codes" on page 319.

Usage Notes

SLI_RECEIVE receives responses, SNA commands, and request unit data from the host. **SLI_RECEIVE** also provides the status of the session to the Windows LUA application. An **SLI_OPEN** request must complete before **SLI_RECEIVE** can be issued. However, if **SLI_OPEN** is issued with lua_init_type set to LUA_INIT_TYPE_PRIM_SSCP, an **SLI_RECEIVE** over the SSCP normal flow may be issued as soon as **SLI_OPEN** returns an IN_PROGRESS.

Data is received by the application in one of four session flows. The four session flows, from highest to lowest priority are:

- SSCP expedited
- LU expedited
- SSCP normal
- LU normal

The data flow type that **SLI_RECEIVE** verb will process is specified in **lua_flag1**. The application can also specify whether it wants to look at more than one type of data flow. When multiple flow bits are set, the highest priority is received first. When **SLI_RECEIVE** completes processing, **lua_flag2** indicates the specific type of flow for which data has been received by the Windows LUA application.

If SLI_BID successfully completes before **SLI_RECEIVE** is issued, the Windows LUA interface can be instructed to reuse the last SLI_BID 's verb control block. To do this, issue **SLI_RECEIVE** with the **lua_flag1.bid_enable** parameter set to 1.

When using **lua_flag1.bid_enable** parameter, the **SLI_BID** storage must not be freed because the last **SLI_BID** verb's verb control block is used. Also, when using the **lua_flag1.bid_enable** parameter, the successful completion of **SLI_BID** will be posted.

If **SLI_RECEIVE** is issued with **lua_flag1.nowait** when no data is available to receive, LUA_NO_DATA will be the secondary return code set by the Windows LUA interface.

If status is available, the application must read it. Until the application reads the status by issuing an **SLI_BID** or **SLI_RECEIVE**, all other operations are rejected, except for:

- **SLI_SEND** verbs on the SSCP flow
- **SLI_CLOSE**

When the primary return code is STATUS, the only **SLI_RECEIVE** parameters returned are **lua_prim_rc, lua_sec_rc,** and **lua_sid**. An active **SLI_RECEIVE** verb can be posted witht he STATUS return code only when there is no active **SLI_BID** verb.

When the value of the primary return code is STATUS, the possible values for the secondary return code are:

- READY

  Indicates the SLI session is now ready for processing all additional commands. The READY status is issued after a prior NOT_READY status was received.
- NOT_READY

  Indicates that a CLEAR command or an UNBIND command with a type value of X'02' or X'01' was received from the host. The SLI session is suspended.
  - When a CLEAR arrives, the session is suspended until an SDT command is received.
  - When an UNBIND type X'02' (UNBIND with BIND forthcoming) arrives, the session is suspended until BIND, optional CRV and STSN, and SDT commands are received. Any user extension routines must be reentrant.
  - When an UNBIND type X'01' (UNBIND normal) arrives and the **SLI_OPEN** verb for this session specified an **lua_session_type** of LUA_SESSION_TYPE_DEDICATED, the session is suspended until BIND, optional CRV and STSN, and SDT commands are received. User extension routines provided to process these commands must be reentrant.

    After the CLEAR, UNBIND type X'02', or UNBIND type X'01' arrives, the application can send SSCP data before reading the NOT_READY status, and can both send and receive SSCP data after reading the NOT_READY status.
- SESSION_END_REQUESTED

  Indicates that a SHUTD command was received from the host. The host is requesting that the SLI application end the session as soon as convenient.

  When the application is ready to end the session, it should issue an **SLI_CLOSE** or an **SLI_CLOSE** Normal.
- INIT_COMPLETE

  Indicates that an **RUI_INIT** verb completed during **SLI_OPEN** processing. This status is returned only when the **SLI_OPEN lua_init_type** parameter is LUA_INIT_TYPE_PRIM_SSCP.

  After this status is received, the application can send and receive data on the SSCP-normal flow.

In addition to the return codes, additional SNA sense data can be returned if a request unit sent by the host application has been converted into an exception request (EXR). An EXR is indicated by having the **SLI_RECEIVE** complete with the following returned verb parameters values:

Parameter
Set to

lua_prim_rc
OK (X'0000')

lua_sec_rc
OK (X'00000000')

lua_rh.rri
bit off (request unit)

lua_rh.sdi
bit on (sense data included)

Under these conditions, the request has been converted into an EXR and up to 7 bytes of information is returned in the application buffer.. The format of the information in the data buffer is:

- Bytes 0—3 contain sense data defining the error detected. If LUA converted the request into an EXR, the sense data is one of the following values:

Sense Data	**Value of byes 0 - 3**
LUA_MODE_INCONSISTENCY	X'08090000'
LUA_BRACKET_RACE_ERROR	X'080B0000'
LUA_BB_REJECT_NO_RTR	X'08130000'
LUA_RECEIVER_IN_TRANSMIT_MODE	X'081B0000'
LUA_CRYPTOGRAPHY_FUNCTION_INOP	X'08480000'
LUA_SYNC_EVENT_RESPONSE	X'10010000'
LUA_RU_DATA_ERROR	X'10020000'
LUA_RU_LENGTH_ERROR	X'10020000'
LUA_INCORRECT_SEQUENCE_NUMBER	X'20010000'
LUA_LCC_NOT_SUPPORTED	X'20010000'

The information returned to bytes 4 through 6 in **lua_peek_data** contain up to the first 3 bytes of the original request unit.

SLI_SEND

This verb transfers, from the LUA application program to the communication link, user data, an SNA command, or an SNA response. **SLI_SEND** for an LU-LU session flow can only be issued on a previously opened session. If the **SLI_OPEN** initiation type is primary with SSCP access and INIT_COMPLETE status is achieved, the application program can issue **SLI_SEND** to transmit data on the SSCP-LU normal flow.

An LUA application can have two active **SLI_SEND** verbs simultaneously for each defined LUA LU. The two verbs can be for any two discrete flows.

Supplied Parameters

The application supplies the following parameters:

lua_verb
LUA_VERB_SLI

The verb-code indicator for the LUA verbs.

lua_verb_length
The length of the verb control block. This number must equal the length expected by the SLI for the **SLI_SEND** verb.

lua_opcode
LUA_OPCODE_SLI_SEND

The operation code for this verb.

lua_correlator
A value that an LUA application program can supply to help correlate this verb with other information that the program supplies. SLI ignores this parameter.

lua_luname
The local LU name in ASCII. If the name contains fewer than 8 characters, you must pad it with blanks. LUA examines this parameter only if **lua_sid** is 0. Using the **lua_luname** parameter on all verbs helps make debugging easier, especially when multiple LUs are configured.

lua_sid
The session ID returned by **SLI_OPEN** that identifies the session to be used. If this parameter is 0, the **lua_luname** parameter is used for identification.

lua_data_length
The length of the data being sent.

lua_data_ptr
A pointer to the application program data that is to be sent to the host application. Because this buffer is used for data and SNA commands, the contents of the buffer are usually in EBCDIC.

lua_post_handle
A 4-byte handle that is used to post the completion of asynchronous verbs.

lua_th.snf
The sequence number of the RU.

lua_rh A 3-byte parameter that contains the SNA request/response header (RH) for the message.

lua_message_type
The type of SNA data and commands. When the SLI application program wants to send data, the application program must set this parameter. For more information about the SNA commands, refer to *Systems Network Architecture Network Product Formats*. The valid message types follow:

LUA_MESSAGE_TYPE_BID
LUA_MESSAGE_TYPE_BIS
LUA_MESSAGE_TYPE_CANCEL
LUA_MESSAGE_TYPE_CHASE
LUA_MESSAGE_TYPE_LU_DATA
LUA_MESSAGE_TYPE_LUSTAT_LU
LUA_MESSAGE_TYPE_LUSTAT_SSCP
LUA_MESSAGE_TYPE_QC
LUA_MESSAGE_TYPE_QEC
LUA_MESSAGE_TYPE_RELQ
LUA_MESSAGE_TYPE_RQR
LUA_MESSAGE_TYPE_RSP
LUA_MESSAGE_TYPE_RTR
LUA_MESSAGE_TYPE_SBI
LUA_MESSAGE_TYPE_SSCP_DATA

lua_flag1.sscp_exp
Specifies SSCP-expedited flow

lua_flag1.sscp_norm
Specifies SSCP-normal flow

lua_flag1.lu_exp
Specifies LU-expedited flow

lua_flag1.lu_norm
Specifies LU-normal flow

Returned Parameters

If the verb executes successfully, LUA returns the following parameters:

lua_data_length
The length of the peek data received.

lua_th A 6-byte parameter that contains the SNA transmission header (TH) for the message.

lua_flag2.async
A flag that indicates that this verb completes asynchronously.

lua_flag2.sscp_exp
Specifies SSCP-expedited flow.

lua_flag2.sscp_norm
Specifies SSCP-normal flow.

lua_flag2.lu_exp
Specifies LU-expedited flow.

lua_flag2.lu_norm
Specifies LU-normal flow.

lua_sequence_number
The sequence number of the first-in-chain or the only-in-chain RU for the **SLI_SEND** verb. It is not byte-reversed.

lua_prim_rc
The primary return code, set by the verb function. For details, see “Appendix B. LUA Verb Return Codes” on page 319.

lua_sec_rc
The secondary return code, set by the verb function. For details, see “Appendix B. LUA Verb Return Codes” on page 319.

Usage Notes

SLI_SEND performs special processing based on the **lua_message_type** parameter, such as setting RH and TH bits and flow flags. For example, if the application sets the **lua_message_type** parameter to X'84' (CHASE), the SLI component automatically sets the **lua_rh** parameter to X'4B8000'. Table 19 shows the parameters that the application program should set if it is appropriate to do so, given the current program state.

Table 19. Parameter Settings Based on Message Type

	Value of lua_message_type parameter						
SLI_SEND parameter	**LU_DATA SSCP_DATA**	**RSP**	**BID, BIS, RTR**	**CHASE QC**	**QEC, RELQ, SBI, SIG**	**RQR**	**LUSTAT_LU LUSTAT_SSCP**
lua_rh	FI, DR1I, DR2I, RI, BBI, EBI, CDI, CSI, EDI	RI	SDI, QRI	SDI, QRI, EBI, CDI	SDI	0	SDI, QRI, DR1I, DR2I, RI, BBI, EBI, CDI
lua_th	0	SNF	0	0	0	0	0
lua_data_ptr	Required (0 if no data)	Required (0 if no data)	0	0	0	0	Required
lua_data_length	Required	Required (0 if no data)	0	0	0	0	Required
lua_flag1 flow flags	0	Required (set one)	0	0	0	0	0

An **SLI_SEND** verb transfers data from the location specified in the **lua_data_ptr** parameter for the length specified in the **lua_data_length**. The SLI chains data as needed. **SLI_SEND** can complete synchronously or asynchronously. When the application program returns from the call to the SLI, the **lua_flag2.async** flag indicates how the verb completes. When **lua_flag2.async** is set to ON, an IN_PROGRESS primary return code indicates that the verb was received and is in progress. A primary return code of OK indicates that the data or the command was written to the RUI. The application program receives the sequence number of the last chain element successfully sent using **RUI_WRITE** with synchronous return from the call to the SLI. After all chain elements are written, the application program receives the final return code and ending sequence number in the TH. These sequence numbers will differ if, for example, the SLI is sending a chain and has to wait for a pacing response from the host before the **SLI_SEND** operation can be completed.

When the SLI sends a response, the information required on the **SLI_SEND** verb depends on the type of response. For all responses, the application program must perform the following steps:

- Set the **lua_message_type** parameter to LUA_MESSAGE_TYPE_RSP
- Supply the sequence number (**lua_th.snf**) that corresponds to the request being responded to
- Set the selected **lua_flag1** flow flag

The rules for supplying additional parameters follow:

- For positive responses that require only the request code, the application program must also supply the following parameters:
  - **lua_rh.ri** set to 0
  - **lua_data_length** set to 0

  The SLI refers to the supplied sequence number to fill in the request code.
- For negative responses, the application program must also supply the following parameters:
  - **lua_rh.ri** set to 1
  - **lua_data_ptr** set to the address of an SNA sense code
  - **lua_data_length** set to the length of the SNA sense code (4 bytes).

  The SLI fills in the request code following the sense data.

SLI_BIND_ROUTINE

This verb tells an SLI application program that an SNA BIND request arrived from the host and allows the application program to examine the session protocols. The **SLI_BIND_ROUTINE** is passed to a programmer-supplied DLL specified in the **SLI_OPEN** extension list bind routine field.

Supplied Parameters

The following parameters for **SLI_BIND_ROUTINE** are supplied by the SLI:

lua_verb
LUA_VERB_SLI

The verb-code indicator for the LUA verbs.

lua_verb_length
The length of the verb control block.

lua_opcode
LUA_OPCODE_SLI_BIND_ROUTINE

The operation code for the routine.

lua_luname
The local LU name in ASCII.

lua_sid
The session ID returned by **SLI_OPEN** that identifies the session to be used.

lua_data_length
The length of the BIND RU.

lua_data_ptr
A pointer to the BIND RU. The BIND RU might contain EBCDIC characters such as the PLU name.

lua_th
The BIND TH.

lua_rh
The BIND RH.

Returned Parameters

If the verb completes successfully, LUA returns the following parameters:

lua_prim_rc
LUA_OK

lua_data_length
The length of the BIND response being sent.

lua_prim_rc
The primary return code, set by the verb function. For details, see "Appendix B. LUA Verb Return Codes" on page 319.

Usage Notes

The verb control block is built in the storage that is allocated by the SLI. The contents of the **lua_th** and **lua_rh** parameters are placed in the **SLI_BIND_ROUTINE** verb control block. The **lua_data_ptr** parameter contains the address of the BIND RU, and the **lua_data_length** parameter contains the length of the RU.

The **SLI_BIND_ROUTINE** is completed when the extension routine returns with the **lua_prim_rc** and the **lua_data_length** parameters set in the **SLI_BIND_ROUTINE** verb control block. Overwrite the BIND RU with the BIND response. A primary return code of OK indicates that the BIND was accepted. If the routine rejects the BIND, set the primary return code to NEGATIVE_RSP and put the negative sense code in the BIND buffer. Do not modify the **lua_data_ptr** parameter.

Note: A negative response from this routine cancels the **SLI_OPEN** verb. The SLI returns a primary return code of SESSION_FAILURE and a secondary return code of NEG_RSP_FROM_BIND_ROUTINE.

SLI_STSN_ROUTINE

This verb tells an SLI application program that an SNA STSN request arrived from the host and allows the application program to examine the STSN RU and prepare a response. The**SLI_STSN_ROUTINE** is passed to a programmer-supplied DLL that is specified in the **SLI_OPEN** extension list bind routine field.

Supplied Parameters

The following parameters for **SLI_STSN_ROUTINE** are supplied by the SLI:

lua_verb
LUA_VERB_SLI

The verb-code indicator for the LUA verbs.

lua_verb_length
The length of the verb control block.

lua_opcode LUA_OPCODE_SLI_STSN_ROUTINE
The operation code for the routine.

lua_luname
The local LU name in ASCII.

lua_sid
The session ID returned by **SLI_OPEN** that identifies the session to be used.

lua_data_length
The length of the STSN RU.

lua_data_ptr
A pointer to the STSN RU.

lua_th
The STSN TH.

lua_rh
The STSN RH.

Returned Parameters

If the verb executes successfully, LUA returns the following parameters:

lua_prim_rc
LUA_OK

lua_data_length
The length of the STSN response being sent.

lua_prim_rc
The primary return code, set by the verb function. For details, see "Appendix B. LUA Verb Return Codes" on page 319.

Usage Notes

The verb control block is built in the storage that is allocated by the SLI. The contents of the **lua_th** and **lua_rh** parameters are placed in the

SLI_STSN_ROUTINE

SLI_STSN_ROUTINE verb control block. The **lua_data_ptr** parameter contains the address of the **STSN** RU, and the **lua_data_length** parameter contains the length of the RU.

The **SLI_STSN_ROUTINE** is completed when the extension routine returns with the **lua_prim_rc** and the **lua_data_length** parameters set in the **SLI_STSN_ROUTINE** verb control block. Overwrite the STSN RU with the STSN response. A primary return code of OK indicates that the STSN was accepted. If the routine rejects the STSN, set the primary return code to NEGATIVE_RSP and put the negative sense code in the STSN buffer. Do not modify the **lua_data_ptr** parameter.

Note: A negative response from this routine cancels the **SLI_OPEN** verb. The SLI returns a primary return code of SESSION_FAILURE, and a secondary return code of NEG_RSP_FROM_STSN_ROUTINE.

SLI_SDT_ROUTINE

This verb tells an SLI application program that an SNA SDT request arrived from the host and allows the application program to examine the SDT RU and prepare a response. The **SLI_SDT_ROUTINE** is passed to a programmer-supplied DLL that is specified in the **SLI_OPEN** extension list bind routine field.

Supplied Parameters

The following parameters for SLI_SDT_ROUTINE are supplied by the SLI:

lua_verb
LUA_VERB_SLI

The verb-code indicator for the LUA verbs.

lua_verb_length
The length of the verb control block.

lua_opcode
LUA_OPCODE_SLI_STSN_ROUTINE

The operation code for the routine.

lua_luname
The local LU name in ASCII.

lua_sid
The session ID returned by **SLI_OPEN** that identifies the session to be used.

lua_data_length
The length of the SDT RU.

lua_data_ptr
A pointer to the SDT RU.

lua_th
The SDT TH.

lua_rh
The SDT RH.

Returned Parameters

Following is a list of the parameters for **SLI_SDT_ROUTINE** that the extension routine must return:

lua_prim_rc
LUA_OK

lua_data_length
The length of the SDT response being sent.

lua_prim_rc
The primary return code, set by the verb function. For details, see "Appendix B. LUA Verb Return Codes" on page 319.

Usage Notes

The verb control block is built in the storage that is allocated by the SLI. The contents of the **lua_th** and **lua_rh** parameters are placed in the **SLI_SDT_ROUTINE** verb control block. The **lua_data_ptr** parameter contains the address of the SDT RU, and the **lua_data_length** parameter contains the length of the RU.

The **SLI_SDT_ROUTINE** is completed when the extension routine returns with the **lua_prim_rc** and the **lua_data_length** parameters set in the **SLI_SDT_ROUTINE** verb control block. Overwrite the SDT RU with the SDT response. A primary return code of OK indicates that the SDT was accepted. If the routine rejects the SDT, set the primary return code to NEGATIVE_RSP and put the negative sense code in the STSN buffer. Do not modify the **lua_data_ptr** parameter.

Note: A negative response from this routine cancels the **SLI_OPEN** verb. The SLI returns a primary return code of SESSION_FAILURE, and a secondary return code of NEG_RSP_FROM_SDT_ROUTINE.

Part 3. Common Services API

© Copyright IBM Corp. 1994, 1997

Chapter 16. Entry Points

Communications Server provides a common services programming interface. This API consists of common services verbs (CSVs) that can be used by application programs that use Communications Server APIs.

Any Communications Server application program can use these common services verbs to do one or more of the following things:

- Maintain a code page translation table for single byte languages (**GET_CP_CONVERT_TABLE**)
- Convert an ASCII string to EBCDIC or EBCDIC to ASCII (**CONVERT**)
- Convert a double byte character string from one code page to another (**TRNSDT**)

Writing Common Services Programs

The table below shows source module usage of supplied header files and libraries needed to compile and link Common Services programs.

Table 20. Header Files and Libraries for Operating Systems

Operating System	Header File	Library	DLL Name
WINNT & WIN95	WINCSV.H	WINCSV32.LIB	WINCSV32.DLL
WIN3.1	WINCSV.H	WINCSV.LIB	WINCSV.DLL
OS/2	ACSSVC.H	ACSSVC.LIB	ACSSVC.DLL

The following sections describe the entry points for common services.

ACSSVC

This is the only entry point supported under OS/2.

This is a synchronous entry point for all CSV verbs. Communications Server provides this entry point for compatibility with existing applications.

Syntax

```
void ACSSVC (long)
```

Input is a verb control block pointer.

Returned Values

Check the primary and secondary return codes for returned values.

WinCSV()

This function provides a synchronous entry point for the CSV API.

Syntax

```
void WINAPI WinCSV(long vcb)
```

Parameter	**Description**
vcb	Pointer to verb control block.

Returned Values

No return value. The **primary_rc** and **secondary_rc** fields in the verb control block indicate any error.

Note: See also “WinAsyncCSV()” on page 271.

WinCSVCleanup()

This function terminates and deregisters an application from the CSV API.

Syntax

```
BOOL WINAPI WinCSVCleanup(void);
```

Returned Values

The return value specifies whether the deregistration was successful. If the value is not `0`, Communications Server successfully deregistered the application. Communications Server did deregister the application if the value is `0`.

Usage Notes

Use **WinCSVCleanup()** to deregister a CSV API application from the CSV API, for example, to free resources allocated to the specific application.

WinAsyncCSV()

This entry point is only for applications supported on the server.

The function provides an asynchronous entry point for **TRANSFER_MS_DATA** only. If an application uses this function for any other verb, the behavior is synchronous.

Syntax

```
HANDLE WlNAPI WinAsyncCSV(HHWND hWnd,
                          long vcb);
```

Parameter	**Description**
hwnd	Window handle to receive completion message.
vcb	Pointer to verb control block.

Returned Values

The return value indicates whether the asynchronous resolution request was successful. If the function was successful, the actual return value is an asynchronous task handle. If the function was not successful, Communications Server returns a `0`.

Usage Notes

Upon completion of the asynchronous operation, the application's window *hWnd* receives the message returned by **RegisterWindowMessage** with **WinAsyncCSV** as the input string. The *wParam* argument contains the asynchronous task handle returned by the original function call. The *lParam* argument contains the original VCB pointer and can be dereferenced to determine the final return code.

If the function returns successfully, Communications Server posts a **WinAsyncCSV()** message to the application when the operation completes or the conversation is canceled.

WinCSVStartup()

This function allows an application to specify the version of the Common Services Verbs API required and to retrieve details of the specific CSV API. This call is not required, but if used, the **WinCSVCleanup** call should be used also.

Syntax

```
int WINAPI WinCSVStartup (WORD wVersionRequired,
                          LPWCSVDATA csvdata);
```

Parameter
Description

wVersionRequired
Specifies the version of CSV API support required. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.

lpwCSVDATA
Contains information about the underlying CSV API DLL.

Returned Values

The return value indicates whether the CSV API successfully registered the application and whether it can support the provided version number. If the value returned is `0`, the CSV API does support the specified version and it successfully registered the application. Otherwise, one of the following values is returned.

WCSVVERNOTSUPPORTED
This particular CSV API does not provide the version of CSV API support requested.

WCSVINVALID
The CSV API could not determine the requested version.

Usage Notes

WinCSVStartup() is intended to help with compatibility with future releases of the API. The current version supported is 1.0.

The following structure describes details of the actual CSV API implementation.

```
typedef struct tagWCSVDATA { WORD wVersion;
                 char szDescription[WCSVDESCRIPTION_LEN+1];
                 } WCSVDATA, FAR *LPWCSVDATA;
```

When an application has made its last CSV API call, it calls **WinCSVCleanup()**.

GetCsvReturnCode()

Use this entry point to convert the primary and secondary return codes in the VCB to a printable string. It returns a standard set of error strings for use by application programs.

Syntax

```
int WINAPI GetCsvReturnCode (struct csv_hdr *vcb,
                             UINT buffer_length,
                             unsigned char *buffer_addr);
```

Parameter
Description

vcb The address of the verb control block.

buffer_length
The length of the buffer pointed to by **buffer_addr**. The recommended length is 256.

buffer_addr
The address of the buffer that will hold the formatted, null-terminated string (length of the string in the specified buffer).

Returned Values

0x20000001
The parameters are not valid; the function could not read from the specified **vcb** or could not write to the specified buffer.

0X20000002
The specified buffer is too small.

Usage Notes

The descriptive error string returned in **buffer_addr** does not terminate with a new line character (**\n**).

TrnsDt

This entry point is only supported for applications on the server.

This function converts the SBCS and DBCS strings from one code page to another. Communications Server provides **TrnsDt** in the **TRNSDT.DLL** file. **TransDt** is available only on a DBCS session.

Syntax

```
TrnsDt (PASSSTRUCT *passparm);
```

Function

This function converts the SBCS and DBCS strings from one code page to another. In the following table, "✓" indicates that Communications Server supports the conversion between the pair of code pages and "**-**" (hyphen) indicates that Communications Server does not support that conversion.

Japan		932	930	931	939	290	037	1027
	932	-	✓	✓	✓	✓	✓	✓
	930	✓	-	-	-	-	-	-
	931	✓	-	-	-	-	-	-
	939	✓	-	-	-	-	-	-
	290	✓	-	-	-	-	-	-
	037	✓	-	-	-	-	-	-
	1027	✓	-	-	-	-	-	-
Korea		949	833	834	933			
	949	-	✓	✓	✓			
	833	✓	-	-	-			
	834	✓	-	-	-			
	933	✓	-	-	-			
Taiwan		950	037	835	937			
	950	-	✓	✓	✓			
	037	✓	-	-	-			
	835	✓	-	-	-			
	937	✓	-	-	-			
China		1381	836	837	935			
	1381	-	✓	✓	✓			
	836	✓	-	-	-			
	837	✓	-	-	-			
	935	✓	-	-	-			

Use the header file **TRNSDT.H** to compile, and use the **TRNSDT.LIB** file from the Communications Server LIB subdirectory to link.

PassParm Format

The *passparm* format is as follows:

WORD *parm_length*
Length of this structure (input)

WORD *exit_code*
Exit code (output)

0000H Normal end.

0001H Not supported conversion specified.

000CH
Exit_code field is not initialized to `0`.

0080H The last character is the left half of a DCBS. Null character is filled instead.

WORD *in_length*
Length of the source buffer (input)

LPBYTE *in_addr*
Source buffer address (input)

WORD *out_length*
Length of target buffer (input)

If the specified length is too small to return all of the converted data, the required length is returned.

LPBYTE *out_addr*
Target address buffer (input)

WORD *trns_id*
Reserved to zero (input)

WORD *in_page*
Source code page (input)

WORD *out_page*
Target code page (input)

WORD *option*
Option (input/output)

Input Input options are as follows;

Bits 15–9
Reserved to zero

Bit 8 Target string has SO/SI

Bits 7–3
Reserved to zero

Bit 2 Use non-editable SBCS table

Bit 1 Source string starts with DBCS

Bit 0 Source string has SO/SI

Output

Output options are as follows:

4	End at DBCS
0	End at non-DBCS

Notes:

1. Bit 8 and Bit 0 should be set as follows:

   Conversion from PC to host Bit 8=1

   Conversion from PC to host Bit 0=0

   Conversion from host to PC Bit 8=0

   Conversion from host to PC Bit 0=1

2. Use **SYSCTBL.EXE** to specify the name of the customized table that **TrnsDt** uses. To convert an SBCS string, **TrnsDt** uses the customized table with the **Option** parameter bit 2 set to `FALSE`. **TrnsDt** uses the default table if bit 2 is set but the name of the table is not specified. To convert a DBCS string when the name of the table is specified using **SYSCTBL.EXE**, **TrnsDt** always uses the customized table. In this case, the **Option** parameter for bit 2 is not used.
3. Generally, **TrnsDt** requires that the host data include SO/SI control characters as a pair. However, to convert a part of a mixed data string, the data must start with a double-byte character without an SO control character. In this case, data does not identify the double-byte character. Bit 1 is useful in such a case. When you set bit 1 to 1, **TrnsDt** processes the start of the buffer as a double-byte character or SO control character.

Return Codes

0 NO_ERROR

2 ERROR_FILE_NOT_FOUND

TrnsDt cannot find the table used for converting the specified code.

87 ERROR_INVALID_PARAMETER

Parameter is not valid.

111 ERROR_BUFFER_OVERFLOW

The target buffer is too small.

150 ERROR_MEMORY_ALLOCATE

Memory allocation error.

Usage Notes

Even a small buffer can handle a large data conversion successfully by using the exit code and option parameters of **TrnsDt**. First, start **TrnsDt** using a small source buffer and a double- or triple-sized destination buffer (for cases from PC to host), and see how the conversion ends, based on the exit code you receive. Then proceed accordingly.

For example, when the conversion divides a double-byte character into two parts, or it ends incompletely between SO and SI control characters, define the buffer pointer and its position, then perform the next call.

VCB Structure

The following example translates the host code 0x4040 to PC code.

```
#include "trnsdt.h"

PASSSTRUCT      passparm;
char            bufs[20], buft[20];
int             rc;

//Setup the string to be translated
bufs[0] = 0x0e;
bufs[1] = 0x40;
bufs[2] = 0x40;
bufs[3] = 0x4f;

//Setup the parameter
passaparm.parm_length = 24;
passparm.exit_code    = 0;
passaparm.in_length   = 4;
passaparm.in_addr     = Created by ActiveSystems. 02/11/97. Entity not defined[0];
passaparm.out_length  = 20;
passaparm.out_addr    = Created by ActiveSystems. 02/11/97. Entity not defined[0];

 passaparm.trns_id    = 0;
 passaparm.in_page    = 930;
 passaparm.out_page   = 932;
 passaparm.option     = 1;

//Translate the string via TrnsDt
if (rc = TrnsDt(&passaparm))
   printf("Error Return Code = %d\n\r", rc);
   printf("Exit Code = %d\n\r", passaparm.exit_code);
   exit(0);
else
  .....
```

Chapter 17. Common Services Verbs (CSV)

Communications Server provides the following verbs for the Common Services API.

GET_CP_CONVERT_TABLE

This verb provides a utility service that builds a conversion table from one code page to another. This verb returns a 256-byte conversion table that applications can use to perform table lookups on characters to convert character strings.

A program might need to perform data conversion when it communicates with a node that expects data encoded in a different code page.

VCB Structure

```
struct get_cp_convert_table
  {
  unsigned short  opcode;          /* Verb identifying operation code.      */
  unsigned char   opext;           /* Reserved.                             */
  unsigned char   reserv2;         /* Reserved.                             */
  unsigned short  primary_rc;      /* Primary return code from verb.        */
  unsigned long   secondary_rc;    /* Secondary (qualifying) return code.   */
  unsigned short  source_cp;       /* Source code page for conversion table */
  unsigned short  target_cp;       /* Target code page for conversion table */
  unsigned char  *conv_tbl_addr;   /* Address to put conversion table at    */
  unsigned char   char_not_fnd;    /* Character not found option: either    */
                                   /* substitute character or round trip    */
  unsigned char   substitute_char; /* Substitute character to use.          */
  } GET_CP_CONVERT_TABLE;
```

Supplied Parameters

source_code_page
The code page number from which the replacement characters are drawn. The number for the code page can be one of the following numbers:

- ASCII code pages (in decimal)
  - 437 US IBM PC
  - 813 Greece
  - 819 ANSI Standard
  - 850 Multilingual
  - 852 Czechoslovakia/Hungary/Poland/Yugoslavia
  - 855 Cyrillic
  - 857 Turkey
  - 860 Portuguese
  - 861 Iceland
  - 862 Hebrew
  - 863 Canada-French
  - 864 Arabic
  - 865 Nordic
  - 866 Cyrillic
  - 874 Thai
  - 912 Latin 2
  - 915 Cyrillic
  - 920 Turkey
  - 1250 Latin 2
  - 1251 ANSI
- EBCDIC code pages (in decimal)

  - 037 United States/Canada-French/Netherlands/Portugal/Brazil
  - 273 Germany/Austria
  - 277 Denmark/Norway
  - 278 Finland/Sweden
  - 280 Italy
  - 284 Latin America/Spain
  - 285 United Kingdom
  - 297 France
  - 500 Belgium/Switzerland-French/Switzerland-German
  - 870 Czechoslovakia/Hungary/Poland/Yugoslavia
  - 871 Iceland
  - 875 Greece
  - 1025 Cyrillic
  - 1026 Turkey
- User defined code pages
  - 65280 through 65534
  - When using user-defined code pages, first define the registry entry with the user-defined path to the CPT files as follows:

```
HKEY_LOCAL_MACHINE/SOFTWARE/IBM/Communications Server/CurrentVersion/COMCPT
```

Note: Only identical characters in the source and target code pages are guaranteed to be converted into each other. Character pairs designated in the standards that merely resemble each other are not usually converted into each other.

target_code_page
The code page number for the target strings to be converted. The number can be any of those shown for **source_code_page**.

convert_table addr
The address of the buffer that is to receive the 256-byte conversion table. This buffer must be in a read/write segment.

character_not_found
The action to be taken if a character in the source code page does not exist in the target code page. Specify one of the following values:

SV_ROUND_TRIP
This option causes the values to be stored in the conversion table so that if a conversion table is generated by reversing the source and target code pages, the result of a conversion from source to target code page and back again results in the original character. You must select the **ROUND_TRIP** option for both table generations for this option to run.

SV_SUBSTITUTE
Store the character specified in the parameter **substitute_character** in the conversion table.

substitute_character
The byte stored in the conversion table if a character in the source code page does not exist on the target code page and if the **character_not_found** parameter is set to **SV_SUBSTITUTE**.

The OK return code indicates that the **GET_CP_CONVERT_TABLE** verb ran successfully.

The following parameter is returned when the return code is OK:

convert_table
The conversion table was built at the address specified by **CONV_table_addr**.

primary_rc
SV_PARAMETER_CHECK

secondary_rc
SV_INVALID_CHAR_NOT_FOUND

SV_INVALID_DATA_SEGMENT
SV_INVALID_SOURCE_CODE_PAGE
SV_INVALID_TARGET_CODE_PAGE

CONVERT

This verb converts ASCII character strings to EBCDIC and EBCDIC character strings to ASCII.

A program might perform data conversion when it communicates with a node that expects EBCDIC data or when it must convert names to pass over an interface, such as APPC, that requires EBCDIC names.

Note: The **CONVERT** verb is not supported by DBCS. You can use **TrnsDt** to convert strings that have double-byte characters.

VCB Structure

```
struct convert
  {
  unsigned short    opcode;        /* Verb identifying operation code.       */
  unsigned char     opext;         /* Reserved.                              */
  unsigned char     reserv2;       /* Reserved.                              */
  unsigned short    primary_rc;    /* Primary return code from verb.         */
  unsigned long     secondary_rc;  /* Secondary (qualifying) return code.    */
  unsigned char     direction;     /* Direction of conversion - ASCII to     */
                                   /* EBCDIC or vice-versa.                  */
  unsigned char     char_set;      /* Character to use for the conversion    */
                                   /* A, AE, or user-defined G.              */
  unsigned short    len;           /* Length of string to be converted.      */
  unsigned char    *source;        /* Pointer to string to be converted.     */
  unsigned char    *target;        /* Address to put converted string at.    */
  } CONVERT;
```

Comments

Returned Parameter Syntax

return_code OK error codes

Supplied Parameters

direction
The nature of the code conversion.

SV_ASCII_TO_EBCDIC
Converts ASCII characters to EBCDIC

SV_EBCDIC_TO_ASCII
Converts EBCDIC characters to ASCII

character_set
The set of characters permitted in the source string. You can specify three types of ASCII/EBCDIC conversion tables for use by the **CONVERT** verb: SV_A, SV_AE, and SV_G. The type-A and type-AE tables are defined within Communications Server. The type-G table is user-defined; its file name is specified on the Communications Server configuration menus. The type-G table can be used to convert any character.

The format of a conversion table consists of 32 lines of 32 characters each. Each line represents 16 printable hexadecimal characters followed by a carriage return and line feed. The first 16 lines provide the information for ASCII-to-EBCDIC conversion. The second 16 lines provide the information for EBCDIC-to-ASCII conversion. The table must include all 32 lines.

When Communications Server performs a conversion, it uses the numeric equivalent of each incoming character as a 0-origin index into the conversion table. This index specifies the table location containing the hexadecimal value of the converted character. For example, assume the 48th position in the table contains a value of X'F0'. Communications Server converts incoming characters with a value of 48 (X'30') to a value of 240 (X'F0').

Table A

Table A converts uppercase letters A through Z, numeric characters 0 through 9, and special characters $, #, and @. The first character of the source string must be either an uppercase letter or one of the three special characters; if it is not, no conversion is done, and the INVALID_FIRST_CHARACTER secondary return code is returned. In the ASCII-to-EBCDIC direction, lowercase ASCII characters are converted to uppercase EBCDIC characters.

Trailing blanks (blanks at the end of the source string) are converted to blanks in both directions. In contrast, embedded blanks are converted to X'00'.

If any source character is converted to X'00', CONVERSION_ERROR is returned. However, the entire conversion is completed.

Table AE

Table AE converts alphanumeric characters (A through Z, a through z, 0 through 9), special characters $, #, and @, and the period (.). There are no restrictions on the first character of the string.

Trailing blanks (blanks at the end of the source string) are converted to blanks in either direction. In contrast, embedded blanks are converted to X'00'.

If any source character is converted to X'00', CONVERSION_ERROR is returned. However, the entire conversion is completed.

Table G

You can use a G table to convert from any character to any other character (not just from ASCII to EBCDIC or EBCDIC to ASCII). However, you must specify ASCII_TO_EBCDIC on the **CONVERT** verb to use the top half of the table and specify EBCDIC_TO_ASCII to use the bottom half.

Communications Server will look in the registry under

```
HKEY_LOCAL_MACHINE/SOFTWARE/IBM/Communications Server/
                                CurrentVersion/COMTBLG
```

to get the full path name to the G table. The other supplied parameters for the **CONVERT** verb are:

length The number of characters to be converted.

The length of the string must not extend beyond the segment size allocated for **source_addr** or **target_addr**.

source_addr
The address of the character string converted.

target_addr address
The address receiving the converted character string.

Note: If the application does not require preservation of the source string, it can specify the same variable for **source_addr** and **target_addr**.

Returned Parameters

The OK return code indicates that the **CONVERT** verb ran successfully.

The following shows the primary and secondary error return codes associated with the **CONVERT** verb and the location of the return code's description.

primary_rc
SV_PARAMETER_CHECK

secondary_rc
SV_INVALID_DIRECTION

SV_TABLE_ERROR
SV_INVALID_CHARACTER_SET
SV_INVALID_FIRST_CHARACTER
SV_CONVERSION_ERROR
SV_INVALID_DATA_SEGMENT

primary_rc
SV_UNEXPECTED_DOS_ERROR

Part 4. EHNAPPC API

© Copyright IBM Corp. 1994, 1997

Chapter 18. EHNAPPC Application Program Interface

This is not available at the server.

The EHNAPPC Communications API provides a method to write cooperative processing applications between personal computers and AS/400 systems. It insulates the programmer from low-level communications programming and hardware connectivity types. Application programmers need to write both the AS/400 and PC programs when using this API. Almost anything that can be accessed by the host application can be extended to the partner PC application. This API can be used for performance-critical applications.

This chapter describes the routines, data structures, and return codes that make up the 32-bit EHNAPPC API for the Windows NT and Windows 95 Communications Server client. Most of these functions are also available in the 16-bit API for Windows 3.1

Writing EHNAPPC Programs

The table below shows source module usage of supplied header files and libraries needed to compile and link EHNAPPC programs.

Table 21. Header Files and Libraries for Operating Systems

Operating System	Header File	Library	DLL Name
WINNT & WIN95	E32APPC.H	E32APPC.LIB	E32APPC.DLL
WIN3.1	EHNAPPC.H	EHNAPPC.LIB	EHNAPPC.DLL

EHNAPPC Routines

The following discussions of each client Windows API routine describe in detail:

- Purpose
- Procedure declaration
- Parameters
- Return codes

EHNAPPC_Allocate

Purpose

This function starts a conversation with a partner transaction program.

Procedure Declaration

```
#include <WINDOWS.H>
include "E32APPC.H"
extern int EHNAPPC_Allocate
HWND                   hWnd,
unsigned                nBufferLength,
ConversationType        bType,
SyncLevelEnumb          SynchLevel,
LPSTR                   lpszLocationName,
LPSTR                   lpszTpn,
int                     nPipLength,
LPVOID                  lpPipData,
LPDWORD                 lpdwConversation);
```

Parameters

hWnd identifies the current window of the application.

bType identifies the type of conversation to allocate. Possible values are:

EHNAPPC_BASIC (0)

EHNAPPC_MAPPED (1)

bSynchLevel identifies the synchronization level between the local and partner programs. Possible values are:

EHNAPPC_SYNCLEVELNONE (0)

EHNAPPC_SYNCLEVELCONFIRM (1)

lpszLocationName points to a null-terminated character string that specifies the host system name. If this pointer is set to NULL, the default system is used.

lpszTpn points to a null-terminated character string that specifies the partner program name. If the first character is less than 0x40, then ASCII-to-EBCDIC translation is not done.

nPipLength identifies the length of the program initialization parameters (PIP) data. If this variable is 0, no PIP data is sent.

lpPipData points to the PIP data. The PIP data must be in GDS format, and must be in EBCDIC.

lpdwConversation points to a doubleword variable that is used to return a handle to be used on subsequent calls. The handle is a unique value for each conversation.

Return Codes

For return codes, see “Return Codes for the EHNAPPC API” on page 308

EHNAPPC_Confirm

Purpose

This function requests a confirmation that all data sent so far has been received by the partner.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern int far pascal EHNAPPC_Confirm(
HWND           hWnd,
DWORD          dwConversation,
LPBYTE         lpRequestToSendRcvd);
```

Parameters

hWnd identifies the current window of the application.

dwConversation identifies the conversation handle returned from either EHNAPPC_Allocate or EHNAPPC_ExtendedAllocate.

lpRequestToSendRcvd points to a variable which is used to store whether the partner transaction program issued a REQUEST_TO_SEND verb. A value of TRUE indicates the partner transaction program issued a REQUEST_TO_SEND verb.

Return Codes

For return codes, see "Return Codes for the EHNAPPC API" on page 308.

EHNAPPC_Confirmed

Purpose

This function sends a confirmation to a partner that has requested confirmation.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern int far pascal EHNAPPC_Confirmed(
     HWND          hWnd,
     DWORD         dwConversation);
```

Parameters

hWnd identifies the current window of the application.

dwConversation identifies the conversation handle returned from either EHNAPPC_Allocate or EHNAPPC_ExtendedAllocate.

Return Codes

For return codes, see "Return Codes for the EHNAPPC API" on page 308.

EHNAPPC_Deallocate

Purpose

This function deallocates an allocated conversation.

Procedure Declaration

```
#include "E32APPC.H"
extern int far pascal EHNAPPC_Deallocate(
         HWND                hWnd,
         DWORD               dwConversation,
         DeallocateEnum      bType);
```

Parameters

hWnd identifies the current window of the application.

dwConversation identifies the conversation handle returned from either EHNAPPC_Allocate or EHNAPPC_ExtendedAllocate.

bType identifies the type of deallocation the client is to perform. Possible values are:

EHNAPPC_DEALLOCATESYNCLEVEL (0)

EHNAPPC_DEALLOCATEFLUSH (1)

EHNAPPC_DEALLOCATEABEND (2)

Return Codes

For return codes, see "Return Codes for the EHNAPPC API" on page 308.

EHNAPPC_ExtendedAllocate

Purpose

This function starts a conversation with a partner transaction program and may override the security or mode specifications.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern int EHNAPPC_ExtendedAllocate(
HWND               hWnd,
unsigned           nBufferLength,
ConversationType   bType,
SyncLevelEnum      bSynchLevel,
LPSTR              lpszLocationName,
LPSTR              lpszTpn,
LPSTR              lpszModeName,
SecurityType       bSecurityType,
LPSTR              lpszUserId,
LPSTR              lpszPassword,
in                 nPipLength,
LPVOID             lpPipData,
LPDWORD            lpdwConversation);
```

Parameters

HWND identifies the current window of the application.

bType identifies the type of conversation to allocate. Possible values are:

EHNAPPC_BASIC (0)

EHNAPPC_MAPPED (1)

bSynchLevel identifies the synchronization level between the local and partner programs. Possible values are:

EHNAPPC_SYNCLEVELNONE (0)

EHNAPPC_SYNCLEVELCONFIRM (1)

lpszLocationName points to a null-terminated character string that specifies the host system name. If this pointer is set to NULL, the default system is used.

lpszTpn points to a null-terminated character string that specifies the partner program name. If the first character is less than X'40', then ASCII-to-EBCDIC translation is not done.

lpszModeName. The following are the rules for naming a mode:

Mode Names are one to eight characters long. The first character of each part must be an uppercase alphabetic character (A–Z), or on of the special characters (@, #, $). The remaining characters can be uppercase alphabetic characters (A–Z), numerals (0–9), or special characters (@, #, $).

bSecurityType identifies the security type to use. Possible values are:

EHNAPPC_SECURITY_NONE (0)

EHNAPPC_SECURITY_SAME (1)

EHNAPPC_SECURITY_PGM (2)

lpszUserId points to a null-terminated character string containing the user ID. The maximum length is 10 characters.

lpszPassword points to a null-terminated character string containing the password. The maximum length is 10 characters.

nPipLength identifies the length of the PIP data. If this variable is 0, no PIP data is sent.

lpPipData points to the PIP data. The PIP data must be in GDS format, and must be in EBCDIC.

lpdwConversation points to a doubleword variable which is used to return a handle to be used on subsequent calls.

Return Codes

For return codes, see "Return Codes for the EHNAPPC API" on page 308.

EHNAPPC_Flush

Purpose

This function causes the client to send any data it may have in its buffers.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern int EHNAPPC_Flush(
     HWND       hWnd,
     DWORD      dwConversation);
```

Parameters

hWnd identifies the current window of the application.

dwConversation identifies the conversation handle returned from either EHNAPPC_Allocate or EHNAPPC_ExtendedAllocate.

Return Codes

For return codes, see "Return Codes for the EHNAPPC API" on page 308.

EHNAPPC_GetAttributes

Purpose

Returns attributes of the specified conversation, including the LU names of the local and partner transaction programs, the level of processing synchronization, and any user ID provided for security.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern unsigned EHNAPPC_GetAttributes(
     HWND            hWnd,
     DWORD           dwConversation,
     LPBYTE          lpbSyncLevel,
     LPSTR           lpszModeName,
     LPSTR           lpszLuName,
     LPSTR           lpszPluName,
     LPSTR           lpszUserId);
```

Parameters

hWnd identifies the current window of the application.

dwConversation identifies the conversation handle returned by EHNAPPC_Allocate or EHNAPPC_Extended Allocate.

lpbSyncLevel points to a byte variable that is used to return the synchronization level.

lpszModeName points to a null-terminated character string that is used to return the 8- character mode name.

lpszLuName points to a null-terminated character string that is used to return the LU of the local trans action program.

lpszPluName points to a null-terminated character string that is used to return the name of the partner LU.

lpszUserId points to a null-terminated character string that is used to return the user ID that was used to establish this connection.

Return Codes

For return codes, see “Return Codes for the EHNAPPC API” on page 308.

EHNAPPC_GetCapabilities

Purpose

This function fills in a data structure indicating the capabilities of the client currently loaded.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern unsigned EHNAPPC_GetCapabilities(
     HWND      hWnd,
     LPSTR     lpList);
```

Parameters

hWnd identifies the current window of the application.

lpList points to a capabilities list that is used to retrieve the capability information. A capabilities list consists of a header followed by a variable number of capability structures. On input, the list specifies the capabilities to be queried. On output, it contains the capability information.

Note: For additional structure information, see “appcrtrcap_hdr” on page 306, “appcrtrcap_mult” on page 307 and “appcrtrcap_query” on page 307.

Return Codes

For return codes, see “Return Codes for the EHNAPPC API” on page 308.

EHNAPPC_GetDefaultSystem

Purpose

This function returns the default system name that the client is connected to.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern unsigned pascal EHNAPPC_GetDefaultSystem(
     HWND      hWnd,
     LPSTR     lpszDefSysName);
```

Parameters

hWnd identifies the current window of the application.

lpszDefSysName points to a character buffer that is used to return the default system name. The system name is stored in this buffer as a null- terminated character string.

Return Codes

For return codes, see “Return Codes for the EHNAPPC API” on page 308.

EHNAPPC_IsClientLoaded

Purpose

This function determines whether the client is loaded in memory.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern bool EHNAPPC_IsClientLoaded(
     HWND     hWnd);
```

Parameters

hWnd identifies the current window of the application.

Return Codes

The return code is FALSE (0) if the Communications Server client is not loaded. Otherwise, the return value is TRUE (1).

EHNAPPC_PrepareToReceive

Purpose

This function prepares the program to receive data. Using this function followed by EHNAPPC_ReceiveImmediate is the same as using EHNAPPC_ReceiveAndWait.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern int EHNAPPC_PrepareToReceive(
     HWND     hWnd,
     DWORD    dwConversation);
```

Parameters

hWnd identifies the current window of the application.

dwConversation identifies the conversation handle returned from either EHNAPPC_Allocate or EHNAPPC_ExtendedAllocate.

Return Codes

For return codes, see “Return Codes for the EHNAPPC API” on page 308.

EHNAPPC_QueryConfiguredSystems

Purpose

This function returns the names of the systems configured on the communications server.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern unsigned EHNAPPC_QueryConfiguredSystems(
     HWND              hWnd,
     LPINT             lpSysCount,
     LPSYSSTRUC        lpSys);
```

Parameters

hWnd identifies the current window of the application.

lpSysCount points to an integer variable which is used to return the number of systems connected.

lpSys points to an AS400_Sys structure that is used to return the names of the systems. The default system is the first system in the structure. For a description of the AS400_Sys structure, see "AS400_SYS" on page 306.

Return Codes

For return codes, see "Return Codes for the EHNAPPC API" on page 308.

EHNAPPC_QueryConvState

Purpose

This function returns the state of the specified conversation.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern unsigned pascal EHNAPPC_QueryConvState(
     HWND       hWnd,
     DWORD      dwConversation);
```

Parameters

hWnd identifies the current window of the application.

dwConversation identifies the conversation handle returned from either EHNAPPC_Allocate or EHNAPPC_ExtendedAllocate.

Return Codes

The return value indicates the current state of the conversation. Possible values are:

EHNAPPC_RESET_STATE (0)
EHNAPPC_SEND_STATE (1)
EHNAPPC_RECEIVE_STATE (2)
EHNAPPC_RCVD_CONF_STATE (3)
EHNAPPC_RCVD_CONF_SEND_STATE (4)
EHNAPPC_RCVD_CONF_DEALL_STATE (5)
EHNAPPC_PEND_DEALLOCATE_STATE (6)
EHNAPPC_INVALID_STATE (7)

EHNAPPC_QueryFullSystems

Purpose

This function returns the names and network names of the systems the client is connected to.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern unsigned EHNAPPC_QueryFullSystems(
     HWND            hWnd,
     LPINT           lpSysCount,
     LPFULLSYSSTRUC  lpSys);
```

Parameters

hWnd identifies the current window of the application.

lpSysCount points to an integer variable which is used to return the number of systems connected.

lpSys points to an AS400_Sys structure that is used to return the names of the systems.

Return Codes

For return codes, see “Return Codes for the EHNAPPC API” on page 308.

EHNAPPC_QueryUserid

Purpose

This function returns the user ID used to connect to the specified system.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern unsigned EHNAPPC_QueryUserId(
     HWND          hWnd,
     LPSTR         lpszLocationName,
     LPSTR         lpszUserId);
```

Parameters

hWnd identifies the current window of the application.

lpszLocationName points to a null-terminated character string containing the system name to be queried. Specify NULL to query the user ID for the default system. lpszUserId points to a null-terminated character string that is used to return the user ID for the specified system.

Return Codes

For return codes, see “Return Codes for the EHNAPPC API” on page 308.

EHNAPPC_QuerySystems

Purpose

This function returns the names of the systems the client is connected to.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern unsigned EHNAPPC_QuerySystems(
     HWND          hWnd,
     LPINT         lpSysCount,
     LPSYSSTRUC    lpSys);
```

Parameters

hWnd identifies the current window of the application.

lpSysCount points to an integer variable which is used to return the number of systems connected.

lpSys points to an AS400_Sys structure that is used to return the names of the systems. The default system is the first system in the structure. For a description of the AS400_Sys structure, see “AS400_SYS” on page 306.

Return Codes

For return codes, see “Return Codes for the EHNAPPC API” on page 308.

EHNAPPC_ReceiveAndWait

Purpose

This function waits for information to arrive on the conversation, then receives the information.

Procedure Declaration

```
#include "E32APPC.H"
extern int EHNAPPC_ReceiveAndWait(
     HWND          hWnd,
     DWORD         dwConversation,
     FillEnu       bFill,
     int           nMaxLength,
     LPVOID        lpReceiveData,
     LPBYTE        lpWhatReceived,
     LPBYTE        lpRequestToSendRcvd,
     LPWORD        lpReceiveDataLength );
```

Parameters

hWnd identifies the current window of the application.

dwConversation identifies the conversation handle returned from either EHNAPPC_Allocate or EHNAPPC_ExtendedAllocate.

bFill indicates the form in which the program is to receive data. Possible values are:

EHNAPPC_BUFFER (0) (fill the buffer)

EHNAPPC_LL (1) (receive a complete or truncated logical record)

nMaxLength indicates the largest amount of data that can be accepted.

lpReceiveData points to a buffer where the data is to be received.

lpWhatReceived indicates what has been received by the client. Possible values are:

- EHNAPPC_DATA (0)
- EHNAPPC_DATACOMPLETE (1)
- EHNAPPC_DATAINCOMPLETE (2)
- EHNAPPC_RECEIVEDCONFIRM (3)
- EHNAPPC_RECEIVEDCONFIRMSEND(4)
- EHNAPPC_RECEIVEDCONFIRMDEALLOC(5)
- EHNAPPC_RECEIVEDSEND (6)

lpRequestToSendRcvd points to a variable that is used to store whether the partner transaction program issued a REQUEST_TO_SEND verb. A value of TRUE (1) indicates the partner transaction program issued a REQUEST_TO_SEND verb.

lpReceiveDataLength points to a variable that is used to return the amount of data received by the client.

Return Codes

For return codes, see “Return Codes for the EHNAPPC API” on page 308.

EHNAPPC_ReceiveImmediate

Purpose

This function checks to see if something has been received. If so, the data is returned.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern int EHNAPPC_ReceiveImmediate(
       HWND        hWnd,
       DWORD       dwConversation,
       FillEnum    bFill,
       int         nMaxLength,
       LPVOID      lpReceiveData,
       LPBYTE      lpWhatReceived,
       LPBYTE      lpRequestToSendRcvd,
       LPWORD      lpReceiveDataLength );
```

Parameters

hWnd identifies the current window of the application.

dwConversation identifies the conversation handle returned from either EHNAPPC_Allocate or EHNAPPC_ExtendedAllocate.

bFill indicates the form in which the program is to receive data. Possible values are:

- EHNAPPC_BUFFER (0) (fill the buffer)
- EHNAPPC_LL (1) (receive a complete or truncated logical record)

nMaxLength indicates the largest amount of data that can be accepted.

lpReceiveData points to a buffer where the data is to be received.

lpWhatReceived identifies what has been received by the client. Possible values are:

EHNAPPC_DATA (0)

EHNAPPC_DATACOMPLETE (1)

EHNAPPC_DATAINCOMPLETE (2)

EHNAPPC_RECEIVEDCONFIRM (3)

EHNAPPC_RECEIVEDCONFIRMSEND (4)

EHNAPPC_RECEIVEDCONFIRMDEALLOC (5)

EHNAPPC_RECEIVEDSEND (6)

lpRequestToSendRcvd points to a variable which is used to store whether the partner transaction program issued a REQUEST_TO_SEND verb. A value of TRUE (1) indicates the partner transaction program issued a REQUEST_TO_SEND verb.

lpReceiveDataLength points to a variable that is used to return the amount of data received by the client.

Return Codes

For return codes, see "Return Codes for the EHNAPPC API" on page 308.

EHNAPPC_RemoteProgramStart

Purpose

This function allows Windows applications to start a program on a remote AS/400 system.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern word EHNAPPC_RemoteProgramStart(
    HWND          hWnd,
    LPSTR         lpszHostSystemName,
    LPSTR         lpszHostProgramName,
    LPSTR         lpszHostLibraryName,
    char FAR      *lpchPipData,
    WORD          wPipDataLength);
```

Parameters

hWnd identifies the current window of the application.

lpszHostSystemName points to a null-terminated character string that contains the name of the remote system. The maximum length of this string is 8 characters. If this pointer is null, the default system name is used.

lpszHostProgramName points to a null-terminated character string that contains the name of the host program to be started.

lpszHostLibraryName points to a null-terminated character string that contains the library path of the host program. If this pointer is null, the library list of the user is searched.

lpchPipData points to the program initialization parameter (PIP) data area for the host program. If this pointer is null, no PIP data is sent.

wPipDataLength contains the length of the PIP data.

Return Codes

For return codes, see “Return Codes for the EHNAPPC API” on page 308.

EHNAPPC_RqsToSend

Purpose

This function requests that the partner give up control of the conversation. The client places the conversation in send state when the local transaction program subsequently receives EHNAPPC_RECEIVEDSEND (6) in the lpWhatReceived parameter of a Receive verb from the partner transaction program.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern int EHNAPPC_RqsToSend(
     HWND      hWnd,
     DWORD     dwConversation);
```

Parameters

hWnd identifies the current window of the application.

dwConversation identifies the conversation handle returned from either EHNAPPC_Allocate or EHNAPPC_ExtendedAllocate.

Return Codes

For return codes, see “Return Codes for the EHNAPPC API” on page 308.

EHNAPPC_SendData

Purpose

This function sends data to the partner transaction program.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern int EHNAPPC_SendData(
     HWND      hWnd,
     DWORD     dwConversation,
     int       nSendDataLength,
     LPVOID    lpSendDataBuffer,
     LPBYTE    lpRequestToSendRcvd);
```

Parameters

hWnd identifies the current window of the application.

dwConversation identifies the conversation handle returned from either EHNAPPC_Allocate or EHNAPPC_ExtendedAllocate.

nSendDataLength identifies the length of the data in the send buffer.

lpSendDataBuffer identifies the address of the send buffer.

lpRequestToSendRcvd points to a variable that is used to store whether the partner transaction program issued a REQUEST_TO_SEND verb. A value of TRUE indicates the partner trans action program issued a REQUEST_TO_SEND verb.

Return Codes

For return codes, see "Return Codes for the EHNAPPC API" on page 308.

EHNAPPC_SendError

Purpose

This function indicates to the partner transaction program that some error has been found. After using this function, the local program is in receive state.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern int EHNAPPC_SendError(
     HWND       hWnd,
     DWORD      dwConversation,
     LPBYTE     lpRequestToSendRcvd);
```

Parameters

hWnd identifies the current window of the application.

dwConversation identifies the conversation handle returned from either EHNAPPC_Allocate or EHNAPPC_ExtendedAllocate.

lpRequestToSendRcvd points to a variable that is used to store whether the partner transaction program issued a REQUEST_TO_SEND verb. A value of TRUE indicates the partner trans action program issued a REQUEST_TO_SEND verb.

Return Codes

For return codes, see "Return Codes for the EHNAPPC API" on page 308.

EHNAPPC_StartHostProgram

Purpose

This function allows Windows applications to start a program on a remote AS/400 system, leaving the conversation active allowing the application to confirm the host program is running. The application will have to use the EHNAPPC_Deallocate function to end the conversation.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern word EHNAPPC_RemoteProgramStart(
     HWND       hWnd,
     LPSTR      lpszHostSystemName,
     LPSTR       lpszHostProgramName,
     LPSTR       lpszHostLibraryName,
     char FAR    *lpchPipData,
     WORD        wPipDataLength);
```

Parameters

hWnd identifies the current window of the application.

lpszHostSystemName points to a null-terminated character string that contains the name of the remote system. The maximum length of this string is 8 characters. If this pointer is null, the default system name is used.

lpszHostProgramName points to a null-terminated character string that contains the name of the host program to be started.

lpszHostLibraryName points to a null-terminated character string that contains the library path of the host program. If this pointer is null, the library list of the user is searched.

lpchPipData points to the program initialization parameter (PIP) data area for the host program. If this pointer is null, no PIP data is sent.

wPipDataLength contains the length of the PIP data.

Return Codes

For return codes, see “Return Codes for the EHNAPPC API” on page 308.

EHNAPPC Structures

AS400_SYS

Purpose

This structure is used to store the names of the systems the client is connected to.

Procedure Declaration

```
struct AS400_sys
     (
unsigned char EHNAPPC_SysName¢EHNAPPC_MAX_SYSTEMS|
     ¢EHNAPPC_SYSNAME_SYSNAME_LENGTH|;
     );
```

Parameters

EHNAPPC_SysName is used to store the name of a connected system. System names are returned as null-terminated strings. The first system returned in the array is the default system (EHNAPPC_MAX_SYSTEMS = 32 and EHNAPPC_SYSNAME_SYSNAME_LENGTH = 10).

appcrtrcap_hdr

Purpose

This is the structure of the client capability list header.

Procedure Declaration

```
struct appcrtrcap_hdr
     (
     unsigned char rc;
     unsigned char opcode;
     unsigned int length;
     );
```

Parameters

rc is used to store the overall return code of the capabilities request.

opcode signals the get capabilities request. Its value must be EHNAPPC_OC_CAPABILITIES (0x17).

length identifies the length of the entire capabilities list. The length includes the size of the header plus the size of each capability structure.

appcrtrcap_mult

Purpose

This is the capability structure used to determine the optimal communications buffer multiplier.

Procedure Declaration

```
struct appcrtrcap_mult
     (
     unsigned int length;
     unsigned char identifier;
     unsigned char rc;
     unsigned int data;
     );
```

Parameters

length identifies the length of this capability structure.

identifier signals the optimal communications buffer multiplier. Its value must be EHNAPPC_CAP_OPTIMAL_COM_SIZE (X'02').

rc is used to store the return code of this capability request.

data is used to return the optimal communications buffer multiplier.

appcrtrcap_query

Purpose

This is the capability structure used to query if the client supports the specified capability.

Procedure Declaration

```
struct appcrtrcap_query
     (
     unsigned int length;
     unsigned char identifier;
     unsigned char rc;
     unsigned char data;
      );
```

Parameters

length identifies the length of this capabilities structure.

identifier identifies the function to be queried. Possible values are:
EHNAPPC_CAP_QUERY_CONV_STATE (3)
EHNAPPC_CAP_EXT_ALLOCATE (4)

rc is used to store the return code of this capability request.

data is used to return whether or not the specified function is supported.

Return Codes for the EHNAPPC API

Functions in the client Windows API use the following return code constants defined in E32APPC.H.

Table 22. Return Codes

Return Code	Hex Value	Description
EHNAPPC_OK	0	Command completed successfully
ENHAPPC_DEALLOCNORMAL	1	Deallocation normal.
ENHAPPC_PROGRAMMERNOTRUNCATION	2	Program error; no truncation.
ENHAPPC_PROGRAMMERTRUNCATION	3	Program error; truncation.
ENHAPPC_PROGRAMMERPURGING	4	Program error; purging.
ENHAPPC_RESOURCEFAILURETRY	5	Resource failure retry.
ENHAPPC_RESOURCEFAILURENORETRY	6	Resource failure no retry.
ENHAPPC_UNSUCCESSFUL	7	Unsuccessful.
ENHAPPC_APPCBUSY	8	APPC busy.
ENHAPPC_PARMCHKINVALIDVERB	14	Parameter check; incorrect verb.
ENHAPPC_PARMCHKINVALIDCONVERID	15	Parameter check; incorrect conversation ID.
ENHAPPC_PARMCHKBUFFERCROSSEG	16	Parameter check; buffer crossed segment.
ENHAPPC_PARMCHKTPNAMELENGTH	17	Parameter check; transaction program name length.
ENHAPPC_PARMCHKINVCONVERTYPE	18	Parameter check; incorrect conversation type.
ENHAPPC_PARMCHKBADSYNCLVLALLOC	19	Parameter check; bad synchronization level allocate.
ENHAPPC_PARMCHKBADRETURNCNTRL	1A	Parameter check; bad return control.
ENHAPPC_ ENHAPPC_PARMCHKPIPTOOLONG	1B	Parameter check: PIP data too long.
ENHAPPC_PARMCHKBADPARTNERNAME	1C	Parameter check; bad partner name.
ENHAPPC_PARMCHKCONFNOTALLOWED	1D	Parameter check; confirm not allowed.

Table 22. Return Codes (continued)

Return Code	Hex Value	Description
ENHAPPC_PARMCHKBADDEALLOCTYPE	1E	Parameter check; bad deallocation type.
ENHAPPC_PARMCHKPREPTORCVTYPE	1F	Parameter check; prepare to receive type.
ENHAPPC_PARMCHKBADFILLTYPE	20	Parameter check; bad fill type.
ENHAPPC_PARMCHKRECMAXLEN	21	Parameter check; receive maximum length.
ENHAPPC_PARMCHKUNKNOWNSECTYPE	22	Parameter check; reserved field not zero.
ENHAPPC_PARMCHKRESFLDNOTZERO	23	Parameter check; reserved field not zero.
ENHAPPC_STATECHKNOTINCONFSTAT	28	State check; not in confirmed state.
ENHAPPC_STATECHKNOTINRECEIVE	29	State check; not in receive.
ENAHAPPC_STATECHKREQSNDBADSTATN	2A	State check; request to send bad state.
ENHAPPC_STATECHKSNDINBADSTATE	2B	State check; send in bad state.
ENHAPPC_STATECHKSNDERRBADSTAT	2C	State check; send error bad state.
ENHAPPC_ALLOCERRNORETRY	32	Allocation error; no retry.
ENHAPPC_ALLOCERRRETRY	33	Allocation error; retry.
ENHAPPC_ALLOCERROGMNOTAVAILNR	34	Allocation error; program not available no retry.
ENHAPPC_ALLOCERRTPNNOTRECOG	35	Allocation error; transaction program name not recognized.
ENHAPPC_ALLOCERRPGMNOTAVAILR	36	Allocation error; program no available retry.
ENHAPPC_ALLOCERRSECNOTVALID	37	Allocation error; security not valid.
ENHAPPC_ALLOCERRCONVTYP	38	Allocation error; conversation type mismatch.
ENHAPPC_ALLOCERRPIPNOTALLOWED	39	Allocation error; PIP data not allowed.
ENHAPPC_ALLOCERRPIPNOTCORRECT	3A	Allocation error; PIP data not correct.
ENHAPPC_ALLOCERRSYNCHLEVEL	3B	Allocation error; synchronization level not supported.
ENHAPPC_DEALLOCABENDPROGRAM	46	Deallocation abend program.
ENHAPPC_INSUFFICIENTMEMORY	47	Insufficient memory.
ENHAPPC_MEMORYALLOCERROR	47	Memory allocation error.
ENHAPPC_MEMORYALLCERROR	48	Memory allocation error.
ENHAPPC_TOOMANYCONVERSATIONS	4A	Too many conversations.
ENHAPPC_CONVTABLEFULL	4B	Conversion table full.
ENHAPPC_CLIENTNOTINSTALLED	4C	Client not installed
ENHAPPC_CLIENTWRONGLEVEL	4C	Client at wrong level.
ENHAPPC_PCSWINNOTLOADED	4D	PSWIN not loaded.
ENHAPPC_PCSWINOUTOFMEMORY	4E	PCSWIN out of memory.
ENHAPPC_INVALIDUSERIDLEN	4F	Incorrect user ID length.
ENHAPPC_INVALIDPASSWORDLEN	50	Incorrect password length.
ENHAPPC_INVALIDUNAME	51	Incorrect LU length.
ENHAPPC_UNDEFINED	63	Undefined.

Running 16-Bit EHNAPPC Programs on Windows 95 and Windows NT

Communications Server Windows 95 and Windows NT clients provide the capability of running your existing 16-bit EHNAPPC programs on Windows 95 and Windows NT. To do so, start the program EHNAPPCD from your Communications Server client subdirectory before you start any of your 16-bit EHNAPPC applications. This program provides the necessary thunking to the 32-bit E32APPC.DLL.

Chapter 19. Data Transform Windows Application Program Interface

The data transform API provides the capability to convert data between AS/400 and PC formats. Translation may be needed when sending and receiving data to and from the AS/400. The data transform API supports conversion of text and numerous numeric formats.

This chapter describes the individual routines and return codes that make up the data transform API.

Data Transform Windows API Routines

The following discussions of each data transform API routine describe in detail:

- Purpose
- Procedure declaration
- Parameters
- Return codes

EHNDT_ANSIToEBCDIC

Purpose

This function translates a string from the Windows ANSI code page to EBCDIC. The router must be loaded so that this routine can access the ASCII-to-EBCDIC translation table.

If the target string is not large enough to contain the translated string, the translation stops at the end of the target string. If the target string is larger than required, it is filled with blanks to the end of the string.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern unsigned int EHNDT_ANSIToEBCDIC(
     HWND           hWnd,
     LPSTR          lpsSource,
     LPSTR          lpsTarget,
     unsigned in    wSource,
     LPWORD         lpwTarget );
```

Parameters

hWnd identifies the current window of the application.

lpsSource points to the source (ANSI) string to convert.

lpsTarget points to the target (translated) string. **wSource** identifies the length of the source string in bytes.

lpwTarget points to a word variable containing the size of the target buffer. This variable will be updated with the total number of translated characters in the target buffer.

© Copyright IBM Corp. 1994, 1997

Return Codes

If the function is successful, EHNDT_SUCCESS (X'0000') is returned. If the router is not loaded, EHNDT_A2E_TABLE_NOT_FOUND (X'FFFC') is returned. If an error occurs while attempting to allocate a temporary buffer, EHNDT_MEMALLOC(X'FFFF') is returned. If incorrect data is found during translation, the return code is the location of the first untranslated character plus one.

EHNDT_ASCIIToEBCDIC

Purpose

This function translates a string from ASCII to EBCDIC. The router must be loaded so that this routine can access the ASCII-to-EBCDIC translation table. If the target string is not large enough to contain the translated string, the translation stops at the end of the target string. If the target string is larger than required, it is blank filled to the end of the string.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern unsigned int EHNDT_ASCIIToEBCDIC(
     HWND            hWnd,
     LPSTR           lpsTarget,
     LPSTR           lpsSource,
     unsigned in     wSource,
     LPWORD          lpwTarget );
```

Parameters

hWnd identifies the current window of the application.

lpsTarget points to the target (translated) string.

lpsSource points to the source (ASCII) string to convert.

wSource identifies the length of the source string in bytes.

lpwTarget points to a word variable containing the size of the target buffer. This variable will be updated with the total number of translated characters in the target buffer.

Return Codes

If the function is successful, EHNDT_SUCCESS (X'0000') is returned. If the router is not loaded, EHNDT_A2E_TABLE_NOT_FOUND (X'FFFC') is returned.

If incorrect data is found during translation, the return code is the location of the first untranslated character plus one.

EHNDT_EBCDICToANSI

Purpose

This function converts a string from EBCDIC to the Windows ANSI code page. The router must be loaded so that this routine can access the ASCII-to-EBCDIC translation table.

If the target string is not large enough to contain the translated string, the translation stops at the end of the target string. If the target string is larger than required, it is blank filled to the end of the string.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern unsigned int EHNDT_EBCDICToANSI(
HWND            hWnd,
LPSTR           lpsTarget,
LPSTR           lpsSource,
unsigned int    wSource,
LPWORD          lpwTarget ); :
```

Parameters

hWnd identifies the current window of the application.

lpsTarget points to the target (translated) string

lpsSource points to the source (EBCDIC) string to convert.

wSource identifies the length of the source string in bytes

lpwTarget points to a word variable containing the size of the target buffer. This variable will be updated with the total number of translated characters in the target buffer.

Return Codes

If the function is successful, EHNDT_SUCCESS ('0000') is returned. If the router is not loaded, EHNDT_E2A_TABLE_.NOT_FOUND ('FFFC') is returned. If incorrect data is found during translation, the return code is the location of the first untranslated character plus one.

EHNDT_EBCDICToASCII

Purpose

This function converts a string from EBCDIC to ASCII. The router must be loaded so that this routine can access the ASCII-to-EBCDIC translation table.

If the target string is not large enough to contain the translated string, the translation stops at the end of the target string. If the target string is larger than required, it is blank filled to the end of the string.

Procedure Declaration

```
#include <WINDOWS.H>
#include "E32APPC.H"
extern unsigned int EHNDT_EBCDICToASCII(
     HWND          hWnd,
     LPSTR         lpsTarget,
     LPSTR         lpsSource,
     unsigned int  wSource,
     LPWORD        lpwTarget );
```

Parameters

hWnd identifies the current window of the application.

lpsTarget points to the target (translated) string.

lpsSource points to the source (EBCDIC) string to convert.

wSource identifies the length of the source string in bytes.

lpwTarget points to a word variable containing the size of the target buffer. This variable will be updated with the total number of translated characters in the target buffer.

Return Codes

If the function is successful, EHNDT_SUCCESS ('0000') is returned. If the router is not loaded, EHNDT_.E2A_TABLE_NOT_FOUND ('FFFC') is returned. If incorrect data is found during translation, the return code is the location of the first untranslated character plus one.

Appendix A. APPC Common Return Codes

This appendix describes the primary (and, if applicable, secondary) return codes that are common to several APPC verbs.

Verb-specific return codes are described in the documentation for the individual verbs.

AP_ALLOCATION_ERROR
Communications Server has failed to allocate a conversation. The conversation state is set to RESET. This code can be returned through a verb issued after **ALLOCATE** or **MC_ALLOCATE**. The associated secondary return codes are as follows:

AP_ALLOCATION_FAILURE_NO_RETRY
The conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not attempt to retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY
The conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation, preferably after a timeout to permit the condition to clear.

AP_SECURITY_NOT_VALID
The user ID or password specified in the allocation request was not accepted by the partner LU.

AP_TRANS_PGM_NOT_AVAIL_RETRY
The remote LU rejected the allocation request because it was unable to start the requested partner transaction program. The requested transaction program (TP) is not available because of a transient condition, such as a timeout. The reason for the error might be logged on the remote node. The condition might clear itself without operator intervention. The transaction program should retry the conversation, preferably after a timeout to permit the condition to clear.

AP_TRANS_PGM_NOT_AVAIL_NO_RTRY
The remote LU rejected the allocation request because it was unable to start the requested partner transaction program. The requested transaction program (TP) is not available because of a permanent or semi-permanent condition. The reason for the error might be logged on the remote node. The condition will not clear itself without operator intervention. The transaction program should not retry the conversation until the error condition has been cleared.

AP_TP_NAME_NOT_RECOGNIZED
The transaction program name specified in the allocation request is not recognized by the partner LU.

AP_PIP_NOT_ALLOWED
The requested transaction program cannot receive program initialization parameters (PIP). This indicates a mismatch between the local and partner transaction programs.

AP_PIP_NOT_SPECIFIED_CORRECTLY
The requested transaction program can receive program initialization

© Copyright IBM Corp. 1994, 1997

parameters (PIP), but detected an error in the supplied PIP. This indicates a mismatch between the local and partner transaction programs.

AP_CONVERSATION_TYPE_MISMATCH
The requested transaction program cannot support conversations of the type (basic or mapped) specified in the allocation request. This indicates a mismatch between the local and partner transaction programs.

AP_SYNC_LEVEL_NOT_SUPPORTED
The requested transaction program cannot support conversations with the **sync_level** (AP_NONE, AP_CONFIRM_SYNC_LEVEL or AP_SYNCPT) specified in the allocation request. This indicates a mismatch between the local and partner transaction programs.

AP_CANCELLED
The verb returned because the conversation was canceled (the transaction program issued a **CANCEL_CONVERSATION** verb).

AP_CONV_FAILURE_NO_RETRY
The conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

AP_CONV_FAILURE_RETRY
The conversation was terminated because of a temporary error. Restart the transaction program to see if the problem occurs again. If it does, the system administrator should examine the error log to determine the cause of the error.

AP_CONVERSATION_TYPE_MIXED
The transaction program has attempted to mix conversation verbs for different conversation types on the same conversation. For example, the transaction program issued an **MC_ALLOCATE** verb followed by a **CONFIRM** verb.

AP_DEALLOC_ABEND
The conversation has been deallocated for one of the following reasons.

- The partner transaction program has issued the **MC_DEALLOCATE** verb with **dealloc_type** set to AP_ABEND.
- The partner transaction program has ended abnormally, causing the partner LU to send an **MC_DEALLOCATE** request.

AP_DEALLOC_ABEND_PROG
The conversation has been deallocated for one of the following reasons.

- The partner transaction program has issued the **DEALLOCATE** verb with **dealloc_type** set to AP_ABEND_PROG.
- The partner transaction program has ended abnormally, causing the partner LU to send a **DEALLOCATE** request.

AP_DEALLOC_ABEND_SVC
The conversation has been deallocated because the partner transaction program issued the **DEALLOCATE** verb with **dealloc_type** set to AP_ABEND_SVC.

AP_DEALLOC_ABEND_TIMER
The conversation has been deallocated because the partner transaction program has issued the **DEALLOCATE** verb with **dealloc_type** set to AP_ABEND_TIMER.

AP_DEALLOC_NORMAL

This return code does not indicate an error. The partner transaction program issued the **DEALLOCATE** or **MC_DEALLOCATE** verb with **dealloc_type** set to one of the following values.

- AP_FLUSH
- AP_SYNC_LEVEL with the synchronization level of the conversation specified as AP_NONE

AP_DUPLEX_TYPE_MIXED

The transaction program has attempted to issue a conversation verb with a different conversation **duplex_type**. For example, the transaction program issued a half-duplex **MC_FLUSH** verb (without **AP_FULL_DUPLEX_CONVERSATION** set in **opext**) on a full-duplex conversation.

AP_ERROR_INDICATION

This return code is used on full-duplex conversations only. A send queue operation has failed because the partner transaction program has terminated the conversation. If the conversation state is send-only, the conversation has now ended. If the conversation state is send-receive or receive-only, the conversation will end when the appropriate return code is returned to a receive queue verb. The associated secondary return codes are:

AP_ALLOCATION_ERROR_PENDING

The remote LU rejected the allocation request.

AP_DEALLOC_ABEND_PROG_PENDING

The conversation has been deallocated for one of the following reasons:

- The partner transaction program has issued the **DEALLOCATE** verb with **dealloc_type** set to AP_ABEND_PROG.
- The partner transaction program has ended abnormally causing the partner LU to send a **DEALLOCATE** request.

AP_DEALLOC_ABEND_SVC_PENDING

The conversation has been deallocated because the partner transaction program issued the **DEALLOCATE** verb with **dealloc_type** set to AP_ABEND_SVC.

AP_DEALLOC_ABEND_TIMER_PENDING

The conversation has been deallocated because the partner transaction program issued the **DEALLOCATE** verb with **dealloc_type** set to AP_ABEND_TIMER.

AP_UNKNOWN_ERROR_TYPE_PENDING

The conversation has been deallocated by the partner transaction program, but the local LU does not recognize the reason.

AP_OPERATION_INCOMPLETE

The transaction program issued a nonblocking verb that started processing, but did not complete. When verb processing completes, the final return code will be set and the stub will notify the transaction program.

AP_PROG_ERROR_NO_TRUNC

The partner transaction program has issued one of the following verbs while the conversation was in SEND state.

- **SEND_ERROR** with **err_type** set to AP_PROG
- **MC_SEND_ERROR**

Data was not truncated.

AP_PROG_ERROR_PURGING
The partner transaction program issued one of the following verbs while in RECEIVE, PENDING_POST, CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state.

- **SEND_ERROR** with **err_type** set to AP_PROG.
- **MC_SEND_ERROR**

Data sent, but not yet received, is purged.

AP_PROG_ERROR_TRUNC
In SEND state, after sending an incomplete logical record, the partner transaction program issued a **SEND_ERROR** verb with **err_type** set to AP_PROG. The local transaction program might have received the first part of the logical record through a **RECEIVE** verb.

AP_SVC_ERROR_NO_TRUNC
While in SEND state, the partner transaction program (or partner LU) issued a **SEND_ERROR** verb with **err_type** set to AP_SVC. Data was not truncated.

AP_SVC_ERROR_PURGING
The partner transaction program (or partner LU) issued a **SEND_ERROR** verb with **err_type** set to AP_SVC while in RECEIVE, PENDING_POST, CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state. Data sent to the partner transaction program might have been purged.

AP_SVC_ERROR_TRUNC
In SEND state, after sending an incomplete logical record, the partner transaction program (or partner LU) issued a **SEND_ERROR** verb. The local transaction program might have received the first part of the logical record.

AP_TP_BUSY
The local transaction program has issued a blocking verb to Communications Server while Communications Server was processing another verb for the same conversation.

AP_UNEXPECTED_SYSTEM_ERROR
Communications Server has encountered an unexpected system error, and cannot complete the verb. Usually these errors arise from a shortage of system resources (for example, memory), and are usually transient. Check the system log for more details.

AP_SEC_REQUESTED_NOT_SUPPORTED
The local LU is unable to allocate a conversation because the session with the partner LU does not support Password Substitution. The security type requested on the ALLOCATE or SEND_CONVERSATION is AP_PGM_STRONG, that requires Password Substitution support.

Appendix B. LUA Verb Return Codes

This appendix describes the primary (and, if applicable, secondary) return codes that are common to several SLI verbs.

Verb-specific return codes are described in the documentation for the individual verbs.

Primary Return Codes

The following section contains the LUA primary return codes: the

LUA_OK
The LUA verb completed successfully.

LUA_PARAMETER_CHECK
The LUA feature detected an incorrect parameter.

LUA_STATE_CHECK
The session was in an incorrect state for the verb that was issued.

LUA_SESSION_FAILURE

The session has been brought down. The specific reason is identified in the secondary return code.

LUA_UNSUCCESSFUL
This verb did not successfully complete.

LUA_NEGATIVE_RESPONSE
One of the following conditions occurred:

- The end-of-chain arrived for a chain that was negatively responded to by the LUA application program. The secondary return code is not set.
- LUA detected an error in a message from the primary LU and sent a negative response. This error is returned when the end-of-chain is received from the primary LU. The secondary return code contains the sense data that was sent with the negative response.

LUA_CANCELED
The verb was canceled because of reasons specified in the secondary return code.

LUA_IN_PROGRESS

This synchronous code is returned when an asynchronous command is received and has not completed.

LUA_STATUS
The SLI has status information for the application in the secondary return code.

LUA_COMM_SUBSYSTEM_ABENDED

Communications Server abnormally ended.

LUA_COMM_SUBSYSTEM_NOT_LOADED

Communications Server was not loaded.

LUA_INVALID_VERB_SEGMENT

LUA could not process the verb because the entire verb control block is not contained in the data segment. The address of the end of the verb control block is beyond the end of the segment.

LUA_UNEXPECTED_DOS_ERROR

An unexpected system error occurs after Communications Server issues a system call, the verb is posted with the primary return code UNEXPECTED_DOS_ERROR. The secondary return code contains the unexpected system error.

LUA_STACK_TOO_SMALL

The LUA application stack is too small for LUA to process the request.

LUA_INVALID_VERB

LUA does not recognize the verb code or the verb operation code (or both) in the verb control block it received.

Secondary Return Codes

The following section contains the LUA seondary return codes: the

LUA_SEC_OK
Additional information is available for the primary return code associated with this secondary return code.

LUA_INVALID_LUNAME
The verb specified an invalid **lua_name**.

LUA_BAD_SESSION_ID
The verb control block specified an incorrect value for the **lua_sid** parameter.

LUA_DATA_TRUNCATED
The buffer length (as specified in **lua_max_length**) was not long enough for the data received, so the data was truncated.

LUA_BAD_DATA_PTR
The command requires data to be supplied or returned, but the **lua_data_ptr** parameter either either contains an invalid pointer or does not point to a read/write segment.

LUA_DATA_SEG_LENGTH_ERROR
One of the following conditions occurred:

- The data segment supplied on an **RUI_READ** or **SLI_RECEIVE** verb is shorter than the length given in the **lua_max_length** parameter.
- The data segment was supplied on an **RUI_WRITE** or **SLI_SEND** verb is shorter than the length given in the **lua_data_length** parameter.
- The data segment supplied on an **RUI_READ**, **RUI_WRITE**, **SLI_RECEIVE**, or **SLI_SEND** verb is not a read/write data segment.

LUA_RESERVED_FIELD_NOT_ZERO
The command that was just issued has a reserved parameter that is not zero.

LUA_INVALID_POST_HANDLE
A valid semaphore was not specified in the LUA verb control block. When an LUA verb does not complete synchronously, a semaphore is needed to signal the completion of the verb.

LUA_PURGED
: An **RUI_READ** or an **SLI_RECEIVE** verb was canceled because an **RUI_PURGE** or an **SLI_PURGE** was issued.

LUA_BID_VERB_SEG_ERROR
: The buffer with the **SLI_BID** verb control block was released before the **SLI_RECEIVE** with **lua_flag1.bid_enable** set to 1 was issued.

LUA_NO_PREVIOUS_BID_ENABLED
: An **RUI_BID** or **SLI_BID** verb was not issued before an **RUI_READ** or **SLI_RECEIVE** verb with **lua_flag1.bid_enable** was issued.

LUA_NO_DATA
: An **RUI_READ** or **SLI_RECEIVE** verb was issued with the NO_WAIT parameter and there was no data available to read.

LUA_BID_ALREADY_ENABLED
: An **RUI_BID** or **SLI_BID** verb was active when an **RUI_READ** or **SLI_RECEIVE** verb with **lua_flag1.bid_enable** was issued.

LUA_VERB_RECORD_SPANS_SEGMENTS
: The LUA verb control block contains a length parameter that, when added to the offset of the segment, goes past the end of the segment.

LUA_INVALID_FLOW
: An LUA verb was issued with the **lua_flag1** flow flags set in error. Check that the correct number of **lua_flag1** flow flags were set as follows:

- For **RUI_READ** or **SLI_RECEIVE**, at least one
- For **RUI_WRITE**, only one
- For **SLI_SEND**, only one **lua_flag1** flow flag must be set when sending an SNA response.

LUA_NOT_ACTIVE
: An application program issued an LUA verb at a time that LUA was not active within Communications Server.

LUA_VERB_LENGTH_INVALID
: A verb was issued with an incorrect **lua_verb_length** parameter. The length specified is not equal to the length that LUA expected.

LUA_REQUIRED_FIELD_MISSING
: The issued **RUI_WRITE** verb either did not include a data pointer (if the data count was not zero) or it did not include an **lua_flag1**flow flag.

LUA_READY
: The SLI session is now ready to process additional commands. This status is issued after a prior NOT_READY status was received, or after a **SLI_CLOSE** verb completed with the primary return code CANCELED and secondary return code RECEIVE_UNBIND_HOLD or RECEIVED_UNBIND_NORMAL.

LUA_NOT_READY
: The SLI session is temporarily syspended for either of the following reasons:

- A CLEAR command was received. The SLI session resumes when an SDT command is received.
- An UNBIND command was received. The session is suspended until BIND, optional STSN and SDT commands are received. Any user extension routines that were supplied by the original **SLI_OPEN** verb are

called again; therefore, these routines must be reentrant. After the SLI prcesses the SDT command, the SLI session resumes. Two types of UNBIND commands are:

– UNBIND type X'02', which means that a new BIND is coming
– UNBIND type X'01', which means that the application specified an **lua_session_type** of LUA_SESSION_TYPE_DEDICATED in the **SLI_OPEN** verb that started this session.

LUA_INIT_COMPLETE
When the LUA interface initializes the session while **SLI_OPEN** is processing, this status is returned on **SLI_RECEIVE** or **SLI_BID** verbs for LUA applications that issue **SLI_OPEN** with the **LUA_INIT_TYPE_PRIM_SSCP** parameter.

LUA_SESSION_END_REQUESTED
SLI received a SHUTD command from the host, indicating the host is ready to shut down the session.

LUA_NO_SLI_SESSION
A command was issued when a session was not open, or a session is being taken down because of an **SLI_CLOSE** verb or session failure. An **SLI_RECEIVE** or **SLI_SEND** verb issued during the processing of an **SLI_OPEN** verb returns this code when:

- The **SLI_OPEN lua_init_type** parameter is not set to LUA_INIT_TYPE_PRIM_SSCP. An SLI_BID verb also returns this code under these circumstances.
- The **SLI_RECEIVE** or **SLI_SEND lua_flag1** parameter does not specify **lua_flag1.sscp_norm**.

The SLI component is in **SLI_OPEN** processing after an UNBIND type X'02' command or UNBIND type X'01' (LUA_SESSION_TYPE_DEDICATED) is received and until the SDT command is processed. UNBIND type X'02' indicates that a new BIND is coming.

LUA_SESSION_ALREADY_OPEN
An **SLI_OPEN** verb was issued for an LU name that already has a session open.

LUA_INVALID_OPEN_INIT_TYPE
An **SLI_OPEN** verb contained an incorrect value in the lua_init_type parameter.

LUA_INVALID_OPEN_DATA
An **SLI_OPEN** verb was issued with the lua_init_type parameter set for secondary initialization with INITSELF (LUA_INIT_TYPE_SEC_IS), and the data buffer does not contain a valid INITSELF command.

LUA_UNEXPECTED_SNA_SEQUENCE
During **SLI_OPEN** processing, an unexpected command or data was received from the host.

LUA_NEG_RSP_FROM_BIND_ROUTINE
The user-provided **SLI_BIND** routine generated a negative response to the BIND. The **SLI_OPEN** verb ends unsucessfully.

LUA_NEG_RSP_FROM_CRV_ROUTINE
The user-provided **SLI_BIND** routine generated a negative response to the BIND. The **SLI_OPEN** verb ends unsucessfully.

LUA_NEG_RSP_FROM_STSN_ROUTINE
The user-supplied SLI STSN routine responded negatively to the STSN. **SLI_OPEN** ended unsuccessfully.

LUA_CRV_ROUTINE_REQUIRED
The user did not provide an SLI CRV routine, but a CRV was received from the host. The SLI issues a negative response to the CRV, and the **SLI_OPEN** verb ends unsuccessfully at this time.

LUA_NEG_RSP_FROM_SDT_ROUTINE
The user-provided SLI SDT routine generated a negative response to an SDT. This condition causes the **SLI_OPEN** verb to end.

LUA_INVALID_OPEN_ROUTINE_TYPE
In the **SLI_OPEN** extension routine list, the **lua_open_routine_type** parameter is not valid.

LUA_MAX_NUMBER_OF_SENDS
The application program issued more than two **SLI_SEND** verbs before one completed.

LUA_SEND_ON_FLOW_PENDING
The application issued an **SLI_SEND** verb for an SNA flow (SSCP-expedited, SSCP-normal, LU-expedited, LU-normal) that already has an **SLI_SEND** verb outstanding.

LUA_INVALID_MESSAGE_TYPE
The SLI does not recognize the **lua_message_type** parameter.

LUA_RECEIVE_ON_FLOW_PENDING
The SLI application issued an **SLI_RECEIVE** verb for an SNA flow that already has an **SLI_RECEIVE** verb outstanding.

LUA_DATA_LENGTH_ERROR
An **SLI_OPEN** command was issued that requires user data that the application program did not supply. Data is required for a secondary-initiated **SLI_OPEN** verb, and 4 bytes of status is required when the application issues an **SLI_SEND** verb for an LUSTAT command.

LUA_CLOSE_PENDING
One of the following has occurred:

- A CLOSE_NORMAL was issued while a CLOSE_NORMAL or a CLOSE_ABEND was pending.
- A CLOSE_ABEND was issued while another CLOSE_ABEND was pending. The only valid reason to issue another CLOSE_ABEND is when a CLOSE_NORMAL is pending.

LUA_NEGATIVE_RSP_CHASE
During **SLI_CLOSE** processing, the SLI received a negative response to a CHASE command from the host. The session is stopped as requested by the **SLI_CLOSE**.

LUA_NEGATIVE_RSP_SHUTC
During **SLI_CLOSE** processing, the SLI received a negative response to a SHUTC command from the host. The session is stopped as requested by the **SLI_CLOSE**.

LUA_NEGATIVE_RSP_SHUTD
During **SLI_CLOSE** processing, the SLI received a negative response to a SHUTD command from the host. The session is stopped as requested by the **SLI_CLOSE**.

LUA_NO_RECEIVE_TO_PURGE
An **SLI_PURGE** verb was issued when no **SLI_RECEIVE** verb was outstanding. Two possible causes are as follows:

- The address contained in the **lua_data_ptr** parameter did not point to the outstanding **SLI_RECEIVE** verb that was to be purged.
- The **SLI_RECEIVE** verb might have completed while the **SLI_PURGE** verb was being processed. This is not an error condition. Code the application program to handle this situation.

LUA_CANCEL_COMMAND_RECEIVED
While processing an **SLI_RECEIVE** verb, the host sent a CANCEL command to cancel the chain of data being received.

LUA_RUI_WRITE_FAILURE
An **RUI_WRITE** verb posted with an unexpected error to the SLI.

LUA_INVALID_SESSION_TYPE
An **SLI_OPEN** verb contained a value that is not valid in the **lua_session_type**.

LUA_SLI_BID_PENDING
An SLI verb was issued while a previously-issued **SLI_BID** is active. Only one **SLI_BID** can be active at a time.

LUA_PURGE _PENDING
An **SLI_PURGE** verb was issued while a previously-issued **SLI_PURGE** is active. Only one **SLI_PURGE** can be active at a time.

LUA_PROCEDURE_ERROR
An NSPE or NOTIFY message was received, indicating a host procedure error occurred. The **SLI_OPEN** is posted with this return code (unless the **SLI_OPEN** verb retry option is used). With **lua_wait** set to a nonzero value, the INITSELF or LOGON message is retried until the host procedure is available or the application issues an **SLI_CLOSE**.

LUA_INVALID_SLI_ENCR_OPTION
The **lua_encr_decr_option** parameter was set to 128 in the **SLI_OPEN** verb. The SLI does not support 128 for the encryption or decryption processing option.

LUA_RECEIVED_UNBIND
The SLI received an UNBIND command from the primary LU while there was an active SLI session. The SLI session is stopped.

LUA_RECEIVED_UNBIND_HOLD
During primary- or secondary-initiated **SLI_CLOSE** normal processing, SLI received an UNBIND type X'02'. Type X'02' means that a new BIND is forthcoming. The session is suspended until BIND, optional CRV and STSN, and SDT commands are received. Any user extension routines that were supplied by the original **SLI_OPEN** verb are called again; these routines must be reentrant. After the SLI processes the SDT command, the SLI session resumes.

LUA_RECEIVED_UNBIND_NORMAL
During primary- or secondary-initiated **SLI_CLOSE** normal processing for a session started with an **SLI_OPEN** verb that specified an **lua_session_type** of LUA_SESSION_TYPE_DEDICATED, SLI received an UNBIND type X'01'. The session is suspended until BIND, optional STSN and SDT commands are received. Any user extension routines that were

supplied by the original **SLI_OPEN** verb are called again; these routines must be reentrant. After the SLI processes the SDT command, the SLI session resumes.

LUA_SLI_LOGIC_ERROR
The SLI detected an internal logic error.

LUA_TERMINATED

A verb that was pending when an **SLI_CLOSE** or **RUI_TERM** verb was issued has been canceled.

LUA_NO_RUI_SESSION
An RUI verb was issued for a session that has not been initialized (with **RUI_INIT**) or a verb other than **RUI_TERM** was issued while an **RUI_INIT** verb for the session was in progress.

This return code can occur when a session outage occurs while no active RUI verbs are outstanding. The next verb issued gets this return code. The application program handles this return code as it would a SESSION_FAILURE.

LUA_DUPLICATE_RUI_INIT
The application program issued an **RUI_INIT** verb for a session that is already initialized or has an **RUI_INIT** verb in progress.

LUA_INVALID_PROCESS
An RUI verb was issued for a session that is already owned by another process.

LUA_API_MODE_CHANGE
A non-SLI request was issued to the RUI on a session that was established by the SLI.

LUA_COMMAND_COUNT_ERROR
The maximum number of issued **RUI_READ** or **RUI_WRITE** verbs was exceeded, or an **RUI_BID** or **RUI_TERM** verb was issued while a previously issued **RUI_BID** or **RUI_TERM** verb was still in progress.

LUA_NO_READ_TO_PURGE
An **RUI_PURGE** verb was issued when no **RUI_READ** verb was outstanding. Two possible causes follow:

- The address contained in the **lua_data_ptr** parameter does not point to the outstanding **RUI_READ** verb to be purged.
- The **RUI_READ** verb completed while the **RUI_PURGE** verb was being processed. This is not an error condition. Code the application program to handle this situation.

LUA_MULTPLE_WRITE_FLOWS
More than one flow flag was turned on in the FLAG1 issued to an **RUI_WRITE** verb.

LUA_DUPLICATE_READ_FLOW
The application program issued an **RUI_READ** for a flow that already has an **RUI_READ** pending.

LUA_DUPLICATE_WRITE_FLOW
The **RUI_WRITE** verb that was issued contained a FLAG1 flow flag that showed a session flow for a previous **RUI_WRITE** verb that had not completed.

LUA_LINK_NOT_STARTED
LUA could not start the data link during session initialization.

LUA_INVALID_ADAPTER
The DLC adapter configuration is incorrect or the configuration file has been damaged.

LUA_ENCR_DECR_LOAD_ERROR
An unexpected error was received while attempting to load the user-provided encryption or decryption dynamic link library.

LUA_ENCR_DECR_PROC_ERROR
An unexpected error was received while attempting to get the procedure address within the user-provided encryption or decryption dynamic link library.

LUA_LINK_NOT_STARTED_RETRY
An **RUI_INIT** or **SLI_OPEN** verb failed because the link could not be activated. This return code implies that something is wrong at the partner location or with the connection between the two machines.

LUA_NEG_NOTIFY_RSP
An **RUI_INIT** was issued that caused a notify request to be sent to the SSCP to indicate the SLU can now be part of a session. The SSCP responded negatively to this notify request. The intended half-session component understood the supported request, but did not process it.

LUA_RUI_LOGIC_ERROR
An RUI internal logic error occurred.

LUA_LU_INOPERATIVE
A severe error occurred while the SLI was attempting to stop the session. This LU is unavailable for any LUA requests until an ACTLU is received from the host.

LUA_RESOURCE_NOT_AVAILABLE
The LU, PU, link station, or link specified in an RU is not available. The **SLI_OPEN** verb cannot can be posted with this return code unless the **SLI_OPEN** retry option is used. With **lua_wait** set to a nonzero value, the INITSELF or LOGON message is retried until the host procedure is available or the application issues an **SLI_CLOSE** verb.

LUA_SESSION_LIMIT_EXCEEDED
The requested session cannot be activated because one of the network addressable units (NAUs) is at its session limit, such as the LU-LU session limit or the LU mode session limit. This sense code applies to the ACTCDRM, the INIT, the BID, and the CINIT requests.

The **SLI_OPEN** verb can be posted with this return code unless the **SLI_OPEN** verb retry option is used. With **lua_wait** set to a nonzero value, the INITSELF or LOGON message is retried until the host procedure is available or the application issues an **SLI_CLOSE** verb.

LUA_SLU_SESSION_LIMIT_EXCEEDED
If accepted, the request would cause the SLU session limit to be exceeded.

LUA_MODE_INCONSISTENCY
The present status does not permit the function to be performed. The intended half-session component understood the supported request, but did not process it. This code can also appear as a sense code in an EXR.

LUA_INSUFFICIENT_RESOURCES
Due to a temporary lack of resources, the receiver cannot act on the request. The intended half-session component understood the supported request, but did not process it.

LUA_RECEIVER_IN_TRANSMIT_MODE
A race condition exists. A normal-flow request was received while the half-duplex contention state was not-receive, or while resources (such as buffers) necessary for handling normal-flow data were unavailable.

This code can also appear as a sense code in an exception request.

LUA_LU_COMPONENT_DISCONNECTED
An LU component is not available because of power-off or some other disconnecting condition.

LUA_NEGOTIABLE_BIND_ERROR
A negotiable BIND was received. The SLI does not allow a negotiable BIND unless there is a user-supplied **SLI_BIND** routine provided through the **SLI_OPEN** verb.

LUA_BIND_FM_PROFILE_ERROR
An unsupported FM profile was detected on the BIND. The SLI supports FM profiles 3 and 4 only.

LUA_BIND_TS_PROFILE_ERROR
An unsupported TS profile was detected on the BIND. The SLI supports TS profiles 3 and 4 only.

LUA_BIND_LU_TYPE_ERROR
An unsupported LU type was detected. LUA supports LU 0, LU 1, LU 2 and LU 3 only.

LUA_SSCP_LU_SESSION_NOT_ACTIVE
The SSCP-LU session required for processing a request is not active. For example, in processing an INITSELF request, the SSCP did not have an active session with the target LU named in the INITSELF.

Bytes 2 and 3 contain sense—code—specific information. The following settings are allowed:

`0000`	No specific code applies.
`0001`	The SSCP-SLU session is being reactivated.
`0002`	The SSCP-PLU session is inactive. The **SLI_OPEN** verb can be posted with this return code unless the **SLI_OPEN** retry option is used. With **lua_wait** set to a nonzero value, the INITSELF or LOGON message is retried untilt he host procedure is available or the application issues an **SLI_CLOSE** verb.
`0003`	The SSCP-SLU session is inactive.
`0004`	The SSCP-SLU session is being reactivated.

LUA_REC_CORR_TABLE_FULL
The session receive correlation table for the flow requested reached its capacity.

LUA_SEND_CORR_TABLE_FULL
The send correlation table for the flow requested reached its capacity.

LUA_SESSION_SERVICES_PATH_ERROR

A session services request cannot be rerouted along a path of SSCP-SSCP sessions. This capability is required, for example, to set up a cross-network LU-LU session.

Bytes 2 and 3 contain sense—code—specific information. The following settings are allowed:

0000 No specific code applies. The **SLI_OPEN** cannot be posted with this return code unless the **SLI_OPEN** retry option is used. With **lua_wait** set to a nonzero value, the INITSELF or LOGON message is retried until the host procedure is available or the application issues an **SLI_CLOSE**.

0001 An SSCP tried unsuccessfully to reroute a session services request to its destination through one or more adjacent SSCPs. This value is sent by a gateway SSCP when it has exhausted trial-and-error rerouting.

SSCP rerouting failed completely. An SSCP tried unsuccessfully to a particular SSCP. For example, this code is associated with specific SSCPs when information about a rerouting failure is displayed in the node that was trying to reroute.

0002

An SSCP is unable to reroute a session services request because a necessary routing table is not available; that is, no adjacent SSCP table corresponds to the rerouting key in the resource identifier control vector.

0003

This SSCP has no predefinition for an LU, but an adjacent SSCP does not support dynamic definition in partner SSCPs. As a result, this SSCP cannot both dynamically define the LU and reroute to that adjacent SSCP.

0005

Retired

0006

Retired

0008

The adjacent SSCP does not support the requested CDINIT function (for example, notification of resource availability or XRF).

000A

An SSCP is unable to reroute a session services request because the request was routed through the same SSCP twice.

000B

The DLU specified in the CDINIT is unknown to the receiving SSCP, and the receiving SSCP cannot reroute the CDINIT.

LUA_RU_LENGTH_ERROR

The requested RU was too long or too short. The RU was delivered to the intended half-session component, but it could not be interpreted or processed. This condition represents a mismatch of half-session capabilities.

This code can also appear as asense code in an EXR.

LUA_FUNCTION_NOT_SUPPORTED
The function that was requested is not supported by LUA. The function may have been specified by a formatted request code, a parameter in an RU, or a control character.

Bytes 2 and 3 that follow the sense code are not used for user-defined data. These bytes contain sense-code-specific information. The following setting is allowed:

0000 The requested function is not supported by LUA.

The RU was delivered to the intended half-session component, but it could not be interpreted or processed. This condition represents a mismatch of half-session capabilities.

LUA_HDX_BRACKET_STATE_ERROR
A protocol machine determined that the current request could not be sent under the existing state error.

LUA_RESPONSE_ALREADY_SENT
A protocol machine determined that the current request could not be sent because a response for the chain had already been sent.

LUA_EXR_SENSE_INCORRECT
The application issued a negative response for a previously received exception request. The sense code in the response was not acceptable.

If the sense code in the exception request is X’0813000’, the sense code in the negative response can be either X’08130000’ or X’08140000’. In all other cases, the sense code in the negative response must be the same as the sense code in the exception request.

LUA_RESPONSE_OUT_OF_ORDER
A protocol machine determined that the current response was not issued to the oldest request.

LUA_CHASE_RESPONSE_REQUIRED
A protocol machine determined that the current request is being attempted with an older CHASE request outstanding.

LUA_CATEGORY_NOT_SUPPORTED
A DFC, SC, NC, or FMD request was received by a half-session not supporting any requests in that category, a network services (NS) request byte 0 was not set to a defined value, or byte 1 was not set to an NS category by the receiver.

LUA_CHAINING_ERROR
An error occurred in the sequence of the chain indicator settings, such as first, middle, first. A request header or a request unit that is not allowed for the receiver’s current session control or data flow control state was detected. This error prevents delivery of the request to the intended half-session component.

LUA_BRACKET
The sender did not enforce bracket rules for the session. A request header or request unit that is not allowed for the receiver’s current session control

or data flow control state was detected. This error prevents delivery of the request to the intended half-session component.

LUA_DIRECTION
A normal-flow request was received while the half-duplex flip-flop state was NOT_RECEIVE. A request header or request unit that is not allowed for the receiver's current session control or data flow control state was detected. This error prevents delivery of the request to the intended half-session component.

LUA_DATA_TRAFFIC_RESET
An FMD or normal-flow DFC request was received by a half-session whose session activation state was active, but whose data traffic state was not active. A request header or a request unit that is not allowed for the receiver's current session control or data flow control state was detected. This error prevents delivery of the request to the intended half-session component.

LUA_DATA_TRAFFIC_QUIESCED
An FMD or a DFC request, received from a half-session that sent a QC command or a SHUTC command, has not responded to a RELQ command. A response header or request unit that is not allowed for the receiver's current session control or data flow control state was detected. This error prevents delivery of the request to the intended half-session component.

LUA_DATA_TRAFFIC_NOT_RESET
A session control request was received while the data traffic state was not reset. A request header or request unit that is not allowed for the receiver's current session control or data flow control state was detected. This error prevents delivery of the request to the intended half-session component.

LUA_NO_BEGIN_BRACKET
A BID or an FMD request that specified BBI=BB was received after the receiver had previously sent a positive response to a BIS command. A request header or request unit that is not allowed for the receiver's current session control or data flow control state was detected. This error prevents delivery of the request to the intended half-session component.

LUA_SC_PROTOCOL_VIOLATION
An SC protocol was violated. A request allowed only after a successful exchange of an SC request and its associated positive response was received before a successful exchange occurred. Byte 4 of the sense data contains the request code. There is no user data associated with this sense code. A request header or request unit that is not allowed for the receiver's current session control or data flow control state was detected. This error prevents delivery of the request to the intended half-session component.

LUA_IMMEDIATE_REQ_MODE_ERROR
The immediate request mode protocol was violated by the request. An RH or RU that is not allowed for the receiver's current session control or data flow control state was detected. This error prevents delivery of the request to the intended half-session component.

LUA_QUEUED_RESPONSE_ERROR
The Queued Response protocol was violated by a request; for example, QRI=¬ QR when an outstanding request has QRI=QR. An RH or an RU that is not allowed for the receiver's current session control or data flow control state was detected. This error prevents delivery of the request to the intended half-session component.

LUA_ERP_SYNC_EVENT_ERROR
The ERP synchronous event protocol was violated. An RH or an RU that is not allowed for the rceiver's current sesion control or data flow control state was detected. This error prevents delivery of the request to the intended half-session component.

LUA_RSP_BEFORE_SENDING_REQ
An attempt was made in half-duplex (flip-flop or contention) send/receive mode to send a normal-flow request when a response to a previously received request has not yet been sent. An RH or an RU that is not allowed for the receiver's current session control or data flow control state was detected. This error prevents delivery of the request to the intended half-session component.

LUA_RSP_CORRELATION_ERROR
A response was received that cannot be correlated with a previously sent request, or a response was sent that cannot be correlated with a previously received request.

LUA_RSP_PROTOCOL_ERROR
A response was received from the primary half-session that violated the response protocol, such as:

- A positive response (+RSP) was received for an RQE chain.
- Two responses were received for one chain.

LUA_INVALID_SC_OR_NC_RH
The RH of a session control (SC) or network control (NC) request was not valid. For example, an SC RH with the pacing request indicator set to 1 is not valid. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the sender's failure to enforce session ru

LUA_BB_NOT_ALLOWED
The begin bracket indicator (BB) was specified incorrectly; for example, BBI=BB with BCI=¬BC. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_EB_NOT_ALLOWED
The end bracket indicator (EB) was specified incorrectly; for example, by EBI=EB with BCI=¬BC, or by the primary half-session when only the secondary can send an EB, or by the secondary half-session when only the primary can send an EB. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_EXCEPTION_RSP_NOT_ALLOWED
An exception response was requested when it was not permitted. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and

are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_DEFINITE_RSP_NOT_ALLOWED
A definite response was requested when it was not permitted. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_PACING_NOT_SUPPORTED
The pacing indicator was set on a request, but the receiving half-session or the boundary function half-session does not support pacing for this session. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_CD_NOT_ALLOWED
The change-direction indicator (CD) was specified incorrectly; for example, CDI=CD with ECI=¬EC or CDI=CD with EBI=EB. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_NO_RESPONSE_NOT_ALLOWED
No-response was specified on a request when it was not permitted. No-response is used only on EXR. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_CHAINING_NOT_SUPPORTED
The chaining indicators (BCI and ECI) were specified incorrectly; for example, chaining bits other than BCI=BC and ECI=EC were indicated, but multiple-request chains are not supported for the session or for the category specified in the request header. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent the delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_BRACKETS_NOT_SUPPORTED
The bracket indicators (BBI and EBI) were specified incorrectly; for example, a bracket indicator was set (BBI=BB or EBI=EB), but brackets are not used for the session. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_CD_NOT_SUPPORTED

The change-direction indicator was set, but is not supported. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_INCORRECT_USE_OF_FI

The format indicator (FI) was specified incorrectly; for example, the FI was set with BCI=¬BC or the FI was not set on a DFC request. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_ALTERNATE_CODE_NOT_SUPPORTED

The code selection indicator (CSI) was set when it was not supported for the session. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_INCORRECT_RU_CATEGORY

The RU category indicator was specified incorrectly; for example, an expedited-flow request or a response was specified with the RU category indicator = FMD. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_INCORRECT_REQUEST_CODE

The request code on a response does not match the request code on its corresponding request. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_INCORRECT_SPEC_OF_SDI_RTI

The sense-data-included indicator (SDI) and the response-type indicator (RTI) were not specified correctly on a response. The proper value pairs are (SDI=SD, RTI=negative) and (SDI=¬SD, RTI=positive). The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_INCORRECT_DR1I_DR2I_ERI

The definite response 1 indicator (DR1I), the definite response 2 indicator (DR2I), and the exception response indicator (ERI) were specified incorrectly. For example, a CANCEL request was not specified with DR1I=DR1, DR2I=¬DR2, and ERI=¬ER. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_INCORRECT_USE_OF_QRI
The queued response indicator (QRI) was specified incorrectly; for example, QRI=QR on an expedited-flow request. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_INCORRECT_USE_OF_EDI
The enciphered data indicator (EDI) was specified incorrectly; for example EDI=ED on a DFC request. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_INCORRECT_USE_OF_PDI
The padded data indicator (PDI) was specified incorrectly, such as PDI=PD on a DFC request. The value of a parameter or combination of parameters in the RH violates the architectural rules or previously selected LOGON options. These errors prevent delivery of the request to the intended half-session component and are independent of the current states of the session. These errors might result from the failure of the sender to enforce session rules.

LUA_NAU_INOPERATIVE
The NAU is unable to process requests or responses. For example, the NAU was disrupted by an abnormal end. The request could not be delivered to the intended receiver, because of a path outage, an incorrect sequence of activation requests, or one of the listed path information unit (PIU) errors. A path error that is received while the session is active generally indicates that the path to the session partner is lost.

LUA_NO_SESSION
No half-session is active in the receiving end node for the indicated origin-destination pair or no boundary function half-session component is active for the origin-destination pair in a node that provides the boundary function. A session activation request is needed. The request could not be delivered to the intended receiver because of a path outage or an incorrect sequence of activation requests. A path error that is received while the session is active generally indicates that the path to the session partner is lost.

LUA_BRACKET_RACE_ERROR
A loss of contention within the bracket protocol occurred. When bracket initiation or bracket termination by both NAUs occurs, contention is lost. The intended half-session component understood the supported request, but did not process it.

LUA_BB_REJECT_NO_RTR
A BID or a begin-bracket indicator was received while the first speaker was

in the in-bracket state or while the first speaker was in the between-brackets state. The first speaker denied permission. No RTR command will be sent. The intended half-session component understood the supported request, but did not process it.

LUA_CRYPTOGRAPHY_INOPERATIVE
The receiver of a request was not able to decipher the request because of a malfunction in its cryptography facility. The intended half-session component understood the supported request, but did not process it.

LUA_SYNC_EVENT_RESPONSE
A negative response to a synchronizing request was received. The intended half-session component understood the supported request, but did not process it.

LUA_RU_DATA_ERROR
Data in the request RU is not acceptable to the receiving FMDS component. For example, a character code is not in the set that is supported, a formatted data parameter is not acceptable to presentation services, or a required name in the request has been omitted. The RU was delivered to the intended half-session component, but it could not be interpreted or processed. This condition represents a mismatch of half-session capabilities.

LUA_INCORRECT_SEQUENCE_NUMBER
The sequence number that was received on a normal-flow request was not greater than the last sequence number. A sequence number error or an RH or RU that is not allowed for this receiver’s current session control or data flow control state was detected. This error prevents the delivery of the request to the intended half-session component.

Appendix C. APPC Conversation State Transitions

The following tables show the conversation states in which each APPC verb can be issued, and the state change that occurs on completion of the verb. In some cases, the state change depends on the **primary_rc** parameter returned to the verb; where this applies, the applicable **primary_rc** values are listed in the Return codes column.

Where no return codes are shown, the state changes are the same for all return codes (except as described in Notes 2 and 3 following the table).

The possible conversation states are shown as column headings. Against each verb, the following information is given under each heading to indicate the results of issuing the verb in this state:

- **X** if the verb cannot be issued in this state.
- **S**, **SP**, **R**, **C**, **CS**, **CD**, or **P** to indicate the state of the conversation after the verb has completed: **R**eset, **S**end, **S**end **P**ending, **R**eceive, **C**onfirm, **C**onfirm **S**end, **C**onfirm **D**eallocate, or **P**ending Post.
- **/** if it is not applicable to consider the previous state. This applies to the **[MC_]ALLOCATE** and **RECEIVE_ALLOCATE** verbs; these verbs always start a new conversation as though they were in Reset state, with no effect on the conversation (if any) in which they were issued.
- Blank if the return code shown cannot occur in this state.

For information on full-duplex conversation state transitions, see Table 24 on page 340 .

Table 23. APPC Half-Duplex Conversation State Transitions

Verb Return Codes	Reset (T)	Send (S)	Send Pending (SP)	Receive (R)	Confirm (C)	Confirm Send (CS)	Confirm Deall (CD)	Pend-Post (PS)
[MC_]ALLOCATE								
AP_OK	S	/	/	/	/	/	/	/
(other)	T							
CANCEL_CONVERSATION	X	T	T	T	T	T	T	T
[MC_]CONFIRM								
AP_OK	X	S	S	X	X	X	X	X
AP_ERROR		R	R					
[MC_]CONFIRMED	X	X	X	X	R	S	T	X
[MC_]DEALLOCATE (Abend)	X	T	T	T	T	T	T	T
[MC_]DEALLOCATE (Other)	X			X	X	X	X	X
AP_ERROR		R	R					
(other)		T	T					
[MC_]FLUSH	X	S	S	X	X	X	X	X
[MC_]GET_ATTRIBUTES	X	S	SP	R	C	CS	CD	P

Table 23. APPC Half-Duplex Conversation State Transitions (continued)

Verb Return Codes	**Reset (T)**	**Send (S)**	**Send Pending (SP)**	**Receive (R)**	**Confirm (C)**	**Confirm Send (CS)**	**Confirm Deall (CD)**	**Pend-Post (PS)**
GET_STATE	X	S	SP	R	C	CS	CD	P
GET_TYPE	X	S	SP	R	C	CS	CD	P
[MC_]PREPARE_TO_ RECEIVE	X	R	R	X	X	X	X	X
RECEIVE_ALLOCATE AP_OK (other)	 R T	/	/	/	/	/	/	/
[MC_]RECEIVE_AND_POST (Note 4)	X	P	P	P	X	X	X	X
[MC_]RECEIVE_AND_WAIT	X	Note 5	Note 5	Note 5	X	X	X	X
[MC_]RECEIVE_IMMEDIATE	X	X	X	Note 5	X	X	X	X
[MC_]REQUEST_TO_SEND	X	X	X	R	C	X	X	P
[MC_]SEND_DATA AP_OK AP_ERROR	X	S R	S	X	X	X	X	X
[MC_]SEND_ERROR AP_OK AP_ERROR	X	S R	S	S	S	S	S	S
[MC_]TEST_RTS	X	S	S	R	C	C	C	P

Notes:

1. In the Return codes column of the table, the abbreviation AP_ERROR is used for the following return codes:
   AP_PROG_ERROR_TRUNC
   AP_PROG_ERROR_NO_TRUNC
   AP_PROG_ERROR_PURGING
   AP_SVC_ERROR_TRUNC
   AP_SVC_ERROR_NO_TRUNC
   AP_SVC_ERROR_PURGING.
2. The conversation will always enter Reset state if any of the following return codes are received:
   AP_ALLOCATION_ERROR
   AP_COMM_SUBSYSTEM_ABENDED
   AP_COMM_SUBSYSTEM_NOT_LOADED
   AP_CONV_FAILURE_RETRY
   AP_CONV_FAILURE_NO_RETRY
   AP_DEALLOC_ABEND
   AP_DEALLOC_ABEND_PROG
   AP_DEALLOC_ABEND_SVC

AP_DEALLOC_ABEND_TIMER

AP_DEALLOC_NORMAL

3. The following non-OK return codes do not cause any state change. The conversation always remains in the state in which the verb was issued:

   AP_CONVERSATION_TYPE_MIXED

   AP_PARAMETER_CHECK

   AP_STATE_CHECK

   AP_TP_BUSY

   AP_UNEXPECTED_SYSTEM_ERROR

   AP_UNSUCCESSFUL

4. After **[MC_]RECEIVE_AND_POST** has been issued and received the initial **primary_rc** of AP_OK, the conversation changes to Pending Post state. Once the supplied callback routine has been called to indicate that the verb has completed, the new conversation state depends on the **primary_rc** and **what_rcvd** parameters as in Note 5.
5. The state change after one of the **RECEIVE** verbs depends on both the **primary_rc** and **what_rcvd** parameters.

   If the **primary_rc** parameter is AP_PROG_ERROR*, AP_SVC_ERROR*, or ([MC_]RECEIVE_IMMEDIATE only) AP_UNSUCCESSFUL, the new state is RECEIVE.

   If the **primary_rc** parameter is AP_DEALLOC*, the new state is RESET.

   If the **primary_rc** parameter is AP_OK, the new state depends on the value of the **what_rcvd** parameter:

   Receive state
   AP_DATA, AP_DATA_COMPLETE, AP_DATA_INCOMPLETE

   Send state
   AP_SEND

   Send Pending state
   AP_DATA_SEND, AP_DATA_COMPLETE_SEND

   Confirm state
   AP_CONFIRM_WHAT_RECEIVED, AP_DATA_CONFIRM, AP_DATA_COMPLETE_CONFIRM

   Confirm Send state
   AP_CONFIRM_SEND, AP_DATA_CONFIRM_SEND, AP_DATA_COMPLETE_CONFIRM_SEND

   Confirm Deallocate state
   AP_CONFIRM_DEALLOCATE, AP_DATA_CONFIRM_DEALLOCATE, AP_DATA_COMPLETE_CONFIRM_DEALL

For information on half-duplex conversation state transitions, see Table 23 on page 337 .

Table 24. APPC Full-Duplex Conversation State Transitions

Verb Return Codes	**Reset (T)**	**Send Receive (SR)**	**Send Only (S)**	**Receive Only (R)**
[MC_]ALLOCATE				
AP_OK	SR	/	/	/
(other)	T			
CANCEL_CONVERSATION	X	T	T	T
[MC_]DEALLOCATE (Abend)	X	T	T	T
[MC_]DEALLOCATE (Flush)	X	R	T	X
[MC_]FLUSH	X	SR	S	X
[MC_]GET_ATTRIBUTES	X	SR	S	R
GET_STATE	X	SR	S	R
GET_TYPE	X	SR	S	R
RECEIVE_ALLOCATE				
AP_OK	SR	/	/	/
(other)	T			
[MC_]RECEIVE_AND WAIT				
AP_OK	X	SR	X	R
AP_ERROR	X	SR	X	R
AP_DEALLOC_NORMAL	X	S	X	T
RECEIVE_EXPEDITED_DATA	X	SR	S	R
[MC_]RECEIVE_ IMMEDIATE				
AP_OK	X	SR	X	R
AP_ERROR	X	SR	X	R
AP_DEALLOC_NORMAL	X	S	X	T
[MC_]SEND_DATA				
AP_OK	X	SR	S	X
AP_ERROR_INDICATION	X	SR	T	X
[MC_]SEND_ERROR				
AP_OK	X	SR	S	X
AP_ERROR_INDICATION	X	SR	T	X

Notes:

1. In the Return codes column of the table, the abbreviation AP_ERROR is used for the following return codes:

   AP_PROG_ERROR_TRUNC

   AP_PROG_ERROR_NO_TRUNC

   AP_SVC_ERROR_TRUNC

   AP_SVC_ERROR_NO_TRUNC

2. The conversation will always enter Reset state if any of the following return codes are received:

   AP_ALLOCATION_ERROR
   AP_COMM_SUBSYSTEM_ABENDED
   AP_COMM_SUBSYSTEM_NOT_LOADED
   AP_CONV_FAILURE_RETRY
   AP_CONV_FAILURE_NO_RETRY
   AP_DEALLOC_ABEND
   AP_DEALLOC_ABEND_PROG
   AP_DEALLOC_ABEND_SVC
   AP_DEALLOC_ABEND_TIMER

3. The following non-OK return codes do not cause any state change. The conversation always remains in the state in which the verb was issued:

   AP_CONVERSATION_TYPE_MIXED
   AP_PARAMETER_CHECK
   AP_STATE_CHECK
   AP_TP_BUSY
   AP_UNEXPECTED_SYSTEM_ERROR
   AP_UNSUCCESSFUL

Glossary

A

abend. (1) Abnormal end of task; the termination of a task before its completion because of an error condition that cannot be resolved by recovery facilities while the task is executing. (2) Synonym for *abnormal termination*.

abnormal termination. (1) The cessation of processing prior to planned termination. (T) (2) A system failure or operator action that causes a job to end unsuccessfully. (3) Synonymous with *abend*.

accept. (1) In a VTAM application program, to establish a session with a logical unit (LU) in response to a CINIT request from a system services control point (SSCP). The session-initiation request may begin when a terminal user logs on, a VTAM application program issues a macroinstruction, or a VTAM operator issues a command. See also *acquire*. (2) An SMP process that moves distributed code and MVS-type programs to the distribution libraries.

ACCESS. In the Simple Network Management Protocol (SNMP), the clause in a Management Information Base (MIB) module that defines the minimum level of support that a managed node provides for an object.

access method. (1) A technique, implemented in software, that controls the flow of information through a network. (2) A technique for moving data between main storage and input/output devices.

ACF/VTAM. Advanced Communications Function for the Virtual Telecommunications Access Method. Synonym for *VTAM*.

acknowledgment. (1) The transmission, by a receiver, of acknowledge characters as an affirmative response to a sender. (T) (2) An indication that an item sent was received.

acquire. (1) In VTAM, to take over resources that were formerly controlled by an access method in another domain or to resume control of resources that were controlled by that domain but released. Contrast with *release*. See also *resource takeover*. (2) In a VTAM application program, to initiate and establish a session with another logical unit (LU). The acquire process begins when the application program issues a macroinstruction. See also *accept*.

action. (1) An operation on a managed object, the semantics of which are defined as part of the managed object class definition. (2) In the AIX operating system, a defined task that an application performs. An action modifies the properties of an object or manipulates the object in some way.

activate. To make a resource ready to perform its function. Contrast with *deactivate*.

active. (1) Operational. (2) Pertaining to a node or device that is connected or is available for connection to another node or device. (3) The state of a resource when it has been activated and is operational.

ACTLU. Activate logical unit. In SNA, a command used to start a session on a logical unit.

ACTPU. Activate physical unit. In SNA, a command used to start a session on a physical unit.

adapter. (1) A hardware component that must be installed in the personal computer to connect to the SDLC, LAN, asynchronous, DFT, or other communication attachment (possibly connecting through a modem). (2) A part that electrically or physically connects a device to a computer or to another device.

adaptive pacing. Synonym for *adaptive session-level pacing*.

adaptive session-level pacing. A form of session-level pacing in which session components exchange pacing windows that may vary in size during the course of a session. This allows transmission within a network to adapt dynamically to variations in availability and demand of buffers on a session-by-session basis. Session-level pacing occurs within independent stages along the session path according to local congestion at the intermediate and endpoint nodes. Synonymous with *adaptive pacing* and *adaptive session pacing*. Contrast with *fixed session-level pacing*.

adaptive session pacing. Synonym for *adaptive session-level pacing*.

address. In data communication, the unique code assigned to each device, workstation, or user connected to a network.

Administrative Domain. A collection of hosts and routers, and the interconnecting networks, managed by a single administrative authority.

Advanced Peer-to-Peer Networking (APPN). An extension to SNA featuring (a) greater distributed network control that avoids critical hierarchical dependencies, thereby isolating the effects of single points of failure; (b) dynamic exchange of network topology information to foster ease of connection, reconfiguration, and adaptive route selection; (c) dynamic definition of network resources; and (d) automated resource registration and directory lookup. APPN extends the LU 6.2 peer orientation for end-user services to network control and supports multiple LU types, including LU 2, LU 3, and LU 6.2.

Advanced Peer-to-Peer Networking (APPN) end node. A node that provides a broad range of end-user services and supports sessions between its local control point (CP) and the CP in an adjacent network node. It uses these sessions to dynamically register its resources with the adjacent CP (its network node server), to send and receive directory search requests, and to obtain management services. An APPN end node can also attach to other end nodes.

Advanced Peer-to-Peer Networking (APPN) network. A collection of interconnected network nodes and their client end nodes.

Advanced Peer-to-Peer Networking (APPN) network node. A node that offers a broad range of end-user services and that can provide the following:

- Distributed directory services, including registration of its domain resources to a central directory server
- Topology database exchanges with other APPN network nodes, enabling network nodes throughout the network to select optimal routes for LU-LU sessions based on requested classes of service
- Session services for its local LUs and client end nodes
- Intermediate routing services within an APPN network

Advanced Peer-to-Peer Networking (APPN) node. An APPN network node or an APPN end node.

advanced program-to-program communication (APPC). (1) (2) An LU 6.2 logical unit protocol implementation of SNA that lets interconnected systems share programming tasks. The general facility characterizing the LU 6.2 architecture and its various implementations in products. (3) Sometimes used to refer to the LU 6.2 architecture and its product implementations as a whole, or to an LU 6.2 product feature in particular, such as an APPC application programming interface.

AID. Attention identifier.

allocate. (1) An LU 6.2 application programming interface (API) verb used to assign a session to a conversation for the conversation's use. (2) Contrast with *deallocate*.

all points addressable (APA). In computer graphics, pertaining to the ability to address and display or not display each picture element (pel) on a display surface.

American National Standards Institute (ANSI). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States. (A)

AND operation. Synonym for *conjunction*.

ANSI. American National Standards Institute.

AP. Alternate printer.

APA. All points addressable.

API. Application programming interface.

APPC. Advanced program-to-program communication.

application. A collection of software components used to perform specific types of user-oriented work on a computer.

application program. (1) A program written for or by a user that applies to the user's work, such as a program that does inventory control or payroll. (2) A program used to connect and communicate with stations in a network, enabling users to perform application-oriented activities.

application programming interface (API). (1) (2) A defined programming language interface between an IBM system control program or an IBM-licensed program and the program user. The set of programming language constructs or statements that can be coded in an application program to obtain the specific functions and services provided by an underlying operating system or service program. (3) In VTAM, the language structure used in control blocks so that application programs can reference them and be identified to VTAM.

Apply. A push button that carries out the selected choices in a window without closing the window.

APPN. Advanced Peer-to-Peer Networking.

APPN end node. See *Advanced Peer-to-Peer Networking (APPN) end node*.

argument. A parameter passed between a calling program and a called program.

ASCII (American National Standard Code for Information Interchange). The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. (A)

ASYNC. Asynchronous.

asynchronous (ASYNC). (1) Pertaining to two or more processes that do not depend upon the occurrence of specific events such as common timing signals. (T) (2) Without regular time relationship; unexpected or unpredictable with respect to the execution of program instructions.

asynchronous operation. Simultaneous operations of software or hardware. In software, an operation, such as a request for session establishment or data transfer, in which the application program is allowed to continue execution while the operation is performed. The access method informs the application program after the operation is completed. Contrast with *synchronous operation*.

asynchronous request. In VTAM, a request for an asynchronous operation. Contrast with *synchronous request*.

attach. (1) In programming, to create a task that can be executed asynchronously with the execution of the mainline code. (2) To connect a device logically to a ring network.

attention identifier (AID). A character in a data stream indicating that the user has pressed a key, such as the Enter key, that requests an action by the system.

attribute. Variable data that is logically a part of an object and that represents a property of the object. For example, a serial number is an attribute of an equipment object.

authorization. (1) In computer security, the right granted to a user to communicate with or make use of a computer system. (T) (2) An access right. (3) The process of granting a user either complete or restricted access to an object, resource, or function.

automatic activation. In VTAM, the activation of links and link stations in adjacent subarea nodes as a result of channel device name or RNAME specifications related to an activation command that names a subarea node. See also *direct activation*.

autotask. (1) An unattended NetView operator station task that does not require a terminal or a logged-on user. Autotasks can run independently of VTAM and are typically used for automated console operations. (2) Contrast with *logged-on operator*.

available. In VTAM, pertaining to a logical unit that is active, connected, enabled, and not at its session limit.

B

background process. (1) A process that does not require operator intervention but can be run by the computer while the workstation is used to do other work. (2) In the AIX operating system, a mode of program execution in which the shell does not wait for program completion before prompting the user for another command. (3) Contrast with *foreground process*.

backout. In IMS/VS, the process of removing all the database updates performed by an application program that has terminated abnormally.

base set. (1) The set of functions, including verbs, parameters, return codes, and what-received indications, that is supported by all products that implement a particular architecture. (2) Contrast with *option set*.

basic conversation. An LU 6.2 conversation type specified by the allocating transaction program. Transaction programs using basic conversation have available to them a wider variety of LU 6.2 functions, but they are responsible for more of their own error recovery and must manage details of the data stream used on the conversation. Contrast with *mapped conversation*.

basic information unit (BIU). In SNA, the unit of data and control information passed between half-sessions. It consists of a request/response header (RH) followed by a request/response unit (RU).

batch. (1) An accumulation of data to be processed. (2) A group of records or data processing jobs brought together for processing or transmission. (3) Pertaining to activity involving little or no user action. Contrast with *interactive*.

batch file. (1) In IBM DOS, a file containing commands to be processed sequentially. A DOS batch file must have a .BAT extension in order to be executable. (2) A file that contains a series of commands to be processed sequentially.

begin bracket. In SNA, the value (binary 1) of the begin-bracket indicator in the request header (RH) of the first request in the first chain of a bracket; the value denotes the start of a bracket. Contrast with *end bracket*. See also *bracket*.

behavior. (1) Ideally, a collection of assertions that describe the allowed states that a managed object can assume. An assertion can be a precondition, a postcondition, or an invariant. In practice, the behavior is often an informal description of the semantics of attributes, operations, and notifications. (2) The way in which managed objects, name bindings, attributes, notifications, and operations interact with the actual resources that they model and with each other.

bid. In the contention form of invitation or selection, an attempt by the computer or by a station to gain control of a line in order to transmit data.

bidder. See *bidder session*.

bidder session. The half-session defined at session activation as having to request and receive permission from the other half-session to begin a bracket. Contrast with *first-speaker session*. Synonym for *contention-loser session*.

binary. Pertaining to the base two system of numbers. The binary digits are 0 and 1. Executable files are generally in binary format rather than the character string format that text files are composed of.

BIND. In SNA, a request to activate a session between two logical units (LUs). See also *session activation request*. Contrast with *UNBIND*.

bis. German (Federal Republic) preliminary standard.

bit. Either of the digits 0 or 1 when used in the binary numeration system. (T)

BIU. Basic information unit.

BIU segment. In SNA, the portion of a basic information unit (BIU) that is contained within a path information unit (PIU). It consists of either (a) a request/response header (RH) followed by all or a part of a request/response unit (RU) or (b) a part of an RU. Synonymous with *segment*.

block. A string of data elements recorded or transmitted as a unit. The elements may be characters, words, or physical records. (T)

blocking mode. (1) A way of requesting a service over an interface so that if the request cannot be completed immediately, the requesting process is suspended until the request is completed. (2) Contrast with *nonblocking mode*.

bracket. In SNA, one or more chains of request units and their responses that are exchanged between two session partners and that represent a transaction between them. A bracket must be completed before another bracket can be started. Examples of brackets are database inquiries/replies, update transactions, and remote job entry output sequences to workstations.

bracket protocol. In SNA, a data flow control protocol in which exchanges between two session partners are achieved through the use of brackets, with one partner designated at session activation as the first speaker and the other as the bidder. The bracket protocol involves bracket initiation and termination rules.

bridge. (1) A functional unit that interconnects two local area networks that use the same logical link control protocol but may use different medium access control protocols. (T) (2) A functional unit that interconnects multiple LANs (locally or remotely) that use the same logical link control protocol but that can use different medium access control protocols. A bridge forwards a frame to another bridge based on the medium access control (MAC) address. (3) In the connection of local loops, channels, or rings, the equipment and techniques used to match circuits and to facilitate accurate data transmission. (4) Contrast with *gateway* and *router*.

browse. (1) To look at records in a file. (2) In the NetView Graphic Monitor Facility, to open a view that cannot receive status changes from the NetView program. Contrast with *monitor*.

buffer. (1) A routine or storage used to compensate for a difference in rate of flow of data, or time of occurrence of events, when transferring data from one device to another. (A) (2) A portion of storage used to hold input or output data temporarily.

bus. (1) A facility for transferring data between several devices located between two end points, only one device being able to transmit at a given moment. (T) (2) A computer configuration in which processors are interconnected in series.

bus master. A device or subsystem that controls data transfers between itself and a subordinate.

byte. (1) A string that consists of a number of bits, treated as a unit, and representing a character. (T) (2) A binary character operated upon as a unit and usually shorter than a computer word. (A) (3) A group of 8 adjacent binary digits that represent one EBCDIC character.

C

call. (1) The action of bringing a computer program, a routine, or a subroutine into effect, usually by specifying the entry conditions and jumping to an entry point. (I) (A) (2) In data communication, the actions necessary to make a connection between two stations on a switched line. (3) In communications, a conversation between two users. (4) To transfer control to a procedure, program, routine, or subroutine. (5) To attempt to contact a user, regardless of whether the attempt is successful.

callback. In the AIX operating system, a procedure that is called if and when certain specified conditions are met.

calling. (1) The process of transmitting selection signals in order to establish a connection between data stations. (I) (A) (2) In X.25 communications, pertaining to the location or user that makes a call.

Cancel. A push button that removes a window without applying any changes made in that window.

case-sensitive. Pertaining to the ability to distinguish between uppercase and lowercase letters.

CD. Compact disc.

chain. (1) A group of logically linked user data records processed by LU 6.2. (2) A group of request units delimited by begin-chain and end-chain. Responses are always single-unit chains. See *RU chain*.

channel. (1) A path along which signals can be sent, for example, data channel, output channel. (A) (2) In data communication, a means of one-way transmission. (3) A functional unit, controlled by the processor, that handles the transfer of data between processor storage and local peripheral equipment.

channel-attached. (1) Pertaining to the attachment of devices directly by input/output channels to a host processor. (2) Pertaining to devices attached to a controlling unit by cables, rather than by telecommunication lines. (3) Contrast with *link-attached*. (4) Synonymous with *local*.

character set. A finite group of characters defined for a keyboard or output device.

child. Pertaining to a secured resource, either a file or library, that uses the user list of a parent resource. A child resource can have only one parent resource. A child is a process, started by a parent process, that shares the resources of the parent process. Contrast with *parent*.

child process. In the AIX and OS/2 operating systems, a process, started by a parent process, that shares the resources of the parent process. See also *fork*.

CICS. Customer Information Control System.

circuit. (1) One or more conductors through which an electric current can flow. See *physical circuit* and *virtual circuit*. (2) A logic device.

C language. A language used to develop software applications in compact, efficient code that can be run on different types of computers with minimal change.

class. (1) In object-oriented design or programming, a group of objects that share a common definition and that therefore share common properties, operations, and behavior. Members of the group are called instances of the class. (2) In the AIX operating system, pertaining to the I/O characteristics of a device. System devices are classified as block or character devices.

cleanup. In SNA products, a network services request, sent by a system services control point (SSCP) to a logical unit (LU), that causes a particular LU-LU session with that LU to be ended immediately without requiring the participation of either the other LU or its SSCP.

clear to send (CTS). In data communication, a signal raised by data circuit-terminating equipment (DCE) when it is ready to accept data, usually in response to request to send (RTS) being raised.

client. (1) A functional unit that receives shared services from a server. (T) (2) A user. (3) In an AIX distributed file system environment, a system that is dependent on a server to provide it with programs or access to programs. (4) Synonymous with *requester*.

client/server. In communications, the model of interaction in distributed data processing in which a program at one site sends a request to a program at another site and awaits a response. The requesting program is called a client; the answering program is called a server.

Close. A choice that removes a window and all of the windows associated with it from the workplace. For example, if a user is performing a task in a window and a message appears, or the user asks for help, both the message and the help windows disappear when the user closes the original window.

CNOS. Change number of sessions.

code page. (1) A table that defines a coded character set by assignment of a character meaning to each code point in the table for a language or a country. (2) A mapping between characters and their internal binary representation. (3) An assignment of graphic characters and control function meanings to all code points; for example, assignment of characters and meanings to 256 code points for an 8-bit code, assignment of characters and meanings to 128 code points for a 7-bit code. (4) In the Print Management Facility, a font library member that associates code points and character identifiers. A code page also identifies invalid code points. (5) A particular assignment of hexadecimal identifiers to graphic characters. (6) In AFP support, a font file that associates code points and graphic character identifiers.

command. (1) A request from a terminal for the performance of an operation or the execution of a particular program. (2) In SNA, any field set in the transmission header (TH), request header (RH), and sometimes portions of a request unit (RU), that initiates an action or that begins a protocol; for example: (a) Bind Session (session-control request unit), a command that activates an LU-LU session, (b) the change-direction indicator in the RH of the last RU of a chain, (c) the virtual route reset window indicator in an FID4 transmission header.

command area. In Basic CUA architecture, the area on a panel that contains the command entry field.

command line. (1) On a display screen, a display line usually at the bottom of the screen, in which only commands can be entered. (2) In CUA architecture, deprecated term for *command area*.

command prompt. A displayed character or string of characters that indicates that a user may enter a command to be processed.

Common Programming Interface for Communications (CPI-C). An evolving application programming interface (API), embracing functions to meet the growing demands from different application environments and to achieve openness as an industry standard for communications programming. CPI-C provides access to interprogram services such as (a) sending and receiving data, (b) synchronizing processing between programs, and (c) notifying a partner of errors in the communication.

Common Services API. In Communications Manager/2, application programming interface (API) verbs used to access services provided by Communications Manager/2 for user-written programs.

common transport semantics (CTS). The layer of the Networking Blueprint above the transport layer that makes the services of transport providers available to the transport user. See also *transport-layer protocol boundary (TLPB)*.

Communications Manager/2. See *Communications Server* and *Personal Communications product family*. The function of the Communications Manager/2 product has been incorporated into the Communications Server product and the Personal Communications product family.

Communications Server. An IBM licensed program that supports the development and use of application programs involving two or more connected systems or workstations. The Communications Server provides multiple concurrent connectivities using different protocols to connect the application programs to other systems and workstations. It supports several application programming interfaces (APIs) that may be called concurrently and are designed for client/server and distributed application programs. The Communications Server includes the necessary interfaces for network management.

compact disc (CD). (1) A disc, usually 4.75 inches in diameter, from which data is read optically by means of a laser. (2) A disc with information stored in the form of pits along a spiral track. The information is decoded by a compact-disc player and interpreted as digital audio data, which most computers can process.

compile. (1) To translate all or part of a program expressed in a high-level language into a computer program expressed in an intermediate language, an assembly language, or a machine language. (T) (2) To prepare a machine language program from a computer program written in another programming language by making use of the overall logic structure of the program, or generating more than one computer instruction for each symbolic statement, or both, as well as performing the function of an assembler. (A) (3) To translate a source program into an executable program (an object program). (4) To translate a program written in a high-level programming language into a machine language program.

compiler. (1) A program that translates a source program into an executable program (an object program). (2) A program that decodes instructions written as pseudo codes and produces a machine language program to be executed at a later time.

component. Hardware or software that is part of a functional unit.

computer. A functional unit that can perform substantial computations, including numerous arithmetic operations and logic operations without human intervention during a run. In information processing, the term computer usually describes a digital computer. A computer may consist of a stand-alone unit or may consist of several interconnected units. (T)

configuration. (1) The manner in which the hardware and software of an information processing system are organized and interconnected. (T) (2) The devices and programs that make up a system, subsystem, or network. (3) In Personal Communications, the arrangement of personal computers connected to one or more host systems by one or more attachment types. Examples are: SDLC, LAN, ASYNCH, X.25, or DFT.

configuration file. A file that specifies the characteristics of a system device or network.

configuration management. The control of information necessary to identify both physical and logical information system resources and their relationship to one another.

configuration services. One of the types of network services in a control point (SSCP, NNCP, or ENCP). Configuration services activates, deactivates, and records the status of physical units, links, and link stations.

configure. To describe to a system the devices, optional features, and programs installed on the system.

congestion. See *network congestion*.

conjunction. The Boolean operation whose result has the Boolean value 1 if and only if each operand has the Boolean value 1. (I) (A) Synonymous with *AND operation*.

connected. In VTAM, the state of a physical unit (PU) or a logical unit (LU) that has an active physical path to the host processor containing the system services control point (SSCP) that controls the respective PU or LU.

connection. (1) In data communication, an association established between functional units for conveying information. (I) (A) (2) In SNA, the network path that links together two logical units (LUs) in different nodes to enable them to establish communications. (3) In TCP/IP, the path between two protocol applications that provides reliable data stream delivery service. In the Internet, a connection extends from a TCP application on one system to a TCP application on another system.

connection point manager. In SNA, a component of the transmission control layer that (a) performs session-level pacing of normal-flow requests, (b) checks sequence numbers of received request units, (c) verifies that request units do not exceed the maximum permissible size, (d) routes incoming request units to their destinations in the half-session, and (e) enciphers and deciphers FMD request units when cryptography is selected. The connection point manager coordinates the normal and expedited flows for one half-session.

Note: The sending connection point manager in a half-session builds the request/response header (RH) for outgoing request/response units (RUs), and the receiving connection point manager interprets the request/response headers that precede incoming request/response units.

connectivity. (1) The capability of a system or device to be attached to other systems or devices without modification. (T) (2) The capability to attach a variety of functional units without modifying them.

contention. In a session, a situation in which both NAUs attempt to initiate the same action at the same time, such as when both attempt to send data in a half-duplex protocol (half-duplex contention), or both attempt to start a bracket (bracket contention). At session initiation, one NAU is defined to be the contention winner; its action will take precedence when contention occurs. The contention loser must get explicit or implicit permission from the contention winner to begin its action.

contention-loser session. (1) To an NAU, a session for which it was defined during session initiation to be the contention loser. (2) Synonymous with *bidder session*.

contention-winner session. (1) To an NAU, a session for which it was defined during session initiation to be the contention winner. (2) Synonymous with *first-speaker session*.

control block. (1) A storage area used by a computer program to hold control information. (I) (2) In the IBM Token-Ring Network, a specifically formatted block of information provided from the application program to the Adapter Support Interface to request an operation.

control character. A character whose occurrence in a particular context specifies a control function. (T)

control point (CP). (1) A component of an APPN or LEN node that manages the resources of that node. In an APPN node, the CP is capable of engaging in CP-CP sessions with other APPN nodes. In an APPN network node, the CP also provides services to adjacent end nodes in the APPN network. (2) A component of a node that manages resources of that node and optionally provides services to other nodes in the network. Examples are a system services control point (SSCP) in a type 5 subarea node, a network node control point (NNCP) in an APPN network node, and an end node control point (ENCP) in an APPN or LEN end node. An SSCP and an NNCP can provide services to other nodes.

Control Program (CP). In VM/ESA, a component that manages the resources of a single computer so multiple computing systems appear to exist. Each of these apparent systems, or virtual machines, is the functional equivalent of an IBM System/370, 370-XA, or ESA computer.

conversation. A logical connection between two transaction programs using an LU 6.2 session. Conversations are delimited by brackets to gain exclusive use of a session.

conversation group ID. An identifier of a particular session between two specific LUs or CPs. Specified in a conversation allocation command, the conversation group ID allows the requested conversation to be allocated to the identified session. This then permits a pair of transaction programs (one in each LU) to serially share use of the designated session with a group of related pairs of transaction programs that use the same conversation group ID.

coordinated universal time (UTC). The time scale, based on the Système International (SI) second, as defined and recommended by the Comité Consultatif International de la Radio (CCIR) and maintained (using an atomic clock) by the Bureau International des Poids et Mesures (BIPM).

Note: The Système International is based on three fundamental units of measure—the meter, the kilogram, and the second—and is sometimes called the “MKS system” because of these units.

For most practical purposes, coordinated universal time (UTC) is equivalent to the mean solar time at the prime meridian (0 degrees longitude) of Greenwich, England, known as *Greenwich mean time (GMT)*. Synonymous with *Z time* and *Zulu time*.

Copy. A choice that places a copy of a selected object onto the clipboard.

correlator. Information that identifies a relation among things. An example is a variable field of a response that identifies the corresponding request.

CP. (1) Control point. (2) In VM, Control Program.

CRV. Cryptography verification.

cryptography verification (CRV) request. A request unit sent by the primary logical unit (PLU) to the secondary logical unit (SLU) as part of cryptographic session establishment, to allow the SLU to verify that the PLU is using the correct session cryptography key and initialization vector (IV).

CS. Current state.

CTS. (1) Common transport semantics. (2) Clear to send.

Customer Information Control System (CICS). An IBM licensed program that enables transactions entered at remote terminals to be processed concurrently by user-written application programs. It includes facilities for building, using, and maintaining databases.

D

DACTLU. Deactivate logical unit.

DACTPU. Deactivate physical unit.

DAF. Destination address field.

DAF'. Destination address field prime.

data. (1) A re-interpretable representation of information in a formalized manner suitable for communication, interpretation, or processing. Operations can be performed upon data by humans or by automatic means. (T) (2) Any representations such as characters or analog quantities to which meaning is or might be assigned. (A) (3) A representation of facts or instructions in a form suitable for communication, interpretation, or processing by human or automatic means. Data include constants, variables, arrays, and character strings.

Note: Programmers make a distinction between instructions and the data they operate on; however, in the usual sense of the word, data includes programs and program instructions.

data circuit. (1) A pair of associated transmit and receive channels that provide a means of two-way data communication. (I) (2) In SNA, synonym for *link connection*. (3) See also *physical circuit* and *virtual circuit*.

Notes:

1. Between data switching exchanges, the data circuit may include data circuit-terminating equipment (DCE), depending on the type of interface used at the data switching exchange.
2. Between a data station and a data switching exchange or data concentrator, the data circuit includes the data circuit-terminating equipment at the data station end, and may include equipment similar to a DCE at the data switching exchange or data concentrator location.

data flow control (DFC). In SNA, a request/response unit (RU) category used for requests and responses exchanged between the data flow control layer in one half-session and the data flow control layer in the session partner.

data flow control (DFC) layer. In SNA, the layer within a half-session that controls whether the half-session can send, receive, or concurrently send and receive, request units (RUs); groups related RUs into RU chains; delimits transactions via the bracket protocol; controls the interlocking of requests and responses in accordance with control modes specified at session activation; generates sequence numbers; and correlates requests and responses.

data flow control (DFC) protocol. In SNA, the sequencing rules for requests and responses by which network addressable units (NAUs) in a session coordinate and control data transfer and other operations; for example, bracket protocol.

data link. In SNA, synonym for *link*.

data link control (DLC). A set of rules used by nodes on a data link (such as an SDLC link or a token ring) to accomplish an orderly exchange of information.

data link control (DLC) layer. In SNA, the layer that consists of the link stations that schedule data transfer over a link between two nodes and perform error control for the link. Examples of data link control are SDLC for serial-by-bit link connection and data link control for the System/370 channel.

Note: The DLC layer is usually independent of the physical transport mechanism and ensures the integrity of data that reaches the higher layers.

data set. (1) Synonym for *file*. (2) Deprecated term for *modem*.

data stream. (1) All information (data and control commands) sent over a data link usually in a single read or write operation. (2) A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format.

data types. In the NetView program, a description of the organization of panels. Data types are alerts, events, and statistics. Data types are combined with resource types and display types to describe the NetView program's display organization. See also *display types* and *resource types*.

DBCS. Double-byte character set.

deactivate. To take a resource of a node out of service, rendering it inoperable, or to place it in a state in which it cannot perform the functions for which it was designed. Contrast with *activate*.

deallocate. An LU 6.2 application programming interface (API) verb that terminates a conversation, thereby freeing the session for a future conversation. Contrast with *allocate*.

decryption. In computer security, transforming encoded text or ciphertext into plaintext.

default. Pertaining to an attribute, condition, value, or option that is assumed when none is explicitly specified. (I)

definite response (DR). In SNA, a protocol requested in the form-of-response-requested field of the request header that directs the receiver of the request to return a response unconditionally, whether positive or negative, to that request chain. Contrast with *exception response* and *no response*.

definition file. A file loaded by default that contains keyboard and screen color characteristics that can be adapted for a particular session. The lines of a definition file must be coded according to a strict layout.

definition statement. (1) In VTAM, the statement that describes an element of the network. (2) In NCP, a type of instruction that defines a resource to the NCP. (3) See also *macroinstruction*. (4) See Figure 9, Figure 10, and Figure 11.

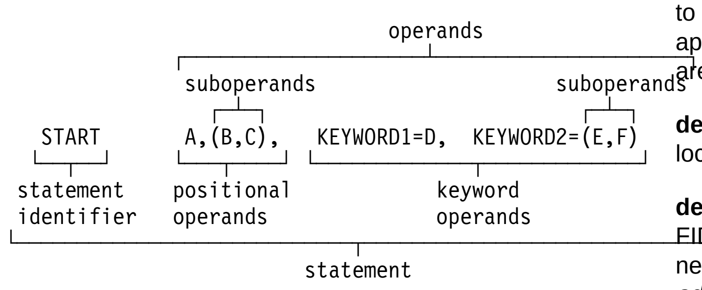


Figure 9. Example of a Language Statement

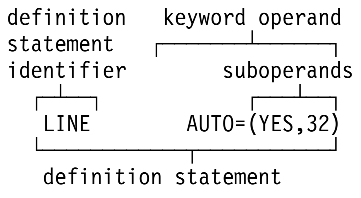


Figure 10. Example of an NCP Definition Statement

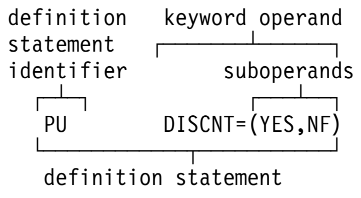


Figure 11. Example of a VTAM Definition Statement

dependent LU. See *SSCP-dependent LU*.

dependent LU requester (DLUR). An APPN end node or an APPN network node that owns dependent LUs, but requests that a dependent LU server provide the SSCP services for those dependent LUs.

destination. (1) Any point or location, such as a node, station, or a particular terminal, to which information is to be sent. (2) An external logical unit (LU) or application program to which messages or other data are directed.

destination address. A code that identifies the location to which information is to be sent.

destination address field (DAF). In SNA, a field in a FID0 or FID1 transmission header that contains the network address of the destination. Contrast with *origin address field (OAF)*.

device. A mechanical, electrical, or electronic contrivance with a specific purpose.

DFC. Data flow control.

dialog. (1) The interaction between a user and a computer. (2) In an interactive system, a series of related inquiries and responses similar to a conversation between two people. (3) In the AIXwindows Toolkit, a two-way text interface between an application and its user. The interface takes the form of a collection of widgets and gadgets, including a DialogShell widget, a BulletinBoard widget (or a subclass of a BulletinBoard widget or some other container widget), plus various children, including Label, PushButton, and Text widgets.

direct activation. In VTAM, the activation of a resource as a result of an activation command specifically naming the resource. See *automatic activation*. Contrast with *indirect activation*.

directory. (1) A table of identifiers and references to the corresponding items of data. (I) (A) (2) A named hierarchical grouping of files in a file system. (3) A database in an APPN node that lists names of resources (in particular, logical units) and records the

CP name of the node where each resource is located. See *distributed directory database* and *local directory database*.

disable. To make nonfunctional.

display. A visual presentation of data. (I) (A)

display levels. Synonym for *display types*.

display types. In the NetView program, a concept to describe the organization of panels. Display types are defined as total, most recent, user action, and detail. Display types are combined with resource types and data types to describe NetView's panel organization. See also *data types* and *resource types*. Synonymous with *display levels*.

distributed directory database. The complete listing of all the resources in the network as maintained in the individual directories scattered throughout an APPN network. Each node has a piece of the complete directory, but it is not necessary for any one node to have the entire list. Entries are created, modified, and deleted through system definition, operator action, automatic registration, and ongoing network search procedures. Synonymous with *distributed network directory* and *network directory database*.

distributed network directory. Synonym for *distributed directory database*.

distributed processing. Processing that takes place across two or more linked systems.

DLC. Data link control.

DLL. Dynamic link library.

domain. (1) That part of a computer network in which the data processing resources are under common control. (T) (2) A set of servers that allocate shared network resources within a single logical system. (3) In SNA, see *end node domain*, *network node domain*, and *system services control point (SSCP) domain*. (4) In Open Systems Interconnection (OSI), a part of a distributed system or a set of managed objects to which a common policy applies. (5) See *Administrative Domain* and *domain name*.

domain name. In the Internet suite of protocols, a name of a host system. A domain name consists of a sequence of subnames separated by a delimiter character. For example, if the fully qualified domain name (FQDN) of a host system is `ralvm7.vnet.ibm.com`, each of the following is a domain name:

- `ralvm7.vnet.ibm.com`
- `vnet.ibm.com`
- `ibm.com`

domain operator. In a multiple-domain network, the person or program that controls operation of resources controlled by one system services control point (SSCP). See also *network operator*.

DOS. Disk Operating System. See *IBM Disk Operating System*.

DOS session. A session in which a personal computer operates as a stand-alone computer, running under Disk Operating System (DOS). See *host session*.

double-byte character set (DBCS). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS. Contrast with *single-byte character set (SBCS)*.

drain. To honor pending allocation requests before deactivating sessions with a partner logical unit. This applies to LU 6.2 only.

drive. A peripheral device, especially one that has addressed storage media.

duplex. Pertaining to communication in which data can be sent and received at the same time. Synonymous with *full-duplex*. Contrast with *half-duplex*.

dynamic. (1) In programming languages, pertaining to properties that can only be established during the execution of a program; for example, the length of a variable-length data object is dynamic. (I) (2) Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time. (3) Contrast with *static*.

dynamic linking. In the OS/2 operating system, the delayed connection of a program to a routine until load time or run time.

dynamic link library (DLL). A file containing executable code and data bound to a program at load time or run time, rather than during linking. The code and data in a dynamic link library can be shared by several applications simultaneously.

E

EBCDIC. Extended binary-coded decimal interchange code. A coded character set of 256 8-bit characters.

EC. Engineering change.

Edit. A choice on a menu bar that provides access to other choices that enable a user to modify data.

element. (1) A field in the network address. (2) In SNA, the particular resource within a subarea that is identified by an element address. See also *subarea*.

emulator. A program that allows a device to operate as if it were a different type of device. Personal Communications, for example, allows supported personal computers and printers to operate as if they were 3270-series workstations.

enabled. (1) Pertaining to a state of the processing unit that allows the occurrence of certain types of interruptions. (2) Pertaining to the state in which a transmission control unit or an audio response unit can accept incoming calls on a line.

enciphered data (ED). Data whose meaning is concealed from unauthorized users or observers.

encryption. In computer security, the process of transforming data into an unintelligible form in such a way that the original data either cannot be obtained or can be obtained only by using a decryption process.

end bracket. In SNA, the value (binary 1) of the end bracket indicator in the request header (RH) of the first request of the last chain of a bracket; the value denotes the end of the bracket. Contrast with *begin bracket*. See also *bracket*.

end node (EN). (1) See *Advanced Peer-to-Peer Networking (APPN) end node* and *low-entry networking (LEN) end node*. (2) In communications, a node that is frequently attached to a single data link and cannot perform intermediate routing functions.

end node domain. An end node control point, its attached links, and its local LUs.

entry point (EP). (1) The address or label of the first instruction executed on entering a computer program, routine, or subroutine. A computer program, routine, or subroutine may have a number of different entry points, each perhaps corresponding to a different function or purpose. (I) (A) (2) In SNA, a type 2.0, type 2.1, type 4, or type 5 node that provides distributed network management support. It sends network management data about itself and the resources it controls to a focal point for centralized processing, and it receives and executes focal-point initiated commands to manage and control its resources.

environment variable. A variable that specifies (a) how an operating system or another program will run or (b) the devices that the operating system will recognize.

ER. Explicit route.

error. A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. (I) (A)

error log. A data set or file in a product or system where error information is stored for later access.

Ethernet. A 10-Mbps baseband local area network that allows multiple stations to access the transmission medium at will without prior coordination, avoids contention by using carrier sense and deference, and resolves contention by using collision detection and delayed retransmission. Ethernet uses carrier sense multiple access with collision detection (CSMA/CD).

event. An occurrence of significance to a task; for example, an SNMP trap, the opening of a window or a submap, or the completion of an asynchronous operation.

exception. An abnormal condition such as an I/O error encountered in processing a data set or a file.

exception request (EXR). In SNA, a request that replaces another message unit in which an error has been detected and that carries sense data identifying the error.

exception response (ER). In SNA, a protocol requested in the form-of-response-requested field of a request header that directs the receiver to return a response only if the request is unacceptable as received or cannot be processed; that is, a negative response, but not a positive response, can be returned. Contrast with *definite response* and *no response*.

execute. To perform the actions specified by a program or a portion of a program. (T)

exit. To execute an instruction within a portion of a computer program in order to terminate the execution of that portion. Such portions of computer programs include loops, subroutines, modules, and so on. (T)

expedited flow. In SNA, a data flow designated in the transmission header (TH) that is used to carry network control, session control, and various data flow control request/response units (RUs); the expedited flow is separate from the normal flow (which carries primarily end-user data) and can be used for commands that affect the normal flow. Contrast with *normal flow*.

Note: The normal and expedited flows move in both the primary-to-secondary and secondary-to-primary directions. Requests and responses on a given flow, whether normal or expedited, usually are processed sequentially within the path, but the expedited flow traffic may be moved ahead of the normal-flow traffic within the path at queuing points in the half-sessions and for half-session support in boundary functions.

explicit route (ER). In SNA, a series of one or more transmission groups that connect two subarea nodes. An explicit route is identified by an origin subarea address, a destination subarea address, an explicit route number, and a reverse explicit route number. Contrast with *virtual route (VR)*.

EXR. Exception request.

EXT. External trace file.

extended binary-coded decimal interchange code (EBCDIC). The standard code, using a character set consisting of 8-bit coded characters, used by Personal Communications for information interchange between personal computers and a host system. See also *American National Standard Code for Information Interchange*.

F

fault. An accidental condition that causes a functional unit to fail to perform its required function. (I) (A)

FDX. Full-duplex.

feature. A part of an IBM product that may be ordered separately by the customer.

FID. Format identification.

field. (1) An area in a record or panel used to contain data. (2) In the IBM 3270 data stream, a group of consecutive positions on a presentation space having similar characteristics that are defined by a field attribute byte at the beginning of the field. (3) An identifiable area in a window. Examples of fields are: an entry field, into which a user can type or place text, and a field of radio button choices, from which a user can select one choice.

file. A named set of records stored or processed as a unit. (T) Synonymous with *data set*.

file specification. In Communications Manager/2, the full identifier for a file, which includes its drive, path, file name, and file extension. The format is dependent on the storage medium of a file; for example, `C:\PATH\FILENAME.EXT`.

file transfer. The transfer of one or more files from one system to another over a data link.

first-in-chain (FIC). A request unit (RU) whose request header (RH) begin chain indicator is on and whose RH end chain indicator is off. See also *RU chain*.

first speaker. See *first-speaker session*.

first-speaker session. The half-session defined at session activation as (a) able to begin a bracket without requesting permission from the other half-session to do so and (b) winning contention if both half-sessions attempt to begin a bracket simultaneously. Synonym for *contention-winner session*. Contrast with *bidder session*.

fixed pacing. Synonym for *fixed session-level pacing*.

fixed session-level pacing. A form of session-level pacing in which the data transfer rate is controlled using fixed pacing-window sizes, which are initialized at session-activation time. Synonymous with *fixed pacing*. Contrast with *adaptive session-level pacing*.

flag. (1) To mark an information item for selection for further processing. (T) (2) A character that signals the occurrence of some condition, such as the end of a word. (A) (3) A character or bit sequence that marks an occurrence or boundary, such as the end of a word or the beginning or end of a data transmission block.

flow. In NetDA/2, the amount of traffic that can pass through a node, connection, or route in both directions during a given period of time.

flow control. (1) In SNA, the process of managing the rate at which data traffic passes between components of the network. The purpose of flow control is to optimize the rate of flow of message units with minimum congestion in the network; that is, to neither overflow the buffers at the receiver or at intermediate routing nodes, nor leave the receiver waiting for more message units. (2) See also *pacing*.

flushing. In logical unit (LU) 6.2, the process of sending through the network all remaining buffered data generated by a transaction program. The transaction program issues the flush verb to begin the process. It also occurs if the network operator issues the command.

FMD. Function management data.

foreground process. (1) In the AIX operating system, a process that must run to completion before another command is issued to the shell. The foreground process is in the foreground process group, which is the group that receives the signals generated by a terminal. (2) Contrast with *background process*.

foreign host. Synonym for *remote host*.

fork. In the AIX operating system, to create and start a child process.

format identification (FID) field. In SNA, a field in each transmission header (TH) that indicates the format of the TH; that is, the presence or absence of certain fields. TH formats differ in accordance with the types of nodes between which they pass. Following are the six FID types:

- FID0, used for traffic involving non-SNA devices between adjacent subarea nodes when either or both nodes do not support explicit route and virtual route protocols
- FID1, used for traffic involving SNA devices between adjacent subarea nodes when either or both nodes do not support explicit route and virtual route protocols

- FID2, used for traffic between a subarea node and an adjacent type 2 peripheral node
- FID3, used for traffic between a subarea node and an adjacent type 1 peripheral node
- FID4, used for traffic between adjacent subarea nodes when both nodes support explicit route and virtual route protocols
- FIDF, used for certain commands (for example, for transmission group control) sent between adjacent subarea nodes when both nodes support explicit route and virtual route protocols.

FQDN. Fully qualified domain name.

frame. (1) In Open Systems Interconnection architecture, a data structure pertaining to a particular area of knowledge and consisting of slots that can accept the values of specific attributes and from which inferences can be drawn by appropriate procedural attachments. (T) (2) The unit of transmission in some local area networks, including the IBM Token-Ring Network. It includes delimiters, control characters, information, and checking characters. (3) In SDLC, the vehicle for every command, every response, and all information that is transmitted using SDLC procedures. (4) A data structure (data frame) composed of fields meeting the field specifications of a type of communication protocol. Frames are used to control data transfer across a data link. (5) In SDLC, a sequence of bits delimited by an opening and closing flag. In X.25 packet switching data networks, frames are composed of 8-bit byte sequences delimited by beginning and ending flags; the frames in X.25 control various functions, data transfer, and transmission checking.

frequency. The rate of signal oscillation, expressed in hertz.

full-duplex (FDX). Synonym for *duplex*.

fully qualified domain name (FQDN). In the Internet suite of protocols, the name of a host system that includes all of the subnames of the domain name. An example of a fully qualified domain name is `ralvm7.vnet.ibm.com`. See also *host name*.

fully qualified name. (1) In SNA, synonym for *network-qualified name*. (2) In the Internet suite of protocols, see *fully qualified domain name (FQDN)*.

function call. An expression that moves the path of execution from the current function to a specified function and evaluates to the return value provided by the called function. A function call contains the name of the function to which control moves and a parenthesized list of values.

function management data (FMD). An RU category used for end-user data exchanged between logical units (LUs) and for requests and responses exchanged between network services components of LUs, PUs, and control points.

G

gateway. (1) A functional unit that interconnects two computer networks with different network architectures. A gateway connects networks or systems of different architectures. A bridge interconnects networks or systems with the same or similar architectures. (T) (2) The combination of machines and programs that provide address translation, name translation, and system services control point (SSCP) rerouting between independent SNA networks to allow those networks to communicate. A gateway consists of one gateway NCP and at least one gateway VTAM. (3) In the IBM Token-Ring Network, a device and its associated software that connect a local area network to another local area network or a host that uses different logical link protocols. (4) In the AIX operating system, an entity that operates above the link layer and translates, when required, the interface and protocol used by one network into those used by another distinct network. (5) In TCP/IP, synonym for *router*. (6) A station in a local area network through which a connection to a host system or a separate network is established.

GDS. General data stream.

general data stream (GDS). The data stream used for conversations in LU 6.2 sessions.

general data stream (GDS) variable. A type of RU substructure that is preceded by an identifier and a length field and includes either application data, user control data, or SNA-defined control data.

generic unbind. Synonym for *session deactivation request*.

GMT. Greenwich mean time.

Greenwich mean time (GMT). The mean solar time at the prime meridian (0 degrees longitude) of Greenwich, England. Greenwich mean time is sometimes called *Z time* or *Zulu time*.

Note: Although *Greenwich mean time (GMT)* and *coordinated universal time (UTC)* are sometimes used interchangeably, they are not synonyms. Greenwich mean time is an approximate time. Because the second is no longer defined in terms of astronomical phenomena, the preferred name for this time scale is *coordinated universal time (UTC)*.

group ID (GID). (1) In RACF, a string of one to eight characters that identifies a group. The first character must be A through Z, #, $, or @. The rest can be A through Z, #, $, @, or 0 through 9. (2) In the AIX operating system, a number that corresponds to a specific group name. The group ID can often be substituted in commands that take a group name as a value.

H

half-duplex (HD, HDX). In data communication, pertaining to transmission in only one direction at a time. Contrast with *duplex*.

half-session. A session-layer component consisting of the combination of data flow control and transmission control components comprising one end of a session. See also *session connector*.

handle. In the Advanced DOS and OS/2 operating systems, a binary value created by the system that identifies a drive, directory, and file so that the file can be found and opened.

HD. Half-duplex.

header. (1) System-defined control information that precedes user data. (2) The portion of a message that contains control information for the message such as one or more destination fields, name of the originating station, input sequence number, character string indicating the type of message, and priority level for the message.

header file. Synonym for *include file*.

Help. A choice that gives a user access to helpful information about objects, choices, tasks, and products. A Help choice can appear on a menu bar or as a push button.

hexadecimal. (1) Pertaining to a selection, choice, or condition that has 16 possible different values or states. (I) (2) Pertaining to a fixed-radix numeration system, with radix of 16. (I) (3) Pertaining to a system of numbers to the base 16; hexadecimal digits range from 0 through 9 and A through F, where A represents 10 and F represents 15.

highlighting. Emphasizing a display element or segment by modifying its visual attributes. (I) (A)

host. (1) In the Internet suite of protocols, an end system. The end system can be any workstation; it does not have to be a mainframe. (2) See *host processor*.

host LU. In SNA, a logical unit in a host processor; for example, a VTAM application program.

host name. In the Internet suite of protocols, the name given to a machine. Sometimes, “host name” is used to mean *fully qualified domain name (FQDN)*; other times, it is used to mean the most specific subname of a fully qualified domain name. For example, if `ralvm7.vnet.ibm.com` is the fully qualified domain name, either of the following may be considered the host name:

- `ralvm7.vnet.ibm.com`
- `ralvm7`

host processor. (1) A processor that controls all or part of a user application network. (T) (2) In a network, the processing unit in which the data communication access method resides.

host session. A logical connection that enables a personal computer to communicate with a host system. A session can be identified by LU address, LT number, or session ID. See *DOS session*. See also *logical terminal*.

host system. In Personal Communications, the computer linked to one or more personal computers by the SDLC, LAN, ASYNCH, X.25, or DFT attachment.

I

IBM Disk Operating System (DOS). A disk operating system based on MS-DOS that operates with all IBM personal computers.

IBM Operating System/2 (OS/2). An IBM licensed program that can be used as the operating system for personal computers. The OS/2 licensed program can perform multiple tasks at the same time.

IBM Token-Ring Network. A general-purpose baseband token-ring local area network for interconnecting information processing equipment at a local site. It uses a token-ring access protocol and supports a data rate of either 4 or 16 million bits per second. It conforms to the IEEE 802.5 (token-ring) and IEEE 802.2 (logical link control) standards.

icon. A graphical representation of an object, consisting of an image, image background, and a label.

ID. (1) Identifier. (2) Identification.

IDLC. Integrated data link control.

inactive. (1) Not operational. (2) Pertaining to a node or device not connected or not available for connection to another node or device. (3) Contrast with *active*.

inbound. In communications, data that is received from the network.

include file. A text file that contains declarations used by a group of functions, programs, or users. Synonymous with *header file*.

indirect activation. In VTAM, the activation of a lower-level resource of the resource hierarchy as a result of SCOPE or ISTATUS specifications related to an activation command naming a higher-level resource. Contrast with *direct activation*.

information (I) format. A format used for information transfer.

information (I) frame. A frame in I format used for numbered information transfer.

INITIATE. A network services request sent from a logical unit (LU) to a system services control point (SSCP) requesting that an LU-LU session be established.

input/output (I/O). (1) Pertaining to input, output, or both. (A) (2) Pertaining to a device, process, or channel involved in data input, data output, or both.

instance. In the AIX operating system, a concrete realization of an abstract object class. An instance of a widget or a gadget is a specific data structure that contains detailed appearance and behavioral information that is used to generate a specific graphical object on-screen at run time.

INT. Internal trace table.

integrated data link control (IDLC). A full-duplex high-level data link control (HDLC) protocol. It is IBM’s implementation of the CCITT Q.922 standard or link access procedure extended (LAPE). IDLC can support point-to-point workstation connections over a full-duplex wide area network (WAN).

interactive. Pertaining to the exchange of information between a user and a computer.

interface. (1) A shared boundary between two functional units, defined by functional characteristics, signal characteristics, or other characteristics, as appropriate. The concept includes the specification of the connection of two devices having different functions. (T) (2) Hardware, software, or both, that links systems, programs, or devices.

intermediate session routing (ISR). A type of routing function within an APPN network node that provides session-level flow control and outage reporting for all sessions that pass through the node but whose end points are elsewhere.

I/O. Input/output.

IPR. Isolated pacing response.

J

JISCII. Japanese Industry Standard Code for Information Interchange. A code used by IBM Personal Computers in Japan. It contains Japanese Industry Standard (JIS) C 6226 “Kanji code for exchanging information” (JIS Kanji set) and user unique characters.

L

LAN. Local area network.

layer. (1) In network architecture, a group of services that is complete from a conceptual point of view, that is one out of a set of hierarchically arranged groups, and that extends across all systems that conform to the network architecture. (T) (2) In the Open Systems Interconnection reference model, one of seven conceptually complete, hierarchically arranged groups of services, functions, and protocols, that extend across all open systems. (T) (3) In SNA, a grouping of related functions that are logically separate from the functions in other groups. Implementation of the functions in one layer can be changed without affecting functions in other layers.

LEN. Low-entry networking.

line. (1) The portion of a data circuit external to data circuit-terminating equipment (DCE), that connects the DCE to a data switching exchange (DSE), that connects a DCE to one or more other DCEs, or that connects a DSE to another DSE. (I) (2) Synonymous with *channel* and *circuit*.

line control discipline. Synonym for *link protocol* and *protocol*.

line discipline. Synonym for *link protocol* and *protocol*.

link. (1) The combination of the link connection (the transmission medium) and two link stations, one at each end of the link connection. A link connection can be shared among multiple links in a multipoint or token-ring configuration. (2) To interconnect items of data or portions of one or more computer programs: for example, the linking of object programs by a linkage editor, linking of data items by pointers. (T) (3) In SNA, synonymous with *data link*.

link-attached. (1) Pertaining to devices that are connected to a controlling unit by a data link. (2) Contrast with *channel-attached*. (3) Synonymous with *remote*.

link connection. (1) The physical equipment providing two-way communication between one link station and one or more other link stations; for example, a telecommunication line and data circuit-terminating equipment (DCE). (2) In SNA, synonymous with *data circuit*.

link connection segment. A portion of the configuration that is located between two resources listed consecutively in the service point command service (SPCS) query link configuration request list.

link protocol. (1) The rules for sending and receiving data at the link level. (2) Synonymous with *line control discipline* and *line discipline*.

link status (LS). Information maintained by local and remote modems.

load. (1) To bring all or part of a computer program into memory from auxiliary storage so that the computer can run the program. (2) To place a diskette into a diskette drive.

local. (1) Pertaining to a device accessed directly without use of a telecommunication line. (2) Contrast with *remote*. (3) Synonym for *channel-attached*.

local area network (LAN). (1) A computer network located on a user's premises within a limited geographical area. Communication within a local area network is not subject to external regulations; however, communication across the LAN boundary may be subject to some form of regulation. (T) (2) A network in which a set of devices are connected to one another for communication and that can be connected to a larger network. (3) See also *Ethernet* and *token ring*. (4) Contrast with *metropolitan area network (MAN)* and *wide area network (WAN)*.

local directory database. That set of resources (LUs) in the network known at a particular node. The resources included are all those in the node's domain as well as any cache entries.

local LU. A logical unit not distributed over the LAN, but controlled by a gateway personal computer. This is normally a physical device such as a workstation, printer, or terminal.

local session identification (LSID). In SNA, a field in an FID3 (format identification type 3) transmission header that contains an indication of the type of session (SSCP-PU, SSCP-LU, or LU-LU) and the local address of the peripheral logical unit (LU) or physical unit (PU).

logged-on operator. (1) A NetView operator station task that requires a terminal and a logged-on user. (2) Contrast with *autotask*.

logical link. (1) A pair of link stations, one in each of two adjacent nodes, and their underlying link connection, providing a single link-layer connection between the two nodes. Multiple logical links can be distinguished while they share the use of the same physical media connecting two nodes. Examples are 802.2 logical links used on local area network (LAN) facilities and LAP E logical links on the same point-to-point physical link between two nodes. The term logical link also includes the multiple X.25 logical channels that share the use of the access link from a DTE to an X.25 network. (2) In APPNTAM, the unidirectional representation in a node of a link.

logical record. A set of related data or words considered to be a record from a logical viewpoint. (T)

logical terminal. (1) A destination with a name that is related to one or more physical terminals. (2) The definition of a specific 3270 or 5250 emulation session.

logical unit (LU). A type of network accessible unit that enables users to gain access to network resources and communicate with each other.

logical unit of work (LUW). The changes to protected resources that are committed or backed out as a unit as a result of a sync point operation. The protected resources may be distributed among different LUs joined together by conversations.

logic error. In VTAM, an error condition that results from an invalid request (a program logic error).

logon. (1) The procedure by which a user begins a terminal session. (2) The process of accessing a domain and assuming a user ID.

logon mode. In VTAM, a subset of session parameters specified in a logon mode table for communication with a logical unit. See also *session parameters*.

low-entry networking (LEN). A capability of nodes to attach directly to one another using basic peer-to-peer protocols to support multiple and parallel sessions between logical units.

low-entry networking (LEN) end node. A LEN node receiving network services from an adjacent APPN network node.

low-entry networking (LEN) node. A node that provides a range of end-user services, attaches directly to other nodes using peer protocols, and derives network services implicitly from an adjacent APPN network node, that is, without the direct use of CP-CP sessions.

LS. Link status.

LSID. Local session identification.

LU. Logical unit.

LU address. In Personal Communications, a number from 02 to 254 assigned by the host to represent the remote partner in an SNA session.

LU-LU session. A logical connection between two logical units (LUs) in an SNA network that typically provides communication between two users.

LUS. Logical unit services.

LU type. The classification of an LU in terms of the specific subset of SNA protocols and options it supports for a given session, namely:

- The mandatory and optional values allowed in the session activation request
- The usage of data stream controls, function management headers (FMHs), request unit parameters, and sense data values

- Presentation services protocols such as those associated with FMH usage

LU types 0, 1, 2, 3, 4, 6.1, 6.2, and 7 are defined.

LUW. Logical unit of work.

LU 2. A type of LU for an application program that communicates with a single display workstation in an interactive environment, using the SNA 3270 data stream.

LU 3. A type of LU for an application program that communicates with a single printer, using the SNA 3270 data stream.

LU 6.2. (1) A type of logical unit that supports general communication between programs in a distributed processing environment. LU 6.2 is characterized by (a) a peer relationship between session partners, (b) efficient utilization of a session for multiple transactions, (c) comprehensive end-to-end error processing, and (d) a generic application programming interface (API) consisting of structured verbs that are mapped into a product implementation. (2) A type of LU that supports sessions between two applications in a distributed data processing environment using the SNA general data stream, which is a structured-field data stream, or a user-defined data stream.

LU 6.2 verb. A syntactical unit in the LU 6.2 application programming interface representing an operation.

M

macroinstruction. (1) An instruction in a source language that is to be replaced by a defined sequence of instructions in the same source language and that may also specify values for parameters in the replaced instructions. (T) (2) In assembler programming, an assembler language statement that causes the assembler to process a predefined set of statements called a macro definition. The statements normally produced from the macro definition replace the macroinstruction in the program. (3) See also *definition statement*.

maintenance services. In SNA, one of the types of network services in system services control points (SSCPs) and physical units (PUs). Maintenance services provide facilities for testing links and nodes and for collecting and recording error information.

MAN. Metropolitan area network.

management services (MS). (1) One of the types of network services in control points (CPs) and physical units (PUs). Management services are the services provided to assist in the management of SNA networks, such as problem management, performance and accounting management, configuration management, and change management. (2) Services that assist in the management of systems and networks in areas such as problem management, performance management, business management, operations management, configuration management, and change management.

manager. (1) In systems management, a user that, for a particular interaction, has assumed a manager role. (2) An entity that monitors or controls one or more managed objects by (a) receiving notifications regarding the objects and (b) requesting management operations to modify or query the objects. (3) A system that assumes a manager role.

mapped conversation. An LU 6.2 conversation type specified by the allocating transaction program. Transaction programs using a mapped conversation can exchange messages of arbitrary format regardless of the underlying data stream. System-defined or user-defined mappers can perform data transformation for the transaction programs. See also *conversation*. Contrast with *basic conversation*.

mapping. The process of converting data that is transmitted in one format by the sender into the data format that can be accepted by the receiver.

matching. In MPTN architecture, pertaining to the relationship between peer transport users or peer transport providers that use the same user protocols or the same transport protocols.

MB. Megabyte.

megabyte (MB). (1) For processor storage, real and virtual storage, and channel volume, 220 or 1 048 576 bytes. (2) For disk storage capacity and communications volume, 1 000 000 bytes.

memory. All of the addressable storage space in a processing unit and other internal storages that is used to execute instructions. (T)

menu. (1) A list of options displayed to the user by a data processing system, from which the user can select an action to be initiated. (T) (2) In text processing, a list of choices displayed to the user by a text processor from which the user can select an action to be initiated. (T) (3) A list of choices that can be applied to an object. A menu can contain choices that are not available for selection in certain contexts. Those choices are indicated by reduced contrast.

message. (1) An assembly of characters and sometimes control codes that is transferred as an entity from an originator to one or more recipients. A message consists of two parts: envelope and content. (T) (2) A communication sent from a person or program to another person or program.

message processing facility (MPF). An MVS facility that controls message display and message processing.

message queue. In IMS/VS, the data set on which messages are queued before being processed by an application program or sent to a terminal.

metric. In Internet communications, a value, associated with a route, which is used to discriminate between multiple exit or entry points to the same autonomous system. The route with the lowest metric is preferred.

metropolitan area network (MAN). A network formed by the interconnection of two or more networks which may operate at higher speed than those networks, may cross administrative boundaries, and may use multiple access methods. (T) Contrast with *local area network (LAN)* and *wide area network (WAN)*.

MID. Machine identifier.

migration. The installation of a new version or release of a program to replace an earlier version or release.

mode. See *mode name*.

modem (modulator/demodulator). (1) A functional unit that modulates and demodulates signals. One of the functions of a modem is to enable digital data to be transmitted over analog transmission facilities. (T) (A) (2) A device that converts digital data from a computer to an analog signal that can be transmitted on a telecommunication line, and converts the analog signal received to data for the computer.

mode name. The name used by the initiator of a session to designate the characteristics desired for the session, such as traffic pacing values, message-length limits, sync point and cryptography options, and the class of service within the transport network.

module. A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to or output from an assembler, compiler, linkage editor, or executive routine. (A)

monitor. (1) A device that observes and records selected activities within a data processing system for analysis. Possible uses are to indicate significant departure from the norm, or to determine levels of utilization of particular functional units. (T) (2) Software or hardware that observes, supervises, controls, or verifies operations of a system. (A) (3) The function required to initiate the transmission of a token on the ring and to provide soft-error recovery in case of lost tokens, circulating frames, or other difficulties. The capability is present in all ring stations. (4) In the NetView Graphic Monitor Facility, to open a view that can receive status changes from the NetView program. Problem determination and correction can be performed directly from the view. Contrast with *browse*.

mouse. A commonly used pointing device, containing one or more buttons, with which a user can interact with a product or the operating environment.

MPF. Message processing facility.

MS. Management services.

MSG. Console messages.

N

NAU. (1) Network accessible unit. (2) Network addressable unit.

NC. Network control.

NCP/EP definition facility (NDF). A program that is part of System Support Programs (SSP) and that is used to generate a load module for a partitioned emulation program (PEP), a Network Control Program (NCP), or an Emulation Program (EP).

NDF. NCP/EP definition facility.

negative response (NR). In SNA, a response indicating that a request did not arrive successfully or was not processed successfully by the receiver. Contrast with *positive response*.

negotiable BIND. In SNA, a capability that allows two half-sessions to negotiate the parameters of a session when the session is being activated.

negotiation. The process of deciding what packet size to transmit between a network and a 3710 Network Controller.

NETID. See *network identifier*.

NetView-NetView task (NNT). The task under which a cross-domain NetView operator session runs. See *operator station task*.

network. (1) An arrangement of nodes and connecting branches. (T) (2) A configuration of data processing devices and software connected for information interchange. (3) A group of nodes and the links interconnecting them.

network accessible unit (NAU). A logical unit (LU), physical unit (PU), control point (CP), or system services control point (SSCP). It is the origin or the destination of information transmitted by the path control network. Synonymous with *network addressable unit*.

network address. (1) According to ISO 7498-3, a name, unambiguous within the OSI environment, that identifies a set of network service access points. (2) An address, consisting of subarea and element fields, that identifies a link, a link station, or a network addressable unit. Subarea nodes use network addresses; peripheral nodes use local addresses. (3) In SNA, an address consisting of subarea and element fields that identifies a link, gateway, or network addressable unit (NAU). (4) In a subarea network, an address, consisting of subarea and element fields, that identifies a link, link station, physical unit, logical unit, or system services control point. Subarea nodes use network addresses; peripheral nodes use local addresses or local-form session identifiers (LFSIDs). The boundary function in the subarea node to which a peripheral node is attached transforms local addresses or LFSIDs to network addresses and vice versa. Contrast with *network name*.

network addressable unit (NAU). Synonym for *network accessible unit*.

network architecture. The logical structure and operating principles of a computer network. (T)

Note: The operating principles of a network include those of services, functions, and protocols.

network congestion. An undesirable overload condition caused by traffic in excess of what a network can handle.

network control (NC). In SNA, a request/response unit (RU) category used for requests and responses exchanged between physical units (PUs) for such purposes as activating and deactivating explicit and virtual routes and sending load modules to adjust peripheral nodes. See also *data flow control*, *function management data*, and *session control*.

network directory database. Synonym for *distributed directory database*.

network identifier. (1) In TCP/IP, that part of the IP address that defines a network. The length of the network ID depends on the type of network class (A, B, or C). (2) A 1- to 8-byte customer-selected name or an 8-byte IBM-registered name that uniquely identifies a specific subnetwork. (3) In MPTN architecture, the address qualifier of a transport provider address that identifies a group of nodes according to the network in which they reside.

network name. The symbolic identifier by which users refer to a network accessible unit, a link, or a link station within a given subnetwork. In APPN networks, network names are also used for routing purposes. Contrast with *network address*.

network node domain. An APPN network-node control point, its attached links, the network resources for which it answers directory search requests (namely, its local LUs and adjacent LEN end nodes), the adjacent APPN end nodes with which it exchanges directory search requests and replies, and other resources (such as a local storage device) associated with its own node or an adjacent end node for which it provides management services.

network operator. (1) A person who controls the operation of all or part of a network. (2) In a multiple-domain network, a person or program responsible for controlling all domains. (3) See also *domain operator*.

network-qualified name. In SNA, a name that uniquely identifies a specific resource (such as an LU or a CP) within a specific network. It consists of a network identifier and a resource name, each of which is a 1- to 8-byte symbol string. Synonymous with *fully qualified name*.

network services. (1) The services within network accessible units that control network operation through SSCP-SSCP, SSCP-PU, SSCP-LU, and CP-CP sessions. (2) The session services (directory and route-selection functions) and management services provided by an APPN network-node control point to its domain.

node. (1) In a network, a point at which one or more functional units connect channels or data circuits. (I) (2) Any device, attached to a network, that transmits and receives data. (3) An endpoint of a link or a junction common to two or more links in a network. Nodes can be processors, communication controllers, cluster controllers, or terminals. Nodes can vary in routing and other functional capabilities.

node name. In VTAM, the symbolic name assigned to a specific major or minor node during network definition.

nonblocking mode. (1) A way of requesting a service over an interface so that if the request cannot be completed immediately, the requesting process is able to continue and is not suspended. (2) Contrast with *blocking mode*.

no response. In SNA, a protocol requested in the form-of-response-requested field of the request header that directs the receiver of the request not to return any response, regardless of whether or not the request is received and processed successfully. Contrast with *definite response* and *exception response*.

normal flow. In SNA, a data flow designated in the transmission header (TH) that is used primarily to carry end-user data. The rate at which requests flow on the normal flow can be regulated by session-level pacing. Normal and expedited flows move in both the primary-to-secondary and secondary-to-primary directions. Contrast with *expedited flow*.

notification. An unscheduled, spontaneously generated report of an event that has occurred.

NOTIFY. A network services request that is sent by a system services control point (SSCP) to a logical unit (LU) to inform the LU of the status of a procedure requested by the LU.

O

OAF. Origin address field.

OAF'. Origin address field prime.

object. (1) In object-oriented design or programming, an abstraction consisting of data and the operations associated with that data. See also *class*. (2) An item that a user can manipulate as a single unit to perform a task. An object can appear as text, an icon, or both.

ODAI. Origin-Destination Assignor indicator, a bit in a FID2 transmission header used to divide the address space so that an address space manager (ASM) in one node may use all possible combinations of OAF′, DAF′ with the ODAI having one setting and the ASM in the adjacent node may use all possible combinations of OAF′, DAF′ with the ODAI having the complementary setting.

Off. A choice that appears in the cascaded menu from the Refresh choice. It sets the refresh function to off.

offline. (1) Pertaining to the operation of a functional unit that takes place either independently of, or in parallel with, the main operation of a computer. (T) (2) Neither controlled by, nor communicating with, a computer. Contrast with *online*.

offset. The number of measuring units from an arbitrary starting point in a record, area, or control block, to some other point.

OIA. Operator information area.

OK. A push button that accepts the information in a window and closes it. If the window contains changed information, those changes are applied before the window is closed.

On. A choice that appears in a cascaded menu from the Refresh choice. It immediately refreshes the view in a window.

one-way bracket. A bracket in which data is sent from one NAU to another in a single chain with begin bracket, conditional end bracket, and exception response requested. When one-way brackets are used on CP-CP sessions, they are always sent on the contention-winner session.

online. (1) Pertaining to the operation of a functional unit when under the direct control of the computer. (T) (2) Pertaining to a user's ability to interact with a computer. (A) (3) Pertaining to a user's access to a computer via a terminal. (A) (4) Controlled by, or communicating with, a computer. (5) Contrast with *offline*.

open. (1) A break in an electrical circuit. (2) To make an adapter ready for use.

Open. A choice that leads to a window in which users can select the object they want to open.

operating system (OS). Software that controls the execution of programs and that may provide services such as resource allocation, scheduling, input/output control, and data management. Although operating systems are predominantly software, partial hardware implementations are possible. (T)

operation. In object-oriented design or programming, a service that can be requested at the boundary of an object. Operations include modifying an object or disclosing information about an object.

operator. (1) In a language statement, the lexical entity that indicates the action to be performed on operands. See also *definition statement*. (2) A person or program responsible for managing activities controlled by a given piece of software such as MVS, the NetView program, or IMS. (3) A person who operates a device. (4) A person who keeps a system running. (5) See also *autotask*, *logged-on operator*, *network operator*, and *operator station task*.

operator information area (OIA). The area near the bottom of the display area where terminal or system status information is displayed.

operator station task (OST). The NetView task that establishes and maintains the online session with the network operator. There is one operator station task for each network operator who logs on to the NetView program. See *NetView-NetView task*.

option set. (1) A set of functions that may be supported by products that implement a particular architecture. A product may support any number of option sets or none. For each option set supported, all functions in that set are supported. (2) Contrast with *base set*.

origin. An external logical unit (LU) or application program from which a message or other data originates. See also *destination*.

origin address field (OAF). In SNA, a field in a FID0 or FID1 transmission header that contains the address of the originating network accessible unit (NAU). Contrast with *destination address field (DAF)*. See also *format identification (FID) field* and *local session identification (LSID)*.

OS. Operating system.

outbound. In communications, data that is transmitted to the network.

output. Pertaining to a device, process, or channel involved in an output process, or to the associated data or states. The word “output” may be used in place of “output data,” “output signal”, “output process”, when such a usage is clear in a given context. (T)

overlay. A collection of predefined data, such as lines, shading, text, boxes, or logos, that can be merged with variable data on a page while printing.

P

pacing. (1) A technique by which a receiving component controls the rate of transmission of a sending component to prevent overrun or congestion. (2) See *receive pacing*, *send pacing*, *session-level pacing*, and *virtual route (VR) pacing*. (3) See also *flow control*.

pacing group. Synonym for *pacing window*.

pacing response. In SNA, an indicator that signifies the readiness of a receiving component to accept another pacing group. The indicator is carried in a response header (RH) for session-level pacing and in a transmission header (TH) for virtual route pacing.

pacing window. (1) The path information units (PIUs) that can be transmitted on a virtual route before a virtual-route pacing response is received, indicating that the virtual route receiver is ready to receive more PIUs on the route. (2) The requests that can be transmitted on the normal flow in one direction on a session before a session-level pacing response is received, indicating that the receiver is ready to accept the next group of requests. (3) Synonymous with *pacing group*.

packet. (1) In data communication, a sequence of binary digits, including data and control signals, that is transmitted and switched as a composite whole. The data, control signals, and, possibly, error control information are arranged in a specific format. (I) (2) A discrete unit of information including addressing and sequence information that can be sent interleaved with other packets of the same and other transmission sequences. Data sent by packet switching protocols, such as the X.25 protocol, do not always arrive in the order sent or even by the same routing on a PSDN.

packet assembler/disassembler (PAD). (1) A functional unit that enables data terminal equipment (DTEs) not equipped for packet switching to access a packet switched network. (T) (A) (2) A program that codes data into packets for transmission and reassembles received packets into the original data format. See also *packet*, *packet-switching data network*.

packet switching. The process of routing and transferring data by means of addressed packets so that a channel is occupied only during transmission of a packet. On completion of the transmission, the channel is made available for transfer of other packets. (I)

packet switching data network (PSDN). (1) A network that uses packet switching as a means of transmitting data. (2) A data transmission system consisting of one or more host systems and one or more network stations that communicates using packets of information. The data is sent in interleaved packets to distribute the flow of information and maximize the data transmission capabilities of the network. X.25 is a packet-switching data network communication protocol. See *X.25 network*.

PAD. Packet assembler/disassembler.

page. (1) In a virtual storage system, a fixed-length block that has a virtual address and is transferred as a unit between real storage and auxiliary storage. (I) (A) (2) The information displayed at the same time on the screen of a display device. (3) To replace the information displayed on the screen with prior or subsequent information from the same file.

parallel. (1) Pertaining to a process in which all events occur within the same interval of time, each handled by a separate but similar functional unit; for example, the parallel transmission of the bits of a computer word along the lines of an internal bus. (T) (2) Pertaining to concurrent or simultaneous operation of two or more devices or to concurrent performance of two or more activities in a single device. (A) (3) Pertaining to concurrent or simultaneous occurrence of two or more related activities in multiple devices or channels. (A) (4) Pertaining to the simultaneity of two or more processes. (A) (5) Pertaining to the simultaneous processing of the individual parts of a whole, such as the bits of a character and the characters of a word, using separate facilities for the various parts. (A) (6) Contrast with *serial*.

parallel sessions. Two or more concurrently active sessions between the same two network accessible units (NAUs) using different pairs of network addresses or local-form session identifiers. Each session can have independent session parameters.

parameter. (1) A variable that is given a constant value for a specified application and that may denote the application. (I) (A) (2) In Basic CUA architecture, a variable used in conjunction with a command to affect its result. (3) An item in a menu for which the user specifies a value or for which the system provides a value when the menu is interpreted. (4) Data passed to a program or procedure by a user or another program, namely as an operand in a language statement, as an item in a menu, or as a shared data structure.

parent. A process that has spawned a child process using the fork primitive. Contrast with *child*.

parent process. In the AIX and OS/2 operating systems, a process that creates other processes. Contrast with *child process*.

password. (1) A value used in authentication or a value used to establish membership in a set of people having specific privileges. (2) A unique string of characters known to a computer system and to a user, who must specify the character string to gain access to a system and to the information stored within it. (3) In computer security, a string of characters known to the computer system and a user, who must specify it to gain full or limited access to a system and to the data stored within it.

path. (1) In a network, any route between any two nodes. A path may include more than one branch. (T) (2) The series of transport network components (path control and data link control) that are traversed by the information exchanged between two network accessible units. See also *explicit route (ER)*, *route extension (REX)*, and *virtual route (VR)*.

path control (PC). The function that routes message units between network accessible units in the network and provides the paths between them. It converts the basic information units (BIUs) from transmission control (possibly segmenting them) into path information units (PIUs) and exchanges basic transmission units containing one or more PIUs with data link control. Path control differs by node type: some nodes (APPN nodes, for example) use locally generated session identifiers for routing, and others (subarea nodes) use network addresses for routing.

path information unit (PIU). A message unit consisting of a transmission header (TH) alone, or a TH followed by a basic information unit (BIU) or a BIU segment.

PC. (1) Personal computer. (2) Path control. (3) Personal Communications.

peer. In network architecture, any functional unit that is in the same layer as another entity. (T)

performance error. Synonym for *temporary error*.

peripheral LU. In SNA, a logical unit in a peripheral node. Contrast with *subarea LU*.

peripheral node. A node that uses local addresses for routing and therefore is not affected by changes in network addresses. A peripheral node requires boundary-function assistance from an adjacent subarea node. A peripheral node can be a type 1, 2.0, or 2.1 node connected to a subarea boundary node.

peripheral PU. In SNA, a physical unit in a peripheral node. Contrast with *subarea PU*.

permanent error. An error that cannot be resolved by error recovery programs. Contrast with *temporary error*.

permanent virtual circuit (PVC). (1) In X.25 and frame-relay communications, a virtual circuit that has a logical channel permanently assigned to it at each data terminal equipment (DTE). Call-establishment protocols are not required. Contrast with *switched virtual circuit (SVC)*. (2) The logical connection between two frame-relay terminating equipment stations, either directly or through one or more frame-relay frame handlers. A PVC consists of one or more PVC segments. (3) A session assigned to a dedicated line telephonic link.

persistent verification. In VTAM, a security function that enables two logical units to verify the identity of each other for the initial conversation on a session and to assume that future conversations are verified for the duration of the session.

Personal Communications product family. A group of IBM licensed programs that emulate 3270 and 5250 terminals and that run on several operating systems such as OS/2, DOS, and Windows.

personal computer (PC). (1) A microcomputer primarily intended for stand-alone use by an individual. (T) (2) A desktop, floor-standing, or portable microcomputer that usually consists of a system unit, a display monitor, a keyboard, one or more diskette drives, internal fixed-disk storage, and an optional printer. PCs are designed primarily for stand-alone operation but may be connected to mainframes or networks.

physical circuit. A circuit established without multiplexing. See also *data circuit*. Contrast with *virtual circuit*.

physical unit (PU). (1) The component that manages and monitors the resources (such as attached links and adjacent link stations) associated with a node, as requested by an SSCP via an SSCP-PU session. An SSCP activates a session with the physical unit in order to indirectly manage, through the PU, resources of the node such as attached links. This term applies to type 2.0, type 4, and type 5 nodes only. (2) See also *peripheral PU* and *subarea PU*.

physical unit (PU) services. In SNA, the components within a physical unit (PU) that provide configuration services and maintenance services for SSCP-PU sessions.

PIP. Program initialization parameters.

PLU. Primary logical unit.

pointer. (1) A data element that indicates the location of another data element. (T) (2) An identifier that indicates the location of an item of data. (A)

pop. To remove an item from the top of a pushdown list. Contrast with *push*.

POP. Post Office Protocol.

positive response. In SNA, a response indicating that a request was received and processed. Contrast with *negative response*.

post. (1) To enter a unit of information on a record. (2) To note the occurrence of an event. (3) To add information to a record in order to keep the record current.

Post Office Protocol (POP). A protocol used for exchanging network mail and accessing mailboxes.

Prepare. A presentation services header that flows as part of commit processing, indicating the partner has begun the first phase of the two-phase commit process.

presentation space ID (PSID). In Communications Manager/2, synonym for *short name*.

primary half-session. In SNA, the half-session that sends the session activation request. See also *primary logical unit (PLU)*. Contrast with *secondary half-session*.

primary logical unit (PLU). In SNA, the logical unit (LU) that sends the BIND to activate a session with its partner LU. Contrast with *secondary logical unit (SLU)*.

problem determination. The process of determining the source of a problem; for example, a program component, machine failure, telecommunication facilities, user or contractor-installed programs or equipment, environmental failure such as a power loss, or user error.

procedure. (1) In a programming language, a block, with or without formal parameters, whose execution is invoked by means of a procedure call. (I) (2) The description of the course of action taken for the solution of a problem. (A)

process. (1) To perform operations on data in a process. (I) (A) (2) In data processing, the course of events that occurs during the execution of all or part of a program. (T) (3) A course of the events defined by its purpose or by its effect, achieved under given conditions. (4) Any operation or combination of operations on data. (5) A function being performed or waiting to be performed.

process identification number (process ID). A unique number assigned to a process by the operating system. The number is used internally by processes to communicate.

processor. In a computer, a functional unit that interprets and executes instructions. A processor consists of at least an instruction control unit and an arithmetic and logic unit. (T)

product-set identification (PSID). (1) In SNA, a technique for identifying the hardware and software products that implement a network component. (2) A management services common subvector that transports the information described in definition (1).

profile. Data that describes the significant characteristics of a user, a group of users, or one or more computer resources.

program. (1) A sequence of instructions suitable for processing by a computer. Processing may include the use of an assembler, a compiler, an interpreter, or a translator to prepare the program for execution, as well as to execute it. (I) (2) In programming languages, a logical assembly of one or more interrelated modules. (I) (3) To design, write, and test computer programs. (I) (A)

program initialization parameters (PIP). The initial parameter values passed to a target program as input or used to set up the process environment.

protocol. (1) A set of semantic and syntactic rules that determine the behavior of functional units in achieving communication. (I) (2) In Open Systems Interconnection architecture, a set of semantic and syntactic rules that determine the behavior of entities in the same layer in performing communication functions. (T) (3) In SNA, the meanings of, and the sequencing rules for, requests and responses used for managing the network, transferring data, and synchronizing the states of network components. Synonymous with *line control discipline* and *line discipline*. See *bracket protocol* and *link protocol*.

PSID. Presentation space ID.

PU. Physical unit.

push. To add an item to the top of a pushdown list. Contrast with *pop*.

pushdown list. (1) A list constructed and maintained so that the next data element to be retrieved is the most recently stored. (T) (2) Synonymous with *stack*.

PUT. Program update tape.

PVC. Permanent virtual circuit.

Q

queue. (1) A list constructed and maintained so that the next data element to be retrieved is the one stored first. (T) (2) A line or list of items waiting to be processed; for example, work to be performed or messages to be displayed. (3) To arrange in or form a queue.

quiesce. To end a process by allowing operations to complete normally.

quiesce protocol. In VTAM, a method of communicating in one direction at a time. Either the primary logical unit (PLU) or the secondary logical unit (SLU) assumes the exclusive right to send normal-flow requests, and the other node does not send such requests. When the sender wants to receive, it releases the other node from its quiesced state.

R

RAM. Random access memory. (A)

RAM semaphore. In the OS/2 operating system, a semaphore used to serialize different threads of a single process. Contrast with *system semaphore*.

random access memory (RAM). A storage device into which data is entered and from which data is retrieved in a nonsequential manner.

receive pacing. In SNA, the pacing of message units that a component is receiving. Contrast with *send pacing*.

record. A set of data treated as a unit. (T)

reentrant. The attribute of a program or routine that allows the same copy of the program or routine to be used concurrently by two or more tasks.

Refresh. A cascading choice that gives a user access to other choices (*On* and *Off*) that control whether changes made to underlying data in a window are displayed immediately, not displayed at all, or displayed at a later time.

release. (1) A distribution of a new product or new function and APAR fixes for an existing product. Normally, programming support for the prior release is discontinued after some specified period of time following availability of a new release. The first version of a product is announced as Release 1, Modification Level 0. (2) In VTAM, to relinquish control of resources (communication controllers or physical units). See also *resource takeover*. Contrast with *acquire*.

remote. (1) Pertaining to a system, program, or device that is accessed through a telecommunication line. (2) Synonym for *link-attached*. (3) Contrast with *local*.

remote host. Any host on a network except the host at which a particular operator is working. Synonymous with *foreign host*.

request. A message unit that signals initiation of a particular action or protocol. For example, Initiate-Self is a request for activation of an LU-LU session.

requester. A computer that accesses shared network resources through a server. Synonym for *client*.

request header (RH). The control information that precedes a request unit (RU). See also *request/response header (RH)*.

request/response header (RH). Control information associated with a particular RU. The RH precedes the request/response unit (RU) and specifies the type of RU (request unit or response unit).

request/response unit (RU). A generic term for a request unit or a response unit. See *request unit (RU)* and *response unit (RU)*.

request to send (RTS). In data communication, a signal raised by data terminal equipment (DTE), while the data terminal is ready, to request facilities from data circuit-terminating equipment (DCE) so that data can be sent. Contrast with *clear to send (CTS)*.

request unit (RU). A message unit that contains control information, end-user data, or both.

reset. On a virtual circuit, reinitialization of data flow control. At reset, all data in transit are eliminated.

resource. Any facility of a computing system or operating system required by a job or task, and including main storage, input/output devices, the processing unit, data sets, and control or processing programs.

resource takeover. In VTAM, an action initiated by a network operator to transfer control of resources from one domain to another without breaking the connections or disrupting existing LU-LU sessions on the connection. See also *acquire* and *release*.

resource types. In the NetView program, a concept to describe the organization of panels. Resource types are defined as central processing unit, channel, control unit, and I/O device for one category; and communication controller, adapter, link, cluster controller, and terminal for another category. Resource types are combined with data types and display types to describe display organization. See also *data types* and *display types*.

response. (1) In data communication, a reply represented in the control field of a response frame. It advises the primary or combined station of the action taken by the secondary or other combined station to one or more commands. (2) See also *command*.

response header (RH). (1) A header, optionally followed by a response unit (RU), that indicates whether the response is positive or negative and that may contain a pacing response. (2) See also *negative response*, *pacing response*, and *positive response*.

response indicator. In the AS/400 system, a 1-character field passed with an input record from the system to a program to provide information about the data record or actions taken by the workstation user.

response unit (RU). A message unit that acknowledges a request unit. It may contain prefix information received in a request unit. If positive, the response unit may contain additional information (such as session parameters in response to BIND SESSION).

If negative, the response unit contains sense data defining the exception condition.

resync. Recovery processing that is performed by sync point services when a failure of a session, transaction program, or LU occurs during sync point processing. The purpose of resync is to return protected resources to consistent states.

return code. (1) A code used to influence the execution of succeeding instructions. (A) (2) A value returned to a program to indicate the results of an operation requested by that program.

REX. Route extension.

RH. Request/response header.

ring. See *ring network*.

ring network. A network configuration in which devices are connected by unidirectional transmission links to form a closed path.

route. (1) An ordered sequence of nodes and transmission groups (TGs) that represent a path from an origin node to a destination node traversed by the traffic exchanged between them. (2) The path that network traffic uses to get from source to destination.

routed. Pronounced "route-d." See *route daemon*.

route daemon. A program that runs under 4BSD UNIX to propagate route information among machines on a local area network. Also referred to as *routed* (pronounced "route-d").

route extension (REX). In SNA, the path control network components, including a peripheral link, that make up the portion of a path between a subarea node and a network addressable unit (NAU) in an adjacent peripheral node. See also *explicit route (ER)*, *path*, and *virtual route (VR)*.

router. (1) A computer that determines the path of network traffic flow. The path selection is made from several paths based on information obtained from specific protocols, algorithms that attempt to identify the shortest or best path, and other criteria such as metrics or protocol-specific destination addresses. (2) An attaching device that connects two LAN segments, which use similar or different architectures, at the reference model network layer. (3) In OSI terminology, a function that determines a path by which an entity can be reached. (4) In TCP/IP, synonymous with *gateway*. (5) Contrast with *bridge*.

routine. A program, or part of a program, that may have some general or frequent use. (T)

routing. (1) The process of determining the path to be used for transmission of a message over a network. (T) (2) The assignment of the path by which a message is to reach its destination. (3) In SNA, the forwarding of a message unit along a particular path through a network, as determined by parameters carried in the message unit, such as the destination network address in a transmission header.

RQD. Request discontact.

RTS. Request to send.

RU. Request/response unit.

RU chain. In SNA, a set of related request/response units (RUs) that are consecutively transmitted on a particular normal or expedited data flow. The request RU chain is the unit of recovery: if one of the RUs in the chain cannot be processed, the entire chain is discarded. Each RU belongs to only one chain, which has a beginning and an end indicated by means of control bits in request/response headers within the RU chain. Each RU can be designated as first-in-chain (FIC), last-in-chain (LIC), middle-in-chain (MIC), or only-in-chain (OIC). Response units and expedited-flow request units are always sent as only-in-chain.

S

SBCS. Single-byte character set.

SC. Session control.

SDLC. Synchronous Data Link Control.

SDT. Start data traffic.

secondary half-session. In SNA, the half-session that receives the session-activation request. See also *secondary logical unit (SLU)*. Contrast with *primary half-session*.

secondary logical unit (SLU). In SNA, the logical unit (LU) that contains the secondary half-session for a particular LU-LU session. An LU may contain secondary and primary half-sessions for different active LU-LU sessions. Contrast with *primary logical unit (PLU)*.

secondary logical unit (SLU) key. A key-encrypting key used to protect a session cryptography key during its transmission to the secondary half-session.

segment. (1) A section of cable between components or devices. A segment may consist of a single patch cable, several patch cables that are connected, or a combination of building cable and patch cables that are connected. (2) In Internet communications, the unit of transfer between TCP functions in different machines. Each segment contains control and data fields; the current byte-stream position and actual data bytes are identified along with a checksum to validate received data. (3) Synonym for *BIU segment*. (4) See also *link connection segment*.

segmentation. A process by which path control (PC) divides basic information units (BIUs) into smaller units, called BIU segments, to accommodate smaller buffer sizes in adjacent nodes. Both segmentation and segment assembly are optional PC features. The support for either or both is indicated in the BIND request and response.

segmenting. In OSI, a function performed by a layer to map one protocol data unit (PDU) from the layer it supports into multiple PDUs.

select. To explicitly identify one or more objects to which a subsequent choice will apply.

selection. The process of explicitly identifying one or more objects to which a subsequent choice will apply.

semaphore. An indicator used to control access to a file; for example, in a multiuser application, a flag that prevents simultaneous access to a file.

send pacing. In SNA, the pacing of message units that a component is sending. Contrast with *receive pacing*.

sequence number. (1) In communications, a number assigned to a particular frame or packet to control the transmission flow and receipt of data. (2) A numerical value assigned by VTAM to each message exchanged between two nodes. The value (one for messages sent from the application program to the logical unit and another for messages sent from the logical unit to the application program) increases by one for each successive message transmitted unless it is reset by the application program with a set and test sequence numbers (STSN) indicator.

serial. (1) Pertaining to a process in which all events occur one after the other; for example, serial transmission of the bits of a character according to V24 CCITT protocol. (T) (2) Pertaining to the sequential or consecutive occurrence of two or more related activities in a single device or channel. (A) (3) Pertaining to the sequential processing of the individual parts of a whole, such as the bits of a character or the characters of a word, using the same facilities for successive parts. (A) (4) Contrast with *parallel*.

server. (1) A functional unit that provides shared services to workstations over a network; for example, a file server, a print server, a mail server. (T) (2) In a network, a data station that provides facilities to other stations; for example, a file server, a print server, a mail server. (A) (3) In the AIX operating system, an application program that usually runs in the background and is controlled by the system program controller. (4) In the Enhanced X-Windows Toolkit, a program that provides the basic windowing mechanism. It handles interprocess communication (IPC) connections from clients, demultiplexes graphics requests onto screens, and multiplexes input back to clients.

service point (SP). An entry point that supports applications that provide network management for resources not under the direct control of itself as an entry point. Each resource is either under the direct control of another entry point or not under the direct control of any entry point. A service point accessing these resources is not required to use SNA sessions (unlike a focal point). A service point is needed when entry point support is not yet available for some network management function.

session. (1) In network architecture, for the purpose of data communication between functional units, all the activities which take place during the establishment, maintenance, and release of the connection. (T) (2) A logical connection between two network accessible units (NAUs) that can be activated, tailored to provide various protocols, and deactivated, as requested. Each session is uniquely identified in a transmission header (TH) accompanying any transmissions exchanged during the session. (3) A logical connection between a server and a requester that was initiated by a successful request for a shared resource. See also *host session* and *DOS session*.

session activation request. In SNA, a request that activates a session between two network accessible units (NAUs) and specifies session parameters that control various protocols during session activity; for example, BIND and ACTPU. Contrast with *session deactivation request*.

session connector. A session-layer component in an APPN network node or in a subarea node boundary or gateway function that connects two stages of a session. Session connectors swap addresses from one address space to another for session-level intermediate routing, segment session message units as needed, and (except for gateway function session connectors) adaptively pace the session traffic in each direction. See also *half-session*.

session control (SC). In SNA, either of the following:

- One of the components of transmission control. Session control is used to purge data flowing in a session after an unrecoverable error occurs, to resynchronize the data flow after such an error, and to perform cryptographic verification.
- A request unit (RU) category used for requests and responses exchanged between the session control components of a session and for session activation and deactivation requests and responses.

session data. Data about a session, collected by the NetView program, that consists of session awareness data, session trace data, and session response time data.

session deactivation request. In SNA, a request that deactivates a session between two network accessible units (NAUs); for example, UNBIND and DACTPU. Synonymous with *generic unbind*. Contrast with *session activation request*.

session ID. An alphabetic ID (*a* through *h*) assigned by Personal Communications to each session or screen. This applies to all types of host sessions and is used in file transfers. See also *short name*.

session-level LU-LU verification. An LU 6.2 security service that is used to verify the identity of each logical unit when a session is established.

session-level pacing. (1) A flow control technique that permits a receiving half-session or session connector to control the data transfer rate (the rate at which it receives request units) on the normal flow. It is used to prevent overloading a receiver with unprocessed requests when the sender can generate requests faster than the receiver can process them. (2) See also *adaptive session-level pacing*, *fixed session-level pacing*, and *virtual route (VR) pacing*.

session limit. The maximum number of concurrently active LU-LU sessions that a particular logical unit (LU) can support.

session parameters. In SNA, the parameters that specify or constrain the protocols (such as bracket protocol and pacing) for a session between two network accessible units. See also *logon mode*.

session services (SS). One of the types of network services in the control point (CP) and in the logical unit (LU). These services provide facilities for an LU or a network operator to request that a control point (an ENCP, NNCP, or SSCP) assist with initiating or terminating sessions between logical units. Assistance with session termination is needed only by SSCP-dependent LUs. See *configuration services*, *maintenance services*, and *management services*.

shared. Pertaining to the availability of a resource for more than one use at the same time.

shell. (1) A software interface between a user and the operating system of a computer. Shell programs interpret commands and user interactions on devices such as keyboards, pointing devices, and touch-sensitive screens and communicate them to the operating system. Shells simplify user interactions by eliminating the user's concern with operating system requirements. A computer may have several layers of shells for various levels of user interaction. (2) In the AIX operating system, a command interpreter that acts as an interface between the user and the operating system. A shell can contain another shell nested inside it; the outer shell is the parent shell, and the inner shell is the child.

shift-in character (SI). A code extension character used to terminate a sequence that has been introduced by the shift-out character to make effective the graphic characters of the standard character set. (I) (A)

shift-out character (SO). A code extension character that substitutes for the graphic characters of the standard character set an alternative set of graphic characters upon which an agreement has been arrived at or that has been designated using code extension procedures. (I) (A)

short name. (1) In Personal Communications, a character displayed in column 7 of the operator information area that shows the session ID. See also *session ID* and *operator information area*. (2) In Communications Manager/2, the one-letter name (A through Z) of the presentation space or emulation session. Synonymous with *presentation space ID (PSID)* and *short-session ID*.

short-session ID. In Communications Manager/2, synonym for *short name*.

shutdown. The process of ending operation of a system or a subsystem, following a defined procedure.

SI. The shift-in character. (I) (A)

single-byte character set (SBCS). A character set in which each character is represented by a one-byte code. Contrast with *double-byte character set (DBCS)*.

SLU. Secondary logical unit.

SNA. Systems Network Architecture.

SNA network. The part of a user-application network that conforms to the formats and protocols of Systems Network Architecture. It enables reliable transfer of data among users and provides protocols for controlling the resources of various network configurations. The SNA network consists of network accessible units (NAUs), boundary function, gateway function, and intermediate session routing function components; and the transport network.

SNAP. Subnetwork Access Protocol.

SO. The shift-out character. (I) (A)

SP. Service point.

SSCP. System services control point.

SSCP-dependent LU. An LU that requires assistance from a system services control point (SSCP) in order to initiate an LU-LU session. It requires an SSCP-LU session.

SSCP-LU session. In SNA, a session between a system services control point (SSCP) and a logical unit (LU). The session enables the LU to request the SSCP to help initiate LU-LU sessions.

SSCP-PU session. In SNA, a session between a system services control point (SSCP) and a physical unit (PU); SSCP-PU sessions allow SSCPs to send requests to and receive status information from individual nodes in order to control the network configuration.

stack. Synonym for *pushdown list*.

stage. A program that processes messages in a NetView pipeline. Stages send messages to each other serially.

statement. A language syntactic unit consisting of an operator, or other statement identifier, followed by one or more operands. See *definition statement*.

static. (1) In programming languages, pertaining to properties that can be established before execution of a program; for example, the length of a fixed length variable is static. (I) (2) Pertaining to an operation that occurs at a predetermined or fixed time. (3) Contrast with *dynamic*.

status. The condition or state of hardware or software, usually represented by a status code.

storage. (1) A functional unit into which data can be placed, in which they can be retained and from which they can be retrieved. (T) (2) The action of placing data into a storage device. (I) (A) (3) A storage device. (A)

stream. (1) To send data from one device to another. (2) See *data stream*.

STSN. Set and test sequence numbers.

stub. (1) A program module that transfers remote procedure calls (RPCs) and responses between a client and a server. Stubs perform marshalling, unmarshalling, and data format conversion. Both clients and servers have stubs. The Network Interface Definition Language (NIDL) compiler generates client and server stub code from an interface definition. (2) Hooking functions used as extensions to the protocol to generate protocol requests for the Enhanced X-Windows Toolkit. (3) A small module, link-edited into application code, that locates and transfers control to a larger body of related code.

subarea. A portion of the SNA network consisting of a subarea node, attached peripheral nodes, and associated resources. Within a subarea node, all network accessible units (NAUs), links, and adjacent link stations (in attached peripheral or subarea nodes) that are addressable within the subarea share a common subarea address and have distinct element addresses.

subarea LU. In SNA, a logical unit in a subarea node. Contrast with *peripheral LU*.

subarea network. Interconnected subareas, their directly attached peripheral nodes, and the transmission groups that connect them.

subarea node (SN). A node that uses network addresses for routing and maintains routing tables that reflect the configuration of the network. Subarea nodes can provide gateway function to connect multiple subarea networks, intermediate routing function, and boundary function support for peripheral nodes. Type 4 and type 5 nodes can be subarea nodes.

subarea PU. In SNA, a physical unit in a subarea node. Contrast with *peripheral PU*.

subdirectory. A directory contained within another directory in a file system hierarchy.

Subnetwork Access Protocol (SNAP). In LANs, a 5-byte protocol discriminator that identifies the non-IEEE standard protocol family to which a packet belongs. The SNAP value is used to differentiate between protocols that use `$AA` as their service access point (SAP) value.

subsystem. A secondary or subordinate system, usually capable of operating independently of, or asynchronously with, a controlling system. (T)

subsystem management. In Communications Manager/2, a group of advanced functions for diagnosis and problem resolution. These functions are typically used by system administrators and application programmers.

SVC. Switched virtual circuit.

switched virtual circuit (SVC). An X.25 circuit that is dynamically established when needed. The X.25 equivalent of a switched line. Contrast with *permanent virtual circuit (PVC)*.

synchronization point. Synonym for *sync point*.

synchronous. (1) Pertaining to two or more processes that depend upon the occurrence of specific events such as common timing signals. (T) (2) Occurring with a regular or predictable time relationship.

Synchronous Data Link Control (SDLC). A discipline conforming to subsets of the Advanced Data Communication Control Procedures (ADCCP) of the American National Standards Institute (ANSI) and High-level Data Link Control (HDLC) of the International Organization for Standardization, for managing synchronous, code-transparent, serial-by-bit information transfer over a link connection. Transmission exchanges may be duplex or half-duplex over switched or nonswitched links. The configuration of the link connection may be point-to-point, multipoint, or loop. (I)

synchronous operation. In VTAM, a communication, or other operation in which VTAM, after receiving the request for the operation, does not return control to the program until the operation is completed. Contrast with *asynchronous operation*.

synchronous request. In VTAM, a request for a synchronous operation. Contrast with *asynchronous request*.

sync point. An intermediate or end point during processing of a transaction at which an update or modification to one or more of the transaction's protected resources is logically complete and error free. Synonymous with *synchronization point*.

system. In data processing, a collection of people, machines, and methods organized to accomplish a set of specific functions. (I) (A)

system semaphore. In the OS/2 operating system, a signal mechanism providing control between threads of multiple processes that can be used by processes that do not share memory. Contrast with *RAM semaphore*.

system services control point (SSCP). A component within a subarea network for managing the configuration, coordinating network operator and problem determination requests, and providing directory services and other session services for users of the network. Multiple SSCPs, cooperating as peers with one another, can divide the network into domains of control, with each SSCP having a hierarchical control relationship to the physical units and logical units within its own domain.

system services control point (SSCP) domain. The system services control point, the physical units (PUs), the logical units (LUs), the links, the link stations, and all the resources that the SSCP has the ability to control by means of activation and deactivation requests.

Systems Network Architecture (SNA). The description of the logical structure, formats, protocols, and operational sequences for transmitting information units through, and controlling the configuration and operation of, networks. The layered structure of SNA allows the ultimate origins and destinations of information, that is, the users, to be independent of and unaffected by the specific SNA network services and facilities used for information exchange.

T

table. A repository for data that NetDA/2 uses to design a network. Each table contains information related to the network.

task. In a multiprogramming or multiprocessing environment, one or more sequences of instructions treated by a control program as an element of work to be accomplished by a computer. (I) (A)

Task-Related User Exit (TRUE). A CICS module used for invoking resource managers that are outside of CICS. A TRUE is one of the components of CICS sockets.

TC. Transmission control.

temporary error. A resource failure that can be resolved by error recovery programs. Synonymous with *performance error*. Contrast with *permanent error*.

TERMINATE. In SNA, a request unit that is sent by a logical unit (LU) to its system services control point (SSCP) to cause the SSCP to start a procedure to end one or more designated LU-LU sessions.

TH. Transmission header.

thread. In the OS/2 operating system, the smallest unit of operation to be performed within a process.

timeout. (1) An event that occurs at the end of a predetermined period of time that began at the occurrence of another specified event. (I) (2) A time interval allotted for certain operations to occur; for example, response to polling or addressing before system operation is interrupted and must be restarted.

TLPB. Transport-layer protocol boundary.

token. (1) In a local area network, the symbol of authority passed successively from one data station to another to indicate the station temporarily in control of the transmission medium. Each data station has an opportunity to acquire and use the token to control the medium. A token is a particular message or bit pattern that signifies permission to transmit. (T) (2) In LANs, a sequence of bits passed from one device to another along the transmission medium. When the token has data appended to it, it becomes a frame.

token ring. (1) According to IEEE 802.5, network technology that controls media access by passing a token (special packet or frame) between media-attached stations. (2) A FDDI or IEEE 802.5 network with a ring topology that passes tokens from one attaching ring station (node) to another. (3) See also *local area network (LAN)*.

token-ring network. (1) A ring network that allows unidirectional data transmission between data stations, by a token passing procedure, such that the transmitted data return to the transmitting station. (T) (2) A network that uses a ring topology, in which tokens are passed in a circuit from node to node. A node that is ready to send can capture the token and insert data for transmission.

TP. Transaction program.

transaction program (TP). A program that processes transactions in an SNA network. There are two kinds of transaction programs: application transaction programs and service transaction programs. See also *conversation*.

translation table. A table used to replace one or more characters with alternative characters; for example, to translate characters representing a virtual address to those representing a real address, characters representing an event to those representing a procedure call, characters of a national character set to those of another national language, or characters representing a relocated address to those representing an absolute address.

transmission control (TC) layer. The layer within a half-session or session connector that synchronizes and paces session-level data traffic, checks session sequence numbers of requests, and enciphers and deciphers end-user data. See also *half-session*.

transmission header (TH). Control information, optionally followed by a basic information unit (BIU) or a BIU segment, that is created and used by path control to route message units and to control their flow within the network. See also *path information unit*.

transport-layer protocol boundary (TLPB). In MPTN architecture, the protocol boundary that provides access in a protocol-independent fashion to multiple transport protocols.

trap. In the Simple Network Management Protocol (SNMP), a message sent by a managed node (agent function) to a management station to report an exception condition.

TRUE. Task-Related User Exit.

tutorial. Information presented in a teaching format.

U

UNBIND. In SNA, a request to deactivate a session between two logical units (LUs). See also *session deactivation request*. Contrast with *BIND*.

uninterpreted name. In SNA, a character string that a system services control point (SSCP) can convert into the network name of a logical unit (LU). Typically, an uninterpreted name is used in a logon or Initiate request from a secondary logical unit (SLU) to identify the primary logical unit (PLU) with which the session is requested.

UPM. User Profile Management.

user. (1) Any person or any thing that may issue or receive commands and messages to or from the information processing system. (T) (2) Anyone who requires the services of a computing system.

user identifier (UID). A name that uniquely identifies a user on a network or system.

user profile. In computer security, a description of a user that includes such information as user ID, user name, password, access authority, and other attributes obtained at logon.

User Profile Management (UPM). In the OS/2 operating system, a function that provides user ID validation and user and group management facilities. Communications Manager/2 uses UPM. Each installation of UPM is local to the particular workstation on which it is installed and validates user access to controlled data or programs that reside on that particular workstation. UPM also provides the logon-logoff mechanism for identifying and authenticating system users.

UTC. Coordinated universal time.

V

validation. The checking of data for correctness or for compliance with applicable standards, rules, and conventions. (A)

value. (1) A specific occurrence of an attribute; for example, “blue” for the attribute “color.” (T) (2) A quantity assigned to a constant, a variable, a parameter, or a symbol.

variable. (1) In programming languages, a language object that may take different values, one at a time. The values of a variable are usually restricted to a certain data type. (I) (2) A quantity that can assume any of a given set of values. (A) (3) A name used to represent a data item whose value can be changed while the program is running.

verb. See *LU 6.2 verb*.

version. A separately licensed program that usually has significant new code or new functions.

virtual circuit. (1) In packet switching, the facilities provided by a network that give the appearance to the user of an actual connection. (T) See also *data circuit*. Contrast with *physical circuit*. (2) A logical connection established between two DTEs.

virtual route (VR). (1) In SNA, either (a) a logical connection between two subarea nodes that is physically realized as a particular explicit route or (b) a logical connection that is contained wholly within a subarea node for intranode sessions. A virtual route between distinct subarea nodes imposes a transmission priority on the underlying explicit route, provides flow control through virtual route pacing, and provides data integrity through sequence numbering of path information units (PIUs). (2) Contrast with *explicit route (ER)*. See also *path* and *route extension (REX)*.

virtual route (VR) pacing. In SNA, a flow control technique used by the virtual route control component of path control at each end of a virtual route to control the rate at which path information units (PIUs) flow over the virtual route. VR pacing can be adjusted according to traffic congestion in any of the nodes along the route. See also *session-level pacing*.

Virtual Telecommunications Access Method (VTAM). An IBM licensed program that controls communication and the flow of data in an SNA network. It provides single-domain, multiple-domain, and interconnected network capability.

VR. Virtual route.

VTAM. (1) Virtual Telecommunications Access Method. (2) Synonymous with *ACF/VTAM*.

W

WAN. Wide area network.

wide area network (WAN). (1) A network that provides communication services to a geographic area larger than that served by a local area network or a metropolitan area network, and that may use or provide public communication facilities. (T) (2) A data communication network designed to serve an area of hundreds or thousands of miles; for example, public and private packet-switching networks, and national telephone networks. (3) Contrast with *local area network (LAN)* and *metropolitan area network (MAN)*.

window. (1) A portion of a display surface in which display images pertaining to a particular application can be presented. Different applications can be displayed simultaneously in different windows. (A) (2) An area with visible boundaries that presents a view of an object or with which a user conducts a dialog with a computer system. (3) In data communication, the number of data packets a data terminal equipment (DTE) or data circuit-terminating equipment (DCE) can send across a logical channel before waiting for authorization to send another data packet. The window is the main mechanism of pacing, or flow control, of packets. (4) See *pacing window*.

WinSock application programming interface (API). A socket-style transport interface developed for the Windows family of operating systems.

workstation. (1) A functional unit at which a user works. A workstation often has some processing capability. (T) (2) One or more programmable or nonprogrammable devices that allow a user to do work. (3) A terminal or microcomputer, usually one that is connected to a mainframe or to a network, at which a user can perform applications.

X

X.25. (1) An International Telegraph and Telephone Consultative Committee (CCITT) recommendation for the interface between data terminal equipment and packet-switched data networks. (2) See also *packet switching*.

X.25 network. A packet-switching data network linked in accordance with ITU-TS recommendation X.25.

Z

Z time. Abbreviation for *Zulu time*. Synonym for *coordinated universal time (UTC)*.

Zulu time (Z). Synonym for *coordinated universal time (UTC)*.

Index

© Copyright IBM Corp. 1994, 1997

Readers' Comments — We'd Like to Hear from You

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name	Address
Company or Organization	
Phone No.	

Readers' Comments — We'd Like to Hear from You
SC31-8425-01

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

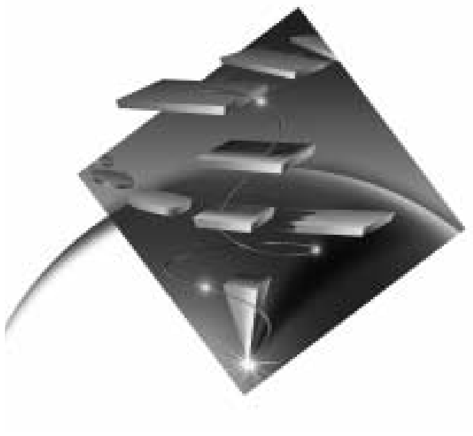
POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Information Development
Department CGMD / Bldg 500
P.O. Box 12195
Research Triangle Park, NC
27709-9990

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.



SC31-8425-01