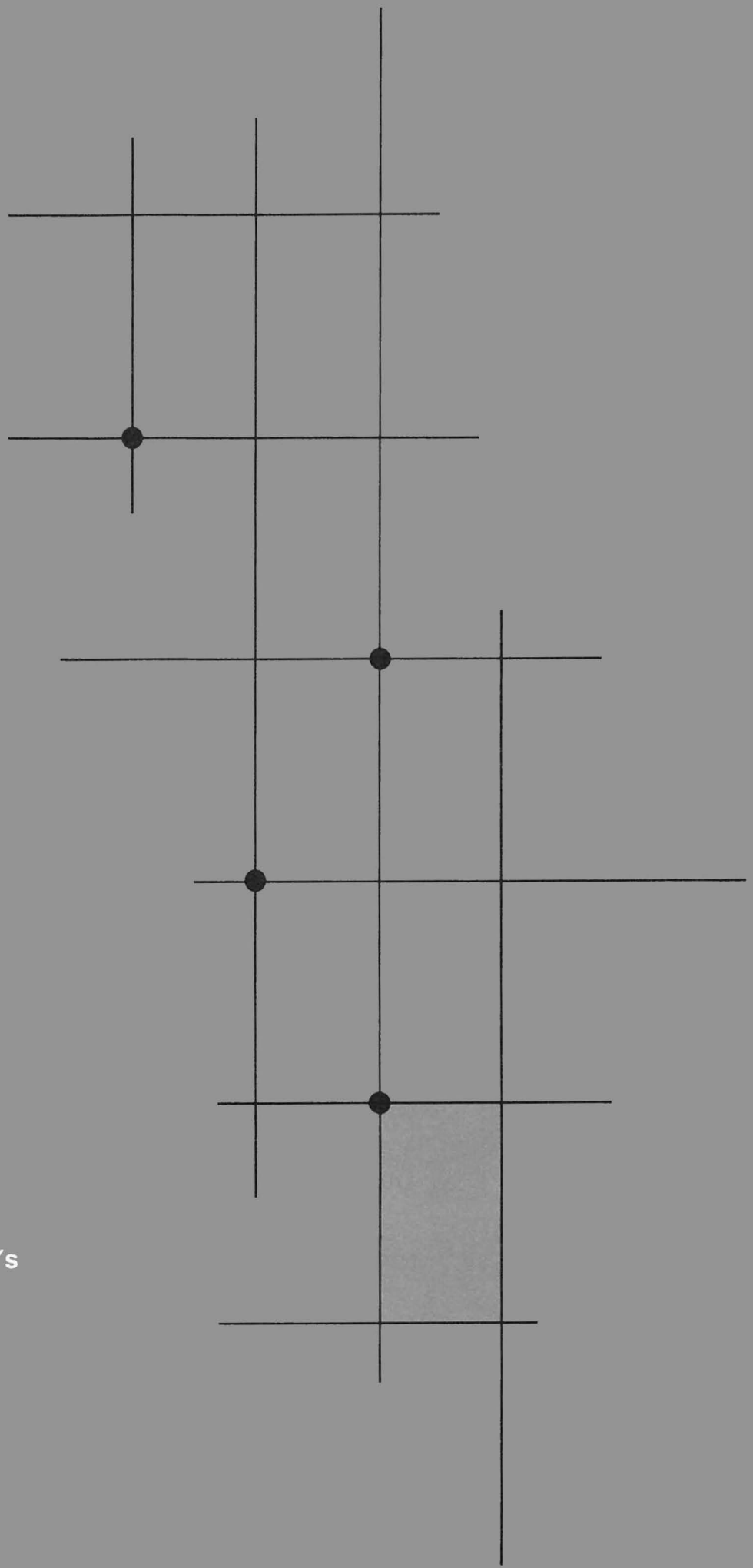
Systems
Network
Architecture



Transaction Programmer's Reference Manual For LU Type 6.2

IBM

GC30-3084-2

Systems Network Architecture

Transaction Programmer's Reference Manual For LU Type 6.2

GC30-3084-2
File No. 370/4300/8100-30

Third Edition (November 1985)

This edition, GC30-3084-2, is a major revision of the previous edition, GC30-3084-1, and obsoletes that edition.

Changes are periodically made to the information in IBM systems publications. Before using this publication in connection with the operation of IBM systems, consult your IBM representative to find out which editions are applicable and current. For a summary of the changes in this book, see page v.

Any reference to an IBM program product in this document is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

It is possible that this material may contain references to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such products, programming, or services in your country.

This book is not intended for production use and is furnished as is. IBM assumes no responsibility for the use of the functions as described in this book in any production manner.

Publications are not stocked at the address below; requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Information Development, Department E02, P.O. Box 12195, Research Triangle Park, North Carolina 27709, U.S.A. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

PREFACE

This book presents detailed information on the functions that Systems Network Architecture (SNA) logical unit type 6.2 (LU 6.2) provides to system and application programs. This book is written for individuals that design system or application programs for use on an implementation of LU 6.2. The information in this book applies to all IBM products that implement LU 6.2, not to any specific IBM product.[1] This book should be used with the applicable product publications for the IBM products that implement LU 6.2.

LU 6.2 provides for interprogram communication between two or more programs, such that:

- The programs can be distributed among multiple SNA nodes within an SNA network.
- The SNA products that make up the network can be different from one another.
- The programs can be designed independently of where in the network they are located and of the SNA products on which they are run.

This book describes the functions that allow programs to communicate with each other independent of the underlying SNA network configuration.

The processing of transactions typically involves several programs distributed over a network communicating with each other. When used in conjunction with applicable IBM product publications, this book is especially useful to those who design transactions and the programs that process the transactions.

This book assumes that the reader is familiar with the SNA concepts presented in Systems Network Architecture Concepts and Products, GC30-3072. The related publications, listed at the end of the preface, are also helpful in understanding the material in this book.

The material in the first three chapters of this book is organized so that one may read the material straight through. Successive sections in these chapters build on the material presented in preceding sections. The material in the remaining chapters is organized for ease of reference. These chapters contain the detailed descriptions of the functions, called verbs, used to invoke LU 6.2 services.

CHAPTERS

The chapters of this book are:

"Chapter 1. Introduction" provides an introduction to LU type 6.2 and its services.

"Chapter 2. LU 6.2 Protocol Boundary" presents a general description of the LU 6.2 protocol boundary.

"Chapter 3. Transaction Program Verbs" gives an overview of the functions available to the programmer for interacting with resources.

---

[1] This book provides a general description of LU 6.2 functions. Implementation of some of the functions is optional. Optional functions may not be available on all IBM products that implement LU 6.2. All IBM products implementing a particular LU 6.2 function provide that function as described in this book; however, the programming interface that a product provides to invoke that function may differ in syntax from the syntax represented in this book.

"Chapter 4. Conversation Verbs" contains a detailed description of the conversation verbs.

"Chapter 5. Control-Operator Verbs" contains a detailed description of the control operator verbs.

APPENDIXES

The appendixes to this book are:

"Appendix A. Base and Option Sets for Product Support" gives a breakdown of the product-support requirements for implementing the verbs.

"Appendix B. Examples Using Basic Conversation Verbs" provides examples of the use of some of the basic conversation verbs. These are examples only; they represent no specific application.

"Appendix C. Symbol String Conventions" defines the symbol strings referred to throughout the manual.

"Appendix D. List of SNA Service Transaction Programs" contains a list of SNA service transaction programs.

"Appendix E. Conversation State Matrices" provides matrix representations of the state transitions and state-check conditions that occur at the conversation protocol boundary for programs using the basic and type-independent conversation verbs.

PREREQUISITE PUBLICATION

Systems Network Architecture Concepts and Products, GC30-3072

RELATED PUBLICATIONS

Systems Network Architecture Reference Summary, GA27-3136

Systems Network Architecture Technical Overview, GC30-3073

Systems Network Architecture Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2, SC30-3269.

Systems Network Architecture Format and Protocol Reference Manual: Distribution Services, SC30-3098.

Systems Network Architecture Format and Protocol Reference Manual: Architectural Logic, SC30-3112.

Systems Network Architecture—Sessions Between Logical Units, GC20-1868.

Document Interchange Architecture: Technical Reference, SC23-0781.

Document Interchange Architecture: Interchange Document Profile Reference, SC23-0764.

Document Interchange Architecture: Transaction Programmer's Guide, SC23-0763.

SUMMARY OF AMENDMENTS

This edition includes the following new functions, technical changes, and editorial changes:

New Functions:

- Session-level LU-LU verification has been added to LU 6.2 security.
- The SECURITY_USER_ID and SECURITY_PROFILE parameters have been added to the MC_GET_ATTRIBUTES and GET_ATTRIBUTES verbs.
- The MC_RECEIVE_IMMEDIATE and MC_TEST mapped conversation verbs have added.
- The RECEIVE_IMMEDIATE and TEST basic conversation verbs have been added.
- The CONFIRM argument has been added to the TYPE parameter of the MC_DEALLOCATE, MC_PREPARE_TO_RECEIVE, DEALLOCATE, and PREPARE_TO_RECEIVE verbs.
- The DATA_TRUNCATED and FMH_DATA_TRUNCATED indications have been added to the WHAT_RECEIVED parameter of the MC_RECEIVE_AND_WAIT and MC_RECEIVE_IMMEDIATE verbs.
- The FORCE parameter has been added to the RESET_SESSION_LIMIT verb.

Technical Changes:

- Conversation-level security has been enhanced for LU 6.2 security.
- The definition of the base and option sets for product support of the verbs, parameter, return codes, and what-received indications has been changed.
- The character set used for SNA-defined transaction program names has been changed.
- The list of return codes for the MC_TEST, TEST, and WAIT verbs has been expanded.
- The way in which the (LU,mode) session limit for single-session connections is specified on the INITIALIZE_SESSION_LIMIT verb has been changed.
- The DEFINE and DISPLAY verbs have been expanded into a set of four DEFINE verbs and four DISPLAY verbs, and a DELETE verb has been added.

Editorial Changes:

- The material in this book has been reorganized:
  - The verb syntax diagrams have been modified to express certain parameters as syntactically optional.
  - The mapped and basic conversation verbs have been combined into a single chapter, and the verbs that apply to both conversation types have been separated into their own section of the chapter.
  - The descriptions of the conversation states for all of the conversation verbs have been consolidated into one section of the chapter.

  - The descriptions of the return codes for all of the conversation verbs have been consolidated into one section of the chapter.
  - The base- and option-set definition for product support of all the verbs—conversation verbs and control operator verbs—has been consolidated into an appendix.

- A clarification has been added to the list of product option sets to differentiate between those for which only local support is needed for their use, and those for which both local and remote support is needed.
- A list is added showing the SNA-defined transaction program names assigned for use by LU 6.2 products.
- A chart is added showing the hexadecimal codes for character sets A and AE.
- An appendix has been added containing a matrix representation of the state transitions that occur at the conversation protocol boundary for programs using the basic and type-independent conversation verbs.
- Other less significant editorial improvements have been made.

All these additions and changes, excluding the reorganization of material, are indicated in the left margin with a vertical line.

CONTENTS

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

This chapter introduces the reader to general concepts used throughout the book.

SYSTEMS NETWORK ARCHITECTURE

Systems Network Architecture (SNA) is the description of the logical structure, formats, protocols, and operational sequences for transmitting information units through networks and for controlling the configuration and operation of networks. A formal description of SNA is provided in SNA Format and Protocol Reference Manual: Architectural Logic. The description of SNA in this book is limited to the services that SNA logical-unit type 6.2 (LU 6.2) provides to transaction programs. A formal description of LU 6.2 is provided in SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2.

LOGICAL UNIT TYPE 6.2

In SNA, the physical network consists of actual processors, called nodes, and data links between the nodes. The logical network consists of logical processors, called logical units (LUs),[1] and logical connections, called sessions. One or more sessions connect one LU to another LU. Information is transmitted from one LU to another LU over a session.

LU 6.2 is a particular type of SNA logical unit. LU 6.2 provides a connection, or port, between its transaction programs and network resources. Each LU 6.2 makes a set of resources available to its transaction programs. The exact set is product- and configuration-dependent; examples are processor machine cycles and main storage, files on magnetic disk or tape, input/output devices such as keyboard and display terminals, and logical resources such as sessions, queues, and data-base records. Some of these resources are local to a program, that is, attached to the same LU as the program. Other resources are remote, that is, attached to other LUs (remote is defined in terms of the logical configuration of the network; the LUs can be within the same physical node).

Resource allocation and control is a central function of LU 6.2. Programs can request the LU for access to a resource. The LU schedules allocation to serially-reusable resources, creating new copies of logical resources, such as sessions, when necessary. The LU provides resource control in order to ensure integrity of the program's access to the resource. For example, the LU maintains a state[2] representation of the resource, allowing the program to perform an operation on the resource only when the resource is in the appropriate state for that operation. The LU may also provide other resource-related services to its programs, such as resource synchronization-point processing that synchronizes committed changes to resources.

TRANSACTION PROGRAM

Transaction programs process transactions. A transaction is a type of application. It usually involves a specific set of input data and triggers the execution of a specific process or job. One example is

---

[1] Other logical processors, such as physical units (PUs) and system services control points (SSCPs), also exist and are described in SNA Concepts and Products.

[2] A specific operating condition of the resource as it appears to the program at a particular time of access. Over time, the resource changes from one state to another in accord with the program's operations on the resource.

the entry of a customer's deposit and the updating of the customer's balance. A second example is the process of recording item sales, arriving at the amount to be paid by or to a customer, verifying checks before accepting them as tender, and receiving payment for the merchandise. A third example is the transfer of a message to one or more destination points.

A transaction program, as the term is used in this publication, is a program that is executed by or within LU 6.2 and performs services related to the processing of a transaction. For example, the program may be an application program that processes a transaction or is one of several programs that make up a transaction processing application, or it may be a system program that performs system services for an application program processing a transaction.

Distributed processing of a transaction within an SNA network occurs when transaction programs communicate by exchanging information over the sessions between their LUs, treating the session as a resource that is shared between the programs. Figure 1-1 illustrates the connection of two programs to SNA resources, including a session between their LUs.

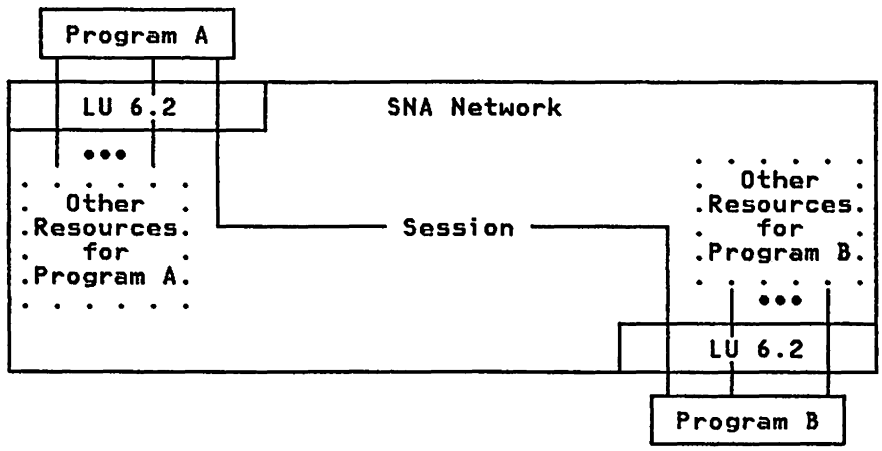


Figure 1-1. Transaction Programs and SNA Resources

The "other resources" shown in the figure may include other sessions as well as local files and devices. The other sessions allow program A or program B to communicate with other programs. During the communication between two programs, one program may send a message over the session to another program, requesting access to a local resource of the other program. In this way, a local resource of program B, for example, may become a remote resource of program A.

PROTOCOL BOUNDARY

The LU 6.2 protocol boundary, as the term is used in this publication, is the generic description of the transaction program's logical interface to an SNA network, from the perspective of the transaction program; LU 6.2 provides the protocol boundary between the program and the network. The description is generic in the sense that it provides a syntactical representation of the functions common to all IBM products that implement LU 6.2; the syntactical description is not necessarily of any specific IBM product. IBM products implementing LU 6.2 may provide a programming interface that differs in syntax from the protocol boundary described herein; however, the results achieved are functionally equivalent to the results described in this book. For information about the functional correspondence between the product's programming interface and the protocol boundary described in this book, refer to the IBM product publication describing the product's programming interface.

The generic protocol boundary described herein represents the transaction program's logical interface to SNA and its services, and is the primary subject of this book. The value of a generic description is that the transaction program designer may plan an application that spans many different products using a single generic interface, and then map the design to the individual product-dependent interfaces.

Note: Products may provide additional functions for their transaction programs, that is, product-unique functions that are not described in this book. A given product-unique function may cause information to be sent on an LU 6.2 session, depending on the function; however, the formats and protocols used on the LU 6.2 session are unchanged. (See SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2 for a definition of the formats and protocols associated with LU 6.2 sessions.) When designing an application that may be executed on different products, the transaction program designer should not depend on the product-specific functions being available across the different products.

INTERPROGRAM COMMUNICATION

Among the services that SNA and, in particular, LU 6.2 provides is interprogram communication. IBM products implementing LU 6.2 provide this service as Advanced Program-to-Program Communication (APPC). Refer to the individual IBM product publications for details of their APPC implementations.

Interprogram communication permits distribution of the processing of a transaction among multiple programs within a network. The programs coordinate the distributed processing by exchanging control information or data. The protocol boundary provides the structure for programs to communicate with one another in order to process a transaction. This structure meets the following requirements, described in terms of their SNA realization:

Simultaneous activation — Many distributed applications require their component programs to be active simultaneously. If the sender of a request waits for the reply, the sending program is depending upon timely execution by its partner. SNA achieves this by simply carrying the program name in the request and letting the receiving LU create an instance of the desired partner program. This concept is recognizably that of transactions, so in SNA the communicating programs are called transaction programs. It follows that distributed transactions are executed by distributed transaction programs.

Efficient allocation — Just as programs use local resources by asking the LU for access to them, programs ask the LU for access to sessions for use as interprogram communication resources. However, the program is not really concerned with the session, which is (usually) a long-term connection between LUs. Nor is the program concerned with the possibility of other programs using the session before or after its own use. What the program asks the LU for is a period of exclusive use of a session, that is, for an abstract resource that is the unit of sharing of the session resource. This resource is called a conversation.

Conversation overhead — Conversations should be efficient in allocation, data transfer, and deallocation. For instance, what the programs see as two short messages, perhaps an inquiry and its reply, should result in two short messages flowing in the network. The LU achieves this by multiplexing conversations over a pool of sessions, scheduling each session as a serially reusable resource.

Conversation lifetime — Conversations last for a time that is determined only by the communicating programs. So, conversations vary from a single, short message to many exchanges of long or short messages. A conversation could continue indefinitely, terminated only by failures.

Two-way alternate data transfer — Conversations use two-way alternate (half duplex) data transfer. This makes it easier to

write transaction programs, in contrast to two-way simultaneous (full duplex) transfer of data which experience shows leads to more complicated and error-prone programs.

Attention mechanism — Conversations include an attention mechanism to handle asynchronous, but non-error, events.

Error notification — Conversations provide each program with a method to notify its partner of errors when they are detected.

Commitment control — When errors occur, recovery is greatly simplified if the changes that a program has been making to its resources can be made to appear atomic; for example, if resources A and B are changed, then after a failure, B will be observed to have changed if and only if A is also observed to have changed. Committing changes atomically is a service that SNA extends to distributed transaction programs. SNA calls this the sync point[3] service. Conversations are defined to the sync point service in each LU as either being protected by sync point or as being unprotected. In the latter case, the transaction programs are themselves responsible for error recovery synchronization.

Symmetry — Conversations are allocated by one active program, but all other protocols (data transfer, attention, error notification, and deallocation) are fully symmetric.

Mode of service — The program allocating the conversation names the desired mode of transmission service, such as "interactive" or "batch," to be provided by the network.

Levels of conversations — In order to adequately serve the needs of system programs and application transactions, two levels of conversations exist: basic conversations, for system transaction programs, and mapped conversations, for application transaction programs.

Subset definition — A subsetting is defined for LU 6.2 by a base set of functions and a limited number of option sets. IBM products that implement LU 6.2 all provide the base set of functions, and may provide any of the option sets.

---

[3] Sync point is a shortened term for synchronization point.

CHAPTER 2. LU 6.2 PROTOCOL BOUNDARY

The LU 6.2 protocol boundary is a generic interface between transaction programs and the SNA network. The protocol boundary permits access to SNA services and resources, especially the services and resources associated with interprogram communication. By means of interprogram communication, distributed transactions can be designed and implemented.

The distributed processing of a transaction also requires access to system services and resources not related to interprogram communication; however, such services and resources are product-dependent. The reader should refer to the individual product publications for information about a particular product's programming interface to resources such as disk or tape files, input/output devices, and processor main storage, and to non-SNA services that the product provides.

INTERPROGRAM COMMUNICATION

The protocol boundary permits transaction programs to communicate with one another without being involved in the interactions that take place within the network. Figure 2-1 shows two programs connected through the SNA network. The LUs are connected by an LU-LU session, and the programs are connected by a conversation allocated on the session. For each LU-LU session, one LU is the contention winner of the session and the other LU is the contention loser of the session. These terms relate to how contention is resolved when the two LUs attempt to allocate a conversation on the session at the same time. Specific details are given in SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2.

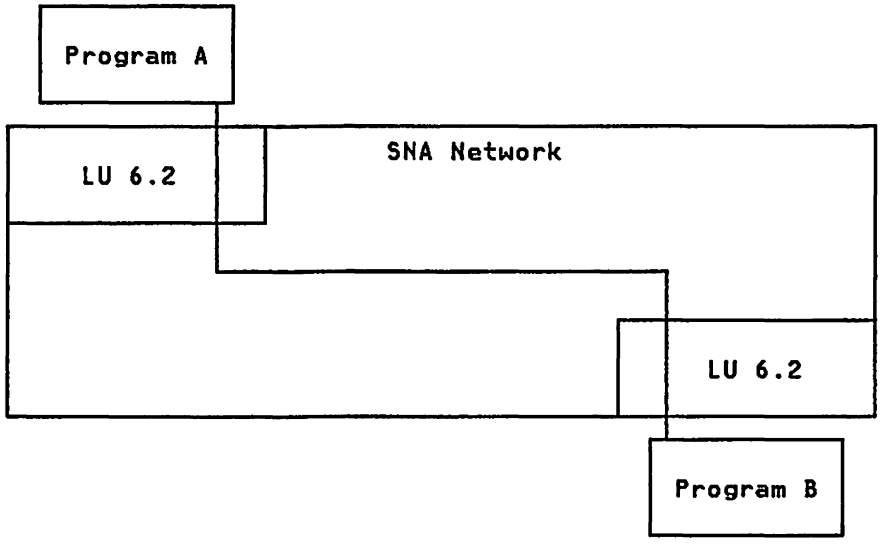


Figure 2-1. Program-to-Program Connection Through the SNA Network

From the programs' view, only the conversation is visible. The activation of the session and actual messages that the LUs exchange on that session are hidden from the programs. Only the delays associated with the buffering and transmission of information within the network are apparent to the programs. The program-to-program connection can therefore be represented as shown in Figure 2-2.

Figure 2-2. Effective Program-to-Program Connection

This view of program-to-program connection can be extended to a more general configuration of interconnected programs. Figure 2-3 shows an example of one way in which seven programs can be interconnected. The interconnection is logical; the physical configuration of the network is not apparent to the programs.

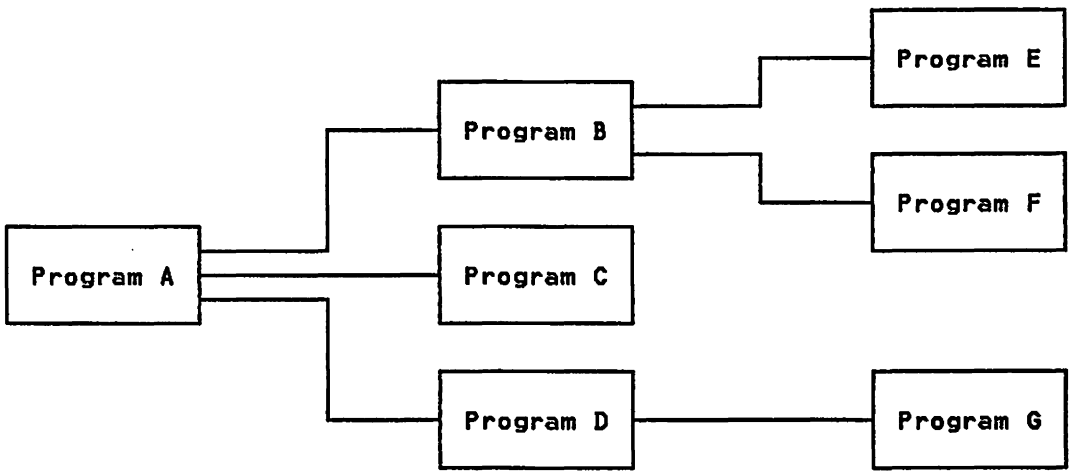


Figure 2-3. A Configuration of Interconnected Programs

The configuration of interconnected programs changes over time. In the example shown in Figure 2-3, the configuration may have evolved as follows:

1. Program A connects to Program B, then to Program C, and then to Program D.
2. Program B connects to Program E and then to Program F.
3. Program D connects to Program G.

This configuration may have evolved in other ways, as well, and it may be an interim configuration that ultimately grows to a much larger configuration of interconnected programs. All configurations of interconnected programs, however large, are made up of program-to-program connections between pairs of programs. One program initiates the interconnection process; in Figure 2-3, the initiating program is Program A.

PROTOCOL BOUNDARY STRUCTURE

The protocol boundary is a structured interface. It is defined by means of formatted functions, called verbs, and the protocols for the verbs. The protocols are the allowed sequences of verbs, that is, the order in which a transaction program can issue verbs. The protocols are defined in terms of resource states. A transaction program can issue a particular verb only when the resource to which that verb applies is in the appropriate state for that verb.

The verbs and states that represent the LU 6.2 protocol boundary enable the user to design distributed transactions, processed by distributed transaction programs. The number of transaction programs can be small, involving only two programs, or large, involving many programs. The transaction can have a fixed structure in which the processing by all programs is predetermined at design time, such as a single inquiry and reply between two programs. In contrast, the transaction can have a flexible structure in which the programs involved and the processing are determined at execution time, possibly varying from one invocation of the transaction to the next; an example is the updating of information in a distributed data base.

An overview description of the verbs is given in "Chapter 3. Transaction Program Verbs". The detailed descriptions of the verbs are given in "Chapter 4. Conversation Verbs" and "Chapter 5. Control-Operator Verbs". Resource states associated with the conversation verbs are described in "Chapter 4. Conversation Verbs".

This page intentionally left blank

CHAPTER 3. TRANSACTION PROGRAM VERBS

The LU 6.2 protocol boundary is defined by verbs that request the LU to perform services. The verbs are described from the transaction programmer's view of the LU 6.2 protocol boundary. Events occurring below the protocol boundary and not apparent at the boundary are not described. Refer to SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2 for details of events that occur below the protocol boundary.

TRANSACTION PROGRAM STRUCTURE AND EXECUTION

All transaction programs have the following general structure:

```
name: PROCEDURE ( resource-id [,pip1 [,... [,pipn] ] ] );
          •
          •   verbs and other program statements
          •
      RETURN;
    END name;
```

The elements of the transaction program structure are:

name is the name of the transaction program. The transaction program name is carried in the allocation request sent by a partner program. The LU receiving the request locates the program by name and creates a new instance,[1] or executable copy, of the program. The location of the program, such as in a program library, is product-dependent.

PROCEDURE begins the main procedure of the transaction program.

resource-id is the name of the variable in which the LU places the resource ID of the conversation on which the allocation request was received. The conversation connects this transaction program to the partner program that sent the allocation request.

Note: The description in this book assumes that transaction programs are always started by means of an allocation request received on a conversation. The manner in which a product starts the first program of an interconnected configuration of programs is product-dependent. For example, the first program may be started in response to a "load" request from an operator.

pip1,...,pipn are the names of the variables in which the LU places program initialization parameters (PIPs) 1 through n. Product send and receive support of PIPs is optional; see Figure C-3 in "Appendix C. Symbol String Conventions" for details. The PIPs are supplied by the allocating program. The contents of the PIPs have meaning only to the transaction programs—they are not examined or acted upon by the LU.

verbs and other program statements represent the combination of verbs, described in this book, and other programming-language statements that make up the transaction-processing portion of the program. Thus, the program's processing of a transaction begins

[1] When it is unambiguous to do so, a program instance is simply referred to as a program.

with the first verb or other program statement after the PROCEDURE statement. It ends with the last verb or other program statement preceding the RETURN statement, or with the processing implied by the RETURN statement (discussed next).

RETURN ends execution of the program by returning control to the LU. As part of the LU's processing of the RETURN statement, it deallocates all conversations (and other resources) that the program has not, itself, deallocated. Depending on the product, the LU may perform other resource-related functions, including the execution of verb functions for conversations still allocated, before deallocating the resources.

END name identifies the physical end of the program. It is the last statement in the program.

Note: The PROCEDURE, RETURN, and END statements are not described elsewhere in this book. They are presented here only to illustrate the general structure of all transaction programs. IBM products implementing LU 6.2 may provide programming language statements that differ in syntax from this description. However, the functions of the product programming language statements are equivalent to the functions described here.

Program execution, in terms of the verbs, occurs when the transaction program issues a verb and the LU executes it; verbs are issued and executed one at a time. When the program issues a verb, the program's processing is suspended while the LU executes the verb. The program resumes processing when the LU returns control to the program. The program may then issue another verb.

Conversations use two-way alternate data transfer. Once a conversation is allocated, send-receive relationship is established between the programs connected to the conversation. One program issues verbs to send data and the other program issues verbs that receive the data. When the sending program finishes sending data, it transfers control of sending data to the other program.

The LUs at each end of a conversation have a buffer for sending and receiving the data on the conversation. When the program issues a verb that sends data, it specifies an area containing the data. The LU moves the data to its send buffer, accumulating the data behind any data from previous verbs. The LU transmits the data, or flushes its send buffer, when either it accumulates a sufficient amount for transmission, or the program issues a verb that explicitly causes the LU to transmit the accumulated data. The amount of data sufficient for transmission is determined by the maximum size request unit that can be sent on the session on which the conversation is allocated. The amount can vary from one session to another, and therefore from one conversation to another.

As incoming data arrives on a conversation, the LU places the data in its receive buffer, accumulating the data behind any it previously received. When the program issues a verb that receives data, it specifies an area in which the LU is to place the data. The LU moves the requested amount of data from the front of its receive buffer to the area specified by the program. In this way, the LU can accumulate incoming data in its receive buffer in advance of the program issuing the verb, or verbs, that receive the data.

Verbs are defined that send information other than data. These verbs cause the LU to flush its send buffer and then place the information at the front of the buffer, behind which it accumulates data from subsequent verbs. The receiving LU accumulates this information in its receive buffer in the order it is received, with reference to other information including data.

Program execution ends when the program returns control to the LU at the completion of the transaction. This is accomplished by the RETURN statement.

VERB OVERVIEW

This section presents an overview of the verbs in terms of their individual functions. The verbs are divided into categories. These categories are:

Conversation verbs
Control-operator verbs

Each category defines a major subdivision of the LU 6.2 protocol boundary. The conversation verbs define the means for program-to-program communication. The control-operator verbs define the means for program or operator control of the LU's resources.

In the following overview, and in the remaining chapters of this book, the verbs are described from the perspective of the transaction program issuing the verb. From this point of view, the program issuing the verb is referred to as the local program, and the program at the other end of the conversation is referred to as the remote program. Similarly, the LU processing the local program is referred to as the local LU, and the LU processing the remote program is referred to as the remote LU.[2] The overview description of the conversation verbs and control-operator verbs follows.

CONVERSATION VERBS

The conversation verbs provide program-to-program communication by means of conversations between programs. The following conversation types are defined:

Mapped
Basic

The verbs defining the conversation protocol boundary are divided into subcategories based on the conversation type. The subcategories of verbs are:

Mapped conversation verbs
Type-independent conversation verbs
Basic conversation verbs

An overview of the conversation verbs follows.

Mapped Conversation Verbs

The mapped conversation verbs are intended for use by application transaction programs. These verbs provide functions that are suitable for application programs written in high-level programming languages. A brief description of the mapped conversation verbs follows.

MC_ALLOCATE allocates a mapped conversation connecting the local transaction program to a remote transaction program. A unique resource ID is assigned to the mapped conversation. This verb is issued prior to any verbs that refer to the mapped conversation.

MC_CONFIRM sends a confirmation request to the remote transaction program and waits for a reply, in order for the two programs to synchronize their processing.

MC_CONFIRMED sends a confirmation reply to the remote transaction program, in order for the two programs to synchronize their processing. The program issues the verb in response to receiving a confirmation request.

MC_DEALLOCATE deallocates a mapped conversation resource from the transaction program. The program issues this verb when it is finished using the mapped conversation.

---

[2] When it is unambiguous to do so, the local program is simply referred to as the program, and the local LU is simply referred to as the LU.

MC_FLUSH transmits all information that the LU has buffered, such as data records from preceding MC_SEND_DATAs.

MC_GET_ATTRIBUTES returns information pertaining to a mapped conversation. Examples of information that may be requested are the mode name, the name of the LU at which the remote transaction program is located, or the synchronization level allocated for the mapped conversation.

MC_POST_ON_RECEIPT requests posting of the specified mapped conversation when information is available for the program to receive. The information can be a data record, mapped conversation status, or a request for confirmation or sync point.

MC_PREPARE_TO_RECEIVE changes the mapped conversation from send state to receive state in preparation to receive data. A SEND indication is sent to the remote program. The remote program's end of the mapped conversation changes to send state when the program receives the SEND indication.

MC_RECEIVE_AND_WAIT waits for information to arrive on the mapped conversation and then receives the information. If information is already available, the program receives it without waiting. The information can be a data record, mapped conversation status, or a request for confirmation or sync point. Control is returned to the program with an indication of the type of information. The verb can be issued when the mapped conversation is in send state. In this case, the verb first sends a SEND indication to the remote program, changing the mapped conversation to receive state, and then waits for information to arrive.

MC_RECEIVE_IMMEDIATE receives any information that is available from the specified mapped conversation, but does not wait for information to arrive. The information (if any) can be a data record, mapped conversation status, or a request for confirmation or sync point. Control is returned to the program with an indication of whether any information was received and, if so, the type of information.

MC_REQUEST_TO_SEND notifies the remote program that the local program is requesting to enter send state for the mapped conversation. The mapped conversation will be changed to send state when the local program subsequently receives a SEND indication from the remote program.

MC_SEND_DATA sends one data record to the remote transaction program. The data record consists entirely of data. The program can specify data mapping as a function of the verb, or it can indicate that the data record includes FM headers.

MC_SEND_ERROR informs the remote transaction program that the local program has detected an application error. For example, the local program can issue this verb to inform the remote program of an error it detected in a data record it received, or to reject a confirmation request. Upon successful completion of the verb, the local program is in send state for the mapped conversation and the remote program is in receive state.

MC_TEST tests the mapped conversation to determine whether it has been posted, as a result of a preceding MC_POST_ON_RECEIPT verb, or whether a request-to-send notification has been received.

Type-Independent Conversation Verbs

The type-independent conversation verbs are intended for use with both mapped and basic conversations. These verbs provide functions that span both conversation types. A brief description of the type-independent verbs follows.

BACKOUT restores all protected resources throughout a distributed transaction to their status as of the last synchronization point. Protected resources are those that are protected by the sync point service of LU 6.2.

GET_TYPE returns the type of resource. For mapped conversations the type is MAPPED_CONVERSATION, and for basic conversations the type is BASIC_CONVERSATION.

SYNCPT advances all protected resources throughout a distributed transaction to the next synchronization point.

WAIT waits for posting to occur on any mapped or basic conversation from among a list of mapped and basic conversations. The posting of mapped conversations is performed as a result of a preceding MC_POST_ON_RECEIPT verb issued for each of the mapped conversations. Similarly, the posting of basic conversations is performed as a result of a preceding POST_ON_RECEIPT verb issued for each of the basic conversations.

Basic Conversation Verbs

The basic conversation verbs are intended for use by LU services programs. The LU services programs can provide end-user services or protocol boundaries for end-user application transaction programs. For example, the mapped conversation LU services component issues basic conversation verbs during its processing of mapped conversation verbs. A brief description of the basic conversation verbs follows.

ALLOCATE allocates a conversation connecting the local transaction program to a remote transaction program. The conversation type can be basic or mapped. A unique resource ID is assigned to the conversation. This verb is issued prior to any verbs that refer to the conversation.

CONFIRM sends a confirmation request to the remote program and waits for a reply, in order for the two programs to synchronize their processing.

CONFIRMED sends a confirmation reply to the remote program, in order for the two programs to synchronize their processing. The program issues this verb in response to receiving a confirmation request.

DEALLOCATE deallocates a conversation from the transaction program. The program issues this verb when it is finished using the conversation.

FLUSH transmits all information that the LU has buffered, such as data from preceding SEND_DATAs.

GET_ATTRIBUTES returns information pertaining to a conversation. Examples of information that may be requested are the mode name, the name of the LU at which the remote transaction program is located, or the synchronization level allocated for the conversation.

POST_ON_RECEIPT requests posting of the specified conversation when information is available for the program to receive. The information can be data, conversation status, or a request for confirmation or sync point.

PREPARE_TO_RECEIVE changes the conversation from send state to receive state in preparation to receive data. A SEND indication is sent to the remote program. The remote program's end of the conversation changes to send state when the program receives the SEND indication.

RECEIVE_AND_WAIT waits for information to arrive on the conversation and then receives the information. If information is already available, the program receives it without waiting. The information can be data, conversation status, or a request for confirmation or sync point. Control is returned to the program with an indication of the type of information. The verb can be issued when the conversation is in send state. In this case, the verb first sends a SEND indication to the remote program, changing the conversation to receive state, and then waits for information to arrive.

RECEIVE_IMMEDIATE receives any information that is available from the specified conversation, but does not wait for information to arrive. The information (if any) can be data, conversation status, or a request for confirmation or sync point. Control is returned to the program with an indication of whether any information was received and, if so, the type of information.

REQUEST_TO_SEND notifies the remote program that the local program is requesting to enter send state for the conversation. The conversation will be changed to send state when the local program subsequently receives a SEND indication from the remote program.

SEND_DATA sends data to a remote program. The data format consists of logical records. The amount of data is specified independently of the data format. A logical record contains a length field and a data field. The length field is 2 bytes long; the data field can be any length within the range of 0 to 32765 bytes.

SEND_ERROR informs the remote program that the local program has detected an error. For example, the local program can issue this verb to truncate an incomplete logical record it is sending, to inform the remote program of an error it detected in data it received, or to reject a confirmation request. Upon successful completion of the verb, the local program is in send state for the conversation and the remote program is in receive state.

TEST tests the conversation to determine whether it has been posted, as a result of a preceding POST_ON_RECEIPT verb, or whether a request-to-send notification has been received.

CONTROL-OPERATOR VERBS

The control-operator verbs are intended for use by control-operator transaction programs, that is, programs that assist the control operator in performing functions related to the control of an LU. The verbs defining the control-operator protocol boundary are divided into subcategories based on their functions. The subcategories are:

Change number of sessions verbs
Session control verbs
LU definition verbs

An overview of the control-operator verbs follows.

Change Number of Sessions Verbs

This subcategory of control-operator verbs consists of four verbs called the change-number-of-sessions, or CNOS, verbs. The CNOS verbs change the (LU,mode) session limit, which controls the number of LU-LU sessions per mode name that are available between two LUs for allocation to conversations. In conjunction with changing the (LU,mode) session limit, the CNOS verbs change related operating parameters of the two LUs.

The two LUs may cooperate in the execution of the CNOS verbs by means of a CNOS request and CNOS reply. The LU executing the control-operator transaction program sends a CNOS request to the partner LU. The partner LU invokes an SNA service transaction program called the "CNOS service transaction program" (see "Appendix D. List of SNA Service Transaction Programs"), which causes the partner LU to process the CNOS request and send back a CNOS reply.

The CNOS verbs that a control-operator transaction program may issue are:

CHANGE_SESSION_LIMIT changes the (LU,mode) session limit from one nonzero value to another nonzero value.

INITIALIZE_SESSION_LIMIT changes the (LU,mode) session limit from 0 to a value greater than 0.

RESET_SESSION_LIMIT changes the (LU,mode) session limit from a value greater than 0 to 0.

The CNOS verb that the CNOS service transaction program issues is:

PROCESS_SESSION_LIMIT causes the LU receiving the CNOS request to process the request and send back a CNOS reply to the partner LU.

Session Control Verbs

This subcategory of control-operator verbs consists of two verbs used for session control, one that activates an LU-LU session and one that deactivates an LU-LU session. These verbs are:

ACTIVATE_SESSION activates an LU-LU session between the local LU and a specified LU.

DEACTIVATE_SESSION deactivates a specified LU-LU session. The type of deactivation can be cleanup or normal.

LU Definition Verbs

This subcategory of control-operator verbs is used to define or modify the local LU's operating parameters, examine the parameters, and delete the parameters. These verbs are:

DEFINE_LOCAL_LU initializes or modifies parameter values that control the operation of the local LU.

DEFINE_REMOTE_LU initializes or modifies parameter values that control the operation of the local LU in conjunction with a remote LU.

DEFINE_MODE initializes or modifies parameter values that control the operation of the local LU in conjunction with a group of sessions with a remote LU, the group being identified by a mode name.

DEFINE_TP initializes or modifies parameter values that control the operation of the local LU in conjunction with a local transaction program.

DISPLAY_LOCAL_LU returns parameter values that control the operation of the local LU.

DISPLAY_REMOTE_LU returns parameter values that control the operation of the local LU in conjunction with the remote LU.

DISPLAY_MODE returns parameter values that control the operation of the local LU in conjunction with a group of sessions with a remote LU, the group being identified by a mode name.

DISPLAY_TP returns parameter values that control the operation of the local LU in conjunction with a local transaction program.

DELETE deletes the local LU's operating-parameter values that have been defined by means of the DEFINE verbs.

ABEND CONDITIONS

Certain errors related to the execution of the verbs can cause an abnormal ending (ABEND) of the transaction program. These ABEND conditions are a direct consequence of an invalid specification or execution of a verb. When the LU terminates a program because of an ABEND condition, it deallocates all of the program's active conversations. Depending on the product, the LU may abnormally deallocate the conversations, or deallocate the conversations in the same way it does for the RETURN statement (see the RETURN statement under "Transaction Program Structure and Execution" on page 3-1).

The ABEND conditions are:

Parameter Check occurs when the program issues a verb for which local support is not available, or when the program specifies a verb parameter with an invalid argument. The source of the invalid argument is considered to be part of the program definition. (Contrast this definition with the definition of the return code, PARAMETER_ERROR, in the section "Return Codes" in Chapter 4.) The detailed verb descriptions list the applicable parameter checks.

The omission of a required parameter, the specification of an undefined parameter, and the specification of an undefined argument on a parameter that requires one of a defined set of keywords are also parameter check conditions. The parameter checks for the omission of a required parameter and for the specification of an undefined parameter apply to all verbs. The parameter check for an undefined keyword argument applies to all verbs that specify one or more parameters with keyword arguments. These parameter checks are not explicitly listed with each detailed verb description.

State Check occurs when the program attempts to issue a verb for a conversation that is in a state in which the verb is not allowed. The section "Conversation States" in Chapter 4 defines the allowable states for each conversation verb. The control-operator verbs do not have states associated with them and therefore do not have any state checks defined.

The individual verb descriptions list the applicable ABEND conditions.

Note: In lieu of treating these ABEND conditions as described here, products may provide a different method for handling the ABEND conditions. For example, a product may return an error indication to the program when it attempts to issue a verb in a state in which the verb is not permitted, allowing the program to continue processing, or a product may provide a compile-time check for the specification of optional verbs and parameters that the product does not support. Refer to the individual product's publications for details about how it treats these conditions.

PRODUCT-SUPPORT SUBSETTING

Product-support subsetting of the verbs is defined by means of functional groups, or sets. A set consists of all the functions that together represent an indivisible group for products to implement. That is, a product implementing a particular set implements all of the functions within that set.

All products implement a subset of LU 6.2 functions called the base set. The functions that are not part of the base set are optional.

The base set and option sets are defined in terms of the LU 6.2 protocol boundary, as follows:

Base set is the set of LU 6.2 verbs, parameters, return codes, and what-received indications that all products support.

Option sets are the sets of LU 6.2 verbs, parameters, return codes, and what-received indications that a product may support, depending on the product. A product may support any number of option sets, or none. For each option set supported, all verbs, parameters, return codes, and what-received indications in that set are supported.

The base set and option sets are further defined in terms of local support and remote support.

Local support is the support of LU 6.2 verbs, parameters, return codes, and what-received indications that the product provides at the local end of a conversation, as seen by the local transaction program. The program may issue an optional verb or parameter only when the local product supports the option set. An attempt by the program to issue an optional verb or parameter for which local support is not available is considered an ABEND condition (see

"ABEND Conditions" on page 3-7). An optional return code or what-received indication can be reported to the program only when the local product supports the option set.

Remote support is the support of verbs and parameters that the product provides at the remote end of a conversation, as seen by the local transaction program. (Remote support does not apply to the return codes and what-received indications.) Only certain verbs and parameters invoke processing at the remote end of the conversation; the other verbs and parameters are processed entirely at the local end of the conversation. When the program issues a verb or parameter that invokes remote processing, and the remote product does not provide remote support for the verb or parameter, a return code indicating the lack of support is reported to the program. The return code can be reported on the verb for which remote support is not available or on a later verb, depending on the verb.

The base and optional support for the conversation verbs and control-operator verbs is defined in "Appendix A. Base and Option Sets for Product Support".

Note: The base- and option-set definition for product support described in this book applies only to LU 6.2 products that provide an application programming interface (API) for user-written programs that is equivalent to the conversation verbs. The definition does not apply to LU 6.2 products that are not user-programmable, or to products that are user-programmable but do not provide an API equivalent to the conversation verbs; such products need support only the LU 6.2 functions required for their applications.

VERB DESCRIPTION FORMAT

This section explains the format used to describe the verbs in the following chapters. The verb descriptions are presented alphabetically, by name, in terms of each verb's function, syntactic format, parameters, state changes, ABEND conditions, and usage notes.

The description of each verb begins with a brief explanation of its function.

The verb's syntax is described next using a format box. The general representation of the format box is shown in Figure 3-1 on page 3-10.

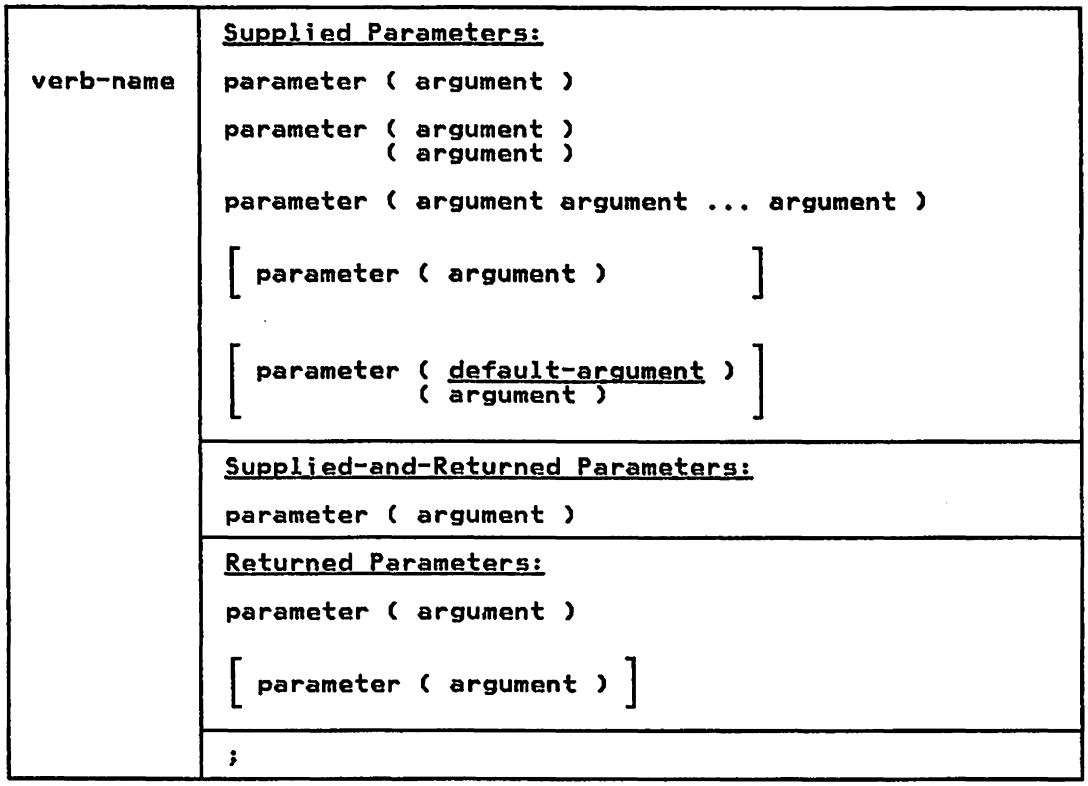


Figure 3-1. Format Box for Representing Verb Syntax

As shown in the preceding general format box, the syntax description for each verb includes a verb name, verb parameters, and the ending delimiter ";" (semicolon). The number of verb parameters depends on the verb, and a verb may not have any parameters.

Parameter names are shown as uppercase keywords. Parameter arguments are shown as uppercase keywords, as "variables," or as a combination of keywords and "variables." An argument keyword is used to show a specific argument value associated with the parameter. An argument "variable" is used to show that the argument value can vary; it can be program data, for example.

Some parameters show a vertical list of argument keywords (possibly combined with "variables"). The vertical list means the arguments are limited to those within the list, one of which is specified when the verb is issued. Other parameters show an argument list as "variable1 ... variablen." The number of arguments in the argument list depends on the verb; the number may be constant or it may vary from one issuance of the verb to the next.

The parameters are grouped as "supplied parameters," "supplied-and-returned parameters," or "returned parameters."

- Supplied parameters contain arguments whose values are supplied by the program when it issues the verb.
- Supplied-and-returned parameters contain arguments whose values are supplied by the program when it issues the verb and whose values are returned to the program when it resumes processing.
- Returned parameters contain arguments whose values are returned to the program when it resumes processing.

Some parameters are shown within brackets. The bracket notation is used to show which parameters may be omitted when the verb is issued. It it also used for cross-publication reference purposes, so that oth-

er SNA and product publications that refer to the verbs in this book may omit references to the bracketed parameters. In particular:

- Some bracketed supplied parameters have multiple arguments with one being a default argument, shown as underscored. Omission of any of these parameters is treated as if the default argument was specified on the parameter.
- Other bracketed supplied parameters have no default argument. Omission of any of these parameters is treated as described for the parameter.
- If a bracketed returned parameter is omitted, the argument value is not returned.

Following the syntax is a description of the verb's parameters. Included is a list of the return codes that can be returned to the transaction program when it resumes processing.

The changes, if any, to the state of the conversation at the protocol boundary are described next. The state changes occur as a result of executing the verb.

After the description of state changes, the ABEND conditions are given for each verb.

Finally, notes are given to describe certain aspects of the verb's usage in order to further clarify the actions of the verb.

Notes:

1. Products may provide application programming interfaces (APIs) that differ from the verb syntax described in this book. For example, a product may have different names for operations that are equivalent to the verbs and parameters described herein; it may combine the functions of certain verbs into one operation, such as the functions of MC_SEND_DATA and MC_CONFIRM; and it may separate the functions of a single verb into distinct operations, such as separating the functions of MC_ALLOCATE into an operation that acquires the session and an operation that allocates the conversation on the session. These are syntactical, not functional, differences.

2. The bracket notation used in the syntax diagrams is unrelated to the product-support subsetting described in this book. See "Product-Support Subsetting" on page 3-8 and "Appendix A. Base and Option Sets for Product Support" in Appendix A for details about product support. The bracket notation is also unrelated to any product's API definition. The product may allow a different set of parameters to be omitted, if any, and have different defaults for the supplied parameters. Refer to the product's programming publications for details of its API.

This page intentionally left blank

CHAPTER 4. CONVERSATION VERBS

This chapter describes the category of verbs called conversation verbs. The conversation verbs define the LU 6.2 conversation protocol boundary. These verbs are used for program-to-program communications over a conversation connecting two programs. Each conversation is of a specific type:

Mapped
Basic

The conversation verbs are divided into subcategories, based on the conversation type to which they apply:

Mapped conversation verbs
Type-independent conversation verbs
Basic conversation verbs

The mapped conversation verbs apply to mapped conversations. The type-independent conversation verbs apply to both mapped and basic conversations. The basic conversation verbs apply to basic conversations, and to mapped conversations for use by the mapped conversation LU services component. Refer to SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2 for a description of the mapped conversation LU services component.

Following the descriptions of the conversation verbs is a description of conversation states that allow issuance of the verbs, and a description of the return codes that apply to the conversation verbs.

VERB DESCRIPTIONS

The detailed descriptions of the mapped, type-independent, and basic conversation verbs follow.

MAPPED CONVERSATION VERBS

This section describes the subcategory of conversation verbs called mapped conversation verbs. These verbs are intended for use by application transaction programs. They provide functions, such as data mapping (a product option), that make the verbs suitable for application programs written in a high-level programming language. Additionally, these verbs conceal from the application program the logical-record data-stream format that a program using the basic conversation verbs must manage. The detailed descriptions of the mapped conversation verbs follow.

Note: Every conversation is either a mapped or basic conversation. The mapped conversation verbs are used for operations only on mapped conversations. Thus, throughout the descriptions of the mapped conversation verbs, references are made only to mapped conversations. The program allocates a mapped conversation when it issues the MC_ALLOCATE verb. Contrast this with the basic conversation verb, ALLOCATE, which can allocate a conversation of either type, mapped or basic.

MC_ALLOCATE

Allocates a session between the local LU and a remote LU, and on that session allocates a mapped conversation between the local transaction program and a remote transaction program. A resource ID is assigned to the mapped conversation. This verb is issued prior to any verbs that refer to the mapped conversation.

```
MC_ALLOCATE

Supplied Parameters:

LU_NAME ( OWN )
        ( OTHER ( variable ) )

MODE_NAME ( variable )

TPN ( variable )

[                  ( WHEN_SESSION_ALLOCATED )                    ]
[ RETURN_CONTROL ( DELAYED_ALLOCATION_PERMITTED )                ]
[                  ( IMMEDIATE )                                 ]

[             ( NONE )                                           ]
[ SYNC_LEVEL  ( CONFIRM )                                        ]
[             ( SYNCPT )                                         ]

[             ( NONE )                                           ]
[ SECURITY    ( SAME )                                           ]
[             ( PGM ( USER_ID ( variable ) PASSWORD ( variable ) ]
[                     PROFILE ( variable ) ) )                   ]

[ PIP ( NO )                                                     ]
[     ( YES ( variable1 variable2 ... variablen ) )              ]

Returned Parameters:

RESOURCE ( variable )

RETURN_CODE ( variable )

;
```

Supplied Parameters:

LU_NAME specifies the name of the remote LU at which the remote transaction program is located. This LU name is any name by which the local LU knows the remote LU for the purpose of allocating a mapped conversation. The local LU transforms this locally-known LU name to an LU name used by the network, if the names are different.

- **OWN** specifies that the remote transaction program is located at the same LU as the local program.
- **OTHER** specifies that the remote transaction program is located at another LU. The specified variable contains the LU name.

MODE_NAME specifies the mode name designating the network properties for the session to be allocated for the mapped conversation. The network properties include, for example, the class of service to be used, and whether data is to be enciphered or translated to ASCII before it is sent. The SNA-defined mode name, SNASVCMG, is not allowed to be specified on the MC_ALLOCATE verb (contrast this with the ALLOCATE verb).

TPN specifies the name of the remote transaction program to be connected at the other end of the mapped conversation. TPN cannot specify an SNA service transaction program name at the mapped conversation protocol boundary. (See "Appendix D. List of SNA Service Transaction

Programs" for more details about SNA service transaction program names.)

RETURN_CONTROL specifies when the local LU is to return control to the local program, in relation to the allocation of a session for the mapped conversation. An allocation error resulting from the local LU's failure to obtain a session for the mapped conversation is reported either on this verb or a subsequent verb, depending on the argument specified for this parameter. An allocation error resulting from the remote LU's rejection of the allocation request is reported on a subsequent verb.

- **WHEN_SESSION_ALLOCATED** specifies to allocate a session for the mapped conversation before returning control to the program. An error in allocating a session is reported on this verb.
- **DELAYED_ALLOCATION_PERMITTED** specifies to allocate a session for the mapped conversation after returning control to the program. An error in allocating a session is reported on a subsequent verb.
- **IMMEDIATE** specifies to allocate a session for the mapped conversation if a session is immediately available, and return control to the program with a return code indicating whether a session is allocated.
  - A return code of OK indicates a session is immediately available and is allocated for the mapped conversation. A session is immediately available when it is active, it is not allocated to another mapped conversation, and the local LU is the contention winner for the session.
  - A return code of UNSUCCESSFUL indicates a session is not immediately available. Allocation is not performed.

  An error in allocating a session that is immediately available is reported on this verb.

SYNC_LEVEL specifies the synchronization level that the local and remote transaction programs can use on this mapped conversation.

- **NONE** specifies that the transaction programs will not perform confirmation or sync point processing on this mapped conversation. The programs will not issue any verbs and will not recognize any returned parameters relating to these synchronization functions.
- **CONFIRM** specifies that the transaction programs can perform confirmation processing but not sync-point processing on this mapped conversation. The programs may issue verbs and will recognize returned parameters relating to confirmation, but they will not issue any verbs and will not recognize any returned parameters relating to sync point.
- **SYNCPT** specifies that the transaction programs can perform both confirmation and sync-point processing on this mapped conversation. The programs may issue verbs and will recognize returned parameters relating to confirmation or sync point. For sync-point processing, a mapped conversation allocated with this synchronization level is a protected resource.

SECURITY specifies access security information that the remote LU uses to verify the identity of the end user and validate access to the remote program and its resources. The access security information consists of a user ID, a password, and a profile.

- **NONE** specifies to omit access security information on this allocation request.
- **SAME** specifies to use the user ID and profile (if present) from the allocation request that initiated execution of the local program. The password (if present) is not used; instead, the user ID is indicated as being already verified. If the allocation request that initiated execution of the local program contained no access

security information, then access security information is omitted on this allocation request.

- **PGM** specifies to use the access security information that the local transaction program provides on this parameter. The local program provides the information by means of the following arguments:

  - **USER_ID** specifies the variable containing the user ID. The remote LU uses this value and the password to verify the identity of the end user making the allocation request. In addition, the remote LU may use the user ID for auditing or accounting purposes, or it may use the user ID, together with the profile (if present), to determine which remote programs the local program may access and which resources the remote program may access.

  - **PASSWORD** specifies the variable containing the password. The remote LU uses this value and the user ID to verify the identity of the end user making the allocation request.

  - **PROFILE** specifies the variable containing the profile. The remote LU may use this value, in addition to or in place of the user ID, to determine which remote programs the local program may access, and which resources the remote program may access.

  Specifying a null value for any of the access security arguments is equivalent to omitting the argument.

PIP specifies program initialization parameters for the remote transaction program.

- **NO** specifies that PIP data is not present.

- **YES** specifies that PIP data is present.

  - **variable1 variable2 ... variablen** contain the PIP data to be sent to the remote transaction program. The PIP data consists of one or more subfields, each of which is specified by a separate variable; variables 1 through n correspond to subfields 1 through n. If a variable is omitted in the PIP parameter or it is of null value, the corresponding PIP subfield is made to be of zero length. The number of PIP subfields must agree with the number of PIP variables specified on the remote program's PROC statement (see "Transaction Program Structure and Execution" in Chapter 3).

Returned Parameters:

RESOURCE specifies the variable in which the resource ID is to be returned. The length and actual format of the resource ID is product dependent. The resource ID is returned to the program when the return code is either OK or ALLOCATION_ERROR.

RETURN_CODE specifies the variable in which a return code is returned to the local program. The return code indicates the result of verb execution. The RETURN_CONTROL parameter determines which of the following return codes can be returned to the program.

- If RETURN_CONTROL(WHEN_SESSION_ALLOCATED) is specified, one of the following return codes is returned:

  - OK
  - ALLOCATION_ERROR (with one of the following subcodes)
    - ALLOCATION_FAILURE_NO_RETRY
    - ALLOCATION_FAILURE_RETRY
    - SYNC_LEVEL_NOT_SUPPORTED_BY_LU
  - PARAMETER_ERROR (for one of the following reasons)
    - Invalid LU name
    - Invalid mode name

- If RETURN_CONTROL(DELAYED_ALLOCATION_PERMITTED) is specified, one of the following return codes is returned:

  - OK
  - PARAMETER_ERROR (for one of the following reasons)
    - Invalid LU name
    - Invalid mode name

- If RETURN_CONTROL(IMMEDIATE) is specified, one of the following return codes is returned:

  - OK
  - ALLOCATION_ERROR (with the following subcode)
    - SYNC_LEVEL_NOT_SUPPORTED_BY_LU
  - PARAMETER_ERROR (for one of the following reasons)
    - Invalid LU name
    - Invalid mode name
  - UNSUCCESSFUL (for the following reason)
    - Session not immediately available

State Changes (When RETURN_CODE indicates OK):

Send state is entered.

ABEND Conditions:

Parameter Check

- The program does not have mapped conversation support defined.
- LU_NAME(OWN) is specified and not supported.
- MODE_NAME specifies the SNA-defined mode name, SNASVCMG.
- TPN specifies an SNA service transaction program.
- TPN specifies a null (zero length) value.
- RETURN_CONTROL (DELAYED_ALLOCATION_PERMITTED) is specified and not supported.
- RETURN_CONTROL(IMMEDIATE) is specified and not supported.
- SYNC_LEVEL(SYNCPT) is specified and not supported.
- SECURITY(SAME) is specified and not supported.
- SECURITY(PGM(USER_ID(variable) PASSWORD(variable))) is specified and not supported.
- SECURITY(PGM(PROFILE(variable))) is specified and not supported.
- PIP(YES(variable)) is specified and not supported.

State Check

None

Notes:

1. Depending on the product, the LU may send the allocation request to the remote LU as soon as it allocates a session for the mapped conversation. Alternatively, the LU may buffer the allocation request until it accumulates from the PIP parameter of this verb or from one or more subsequent MC_SEND_DATA verbs a sufficient amount of information for transmission, or until the local program issues a subsequent verb other than MC_SEND_DATA that explicitly causes the LU to flush its send buffer. The amount of information that is sufficient for transmission depends on the characteristics of the session allocated for the mapped conversation, and can vary from one session to another.

2. The local program can ensure that the remote program is connected as soon as possible by issuing MC_FLUSH immediately after MC_ALLOCATE.

3. Two LUs connected by a session may both attempt to allocate a mapped conversation on the session at the same time. This is called contention. Contention is resolved by making one LU the contention winner of the session and the other LU the contention loser of the session. The contention-winner LU allocates a mapped conversation on a session without asking permission from the contention-loser LU. Conversely, the contention-loser LU requests permission from the contention-winner LU to allocate a mapped conversation on the session, and the contention-winner LU either grants or rejects the request.

4. If the program issues MC_ALLOCATE with the parameter RETURN_CONTROL(DELAYED_ALLOCATION_PERMITTED), the LU delays allocation of the session until it flushes its send buffer. At that time the LU allocates the session and transmits the allocation request to the remote LU. The program is unaffected by the delayed allocation of the session, with one exception: When the LU allocates a contention-loser session, it does so by transmitting the allocation request and then waiting for information to arrive before returning control to the program. This can affect the sequence of the verbs that the program can issue.

   For example, suppose the program has the following sequence of verbs:

   MC_ALLOCATE with
   RETURN_CONTROL(DELAYED_ALLOCATION_PERMITTED)

   MC_PREPARE_TO_RECEIVE with TYPE (FLUSH)

   MC_REQUEST_TO_SEND

   In this example, assume the program is using MC_REQUEST_TO_SEND to prompt the remote program to begin sending information, instead of requesting send control. However, if the LU allocates a contention-loser session (and an allocation error or resource failure does not occur), control is not returned to the program after it issues the MC_PREPARE_TO_RECEIVE until the remote program sends some information. If the remote program waits for the REQUEST_TO_SEND notification before sending any information, a deadlock condition occurs. This deadlock can be avoided by issuing the MC_ALLOCATE with either RETURN_CONTROL (WHEN_SESSION_ALLOCATED) or RETURN_CONTROL (IMMEDIATE).

5. SYNC_LEVEL(SYNCPT) permits use of the SYNCPT and BACKOUT verbs and the Resynchronization transaction program (an SNA service transaction program), to aid in maintaining consistency across all protected resources within a distributed logical unit of work. The Resynchronization program performs sync point resynchronization, which maintains this consistency when session failure and reinitiation occurs. See SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2 for more details of sync point resynchronization.

6. Each LU indicates at session activation time whether it will accept LU security parameters on allocation requests the partner LU sends. If the remote LU will not accept any security parameters from the local LU, and the local program specifies SECURITY(SAME) or SECURITY(PGM(...)), the local LU downgrades the specification to SECURITY(NONE). Similarly, if the remote LU will not accept the local LU's verification of the user ID and password, and the local program specifies SECURITY(SAME), the local LU downgrades the specification to SECURITY(NONE).

7. The remote program is connected to the other end of the mapped conversation in receive state.

8. The program uses the resource ID, returned to the program on the RESOURCE parameter, on all subsequent mapped conversation verbs it issues for this mapped conversation.

9. References in this verb description to a program being in a particular state are only in terms of the allocated mapped conversation.

MC_CONFIRM

Sends a confirmation request to a remote transaction program and waits for a reply. This verb allows the local and remote programs to synchronize their processing with one another. The LU flushes its send buffer as a function of this verb.

MC_CONFIRM	**Supplied Parameters:** **RESOURCE (variable)**
	Returned Parameters: **RETURN_CODE (variable)** **REQUEST_TO_SEND_RECEIVED (variable)**
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID. The mapped conversation must be allocated with a synchronization level of CONFIRM or SYNCPT.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the local program. The return code indicates the result of verb execution.

- OK (remote program replied MC_CONFIRMED)
- ALLOCATION_ERROR
- BACKED_OUT
- DEALLOCATE_ABEND
- FMH_DATA_NOT_SUPPORTED
- MAPPING_NOT_SUPPORTED
- MAP_NOT_FOUND
- MAP_EXECUTION_FAILURE
- PROG_ERROR_PURGING
- RESOURCE_FAILURE_NO_RETRY
- RESOURCE_FAILURE_RETRY

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received. The indication is either YES or NO.

- YES indicates a REQUEST_TO_SEND notification has been received from the remote transaction program. The remote program has issued MC_REQUEST_TO_SEND, requesting the local program to enter receive state and thereby place the remote program in send state.
- NO indicates a REQUEST_TO_SEND notification has not been received.

State Changes (when RETURN CODE indicates OK):

Receive state is entered when the verb is issued in defer state following MC_PREPARE_TO_RECEIVE.

Reset state is entered when the verb is issued in defer state following MC_DEALLOCATE.

No state change occurs when the verb is issued in send state.

ABEND Conditions:

Parameter Check

- This mapped conversation was allocated with SYNC_LEVEL(NONE).
- RESOURCE specifies an unassigned resource ID.

State Check

The mapped conversation is not in send or defer state.

Notes:

1. The program may use this verb for various application-level functions. For example:

   - The program may issue this verb immediately following an MC_ALLOCATE in order to determine whether the allocation of the mapped conversation is successful before sending any data records.

   - The program may issue this verb as a request for acknowledgement of data records it sent to the remote program. The remote program may respond by issuing MC_CONFIRMED as an indication that it received and processed the data records without error, or by issuing MC_SEND_ERROR as an indication that it encountered an error.

2. When REQUEST_TO_SEND_RECEIVED indicates YES, the remote program requests the local program to enter receive state and thereby place the remote program in send state. A program enters receive state by means of the MC_PREPARE_TO_RECEIVE or MC_RECEIVE_AND_WAIT verb. The partner program enters the corresponding send state when it issues an MC_RECEIVE_AND_WAIT or MC_RECEIVE_IMMEDIATE verb and receives the SEND indication (on the WHAT_RECEIVED parameter).

3. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

MC_CONFIRMED

Sends a confirmation reply to the remote transaction program. This verb allows the local and remote programs to synchronize their processing with one another. The local program can issue this verb when it receives a confirmation request (see the WHAT_RECEIVED parameter of the MC_RECEIVE_AND_WAIT or MC_RECEIVE_IMMEDIATE verb).

MC_CONFIRMED	Supplied Parameters: RESOURCE (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

State Changes:

Receive state is entered when CONFIRM was received on the preceding MC_RECEIVE_AND_WAIT or MC_RECEIVE_IMMEDIATE.

Send state is entered when CONFIRM_SEND was received on the preceding MC_RECEIVE_AND_WAIT or MC_RECEIVE_IMMEDIATE.

Deallocate state is entered when CONFIRM_DEALLOCATE was received on the preceding MC_RECEIVE_AND_WAIT or MC_RECEIVE_IMMEDIATE.

ABEND Conditions:

Parameter Check

RESOURCE specifies an unassigned resource ID.

State Check

The mapped conversation is not in confirm state.

Notes:

1. The program can issue this verb only as a reply to a confirmation request; the verb cannot be issued at any other time.

2. The program may use this verb for various application-level functions. For example, the remote program may send some data records followed by a confirmation request. When the local program receives the confirmation request, it may issue this verb as an indication that it received and processed the data records without error.

3. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

MC_DEALLOCATE

Deallocates the specified mapped conversation from the transaction program. The deallocation can be either completed as part of this verb, or deferred until the program issues an MC_FLUSH, MC_CONFIRM, or SYNCPT verb. When it is completed as part of this verb it can include the function of the MC_FLUSH or MC_CONFIRM verb. The resource ID becomes unassigned when deallocation is complete.

MC_DEALLOCATE	**Supplied Parameters:** RESOURCE (variable) [TYPE (SYNC_LEVEL) | (FLUSH) | (CONFIRM) | (ABEND) | (LOCAL)]
	Returned Parameters: RETURN_CODE (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID of the mapped conversation to be deallocated.

TYPE specifies the type of deallocation to be performed.

- **SYNC_LEVEL** specifies to perform deallocation based on the synchronization level allocated to this mapped conversation:
  - If SYNC_LEVEL(NONE), execute the function of the MC_FLUSH verb and deallocate the mapped conversation normally.
  - If SYNC_LEVEL(CONFIRM), execute the function of the MC_CONFIRM verb and if it is successful (as indicated by a return code of OK on this MC_DEALLOCATE verb), deallocate the mapped conversation normally; if it is not successful, the state of the mapped conversation is determined by the return code.
  - If SYNC_LEVEL(SYNCPT), defer the deallocation until the program issues a SYNCPT, or the program issues an MC_CONFIRM or MC_FLUSH for this mapped conversation. If the SYNCPT or MC_CONFIRM is successful (as indicated by a return code of OK on that verb) or MC_FLUSH is issued, the mapped conversation is then deallocated normally; otherwise, the state of the mapped conversation is determined by the return code.
- **FLUSH** specifies to execute the function of the MC_FLUSH verb and deallocate the mapped conversation normally.
- **CONFIRM** specifies to execute the function of the MC_CONFIRM verb and if it is successful (as indicated by a return code of OK on this MC_DEALLOCATE verb), deallocate the mapped conversation normally; if it is not successful, the state of the mapped conversation is determined by the return code.
- **ABEND** specifies to execute the function of the MC_FLUSH verb when the mapped conversation is in send or defer state, and deallocate the mapped conversation abnormally. Data purging can occur when the mapped conversation is in receive state.
- **LOCAL** specifies to deallocate the mapped conversation locally. This type of deallocation must be specified if, and only if, the mapped conversation is in deallocate state. Deallocate state is

entered when the program receives on a previously issued verb a return code indicating the mapped conversation has been deallocated (see "Return Codes" on page 4-99).

The execution of the MC_FLUSH or MC_CONFIRM function as part of this verb includes the flushing of the LU's send buffer. When, instead, the deallocation is deferred, the LU also defers flushing its send buffer until the program issues a subsequent verb for this mapped conversation.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the local program. The return code indicates the result of verb execution. The TYPE parameter determines which of the following return codes can be returned to the program.

- If TYPE(SYNC_LEVEL) is specified and the synchronization level allocated to this mapped conversation is NONE, or TYPE(FLUSH), TYPE(ABEND), or TYPE(LOCAL) is specified, the following return code is returned:
  - OK (deallocation is complete)
- If TYPE(SYNC_LEVEL) is specified and the synchronization level allocated to this mapped conversation is CONFIRM, or TYPE(CONFIRM) is specified, one of the following return codes is returned:
  - OK (deallocation is complete)
  - ALLOCATION_ERROR
  - DEALLOCATE_ABEND
  - FMH_DATA_NOT_SUPPORTED
  - MAPPING_NOT_SUPPORTED
  - MAP_NOT_FOUND
  - MAP_EXECUTION_FAILURE
  - PROG_ERROR_PURGING
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY
- If TYPE(SYNC_LEVEL) is specified and the synchronization level allocated to this mapped conversation is SYNCPT, the following return code is returned:
  - OK (deallocation is deferred)

State Changes (when RETURN CODE indicates OK):

Defer state is entered when TYPE(SYNC_LEVEL) is specified and the synchronization level is SYNCPT.

Reset state is entered when TYPE(FLUSH), TYPE(CONFIRM), TYPE(LOCAL), or TYPE(ABEND) is specified, or when TYPE(SYNC_LEVEL) is specified and the synchronization level is NONE or CONFIRM.

ABEND Conditions:

Parameter Check

- RESOURCE specifies an unassigned resource ID.
- TYPE(CONFIRM) is specified and the mapped conversation is allocated with SYNC_LEVEL(NONE).

State Check

- TYPE(FLUSH), TYPE(CONFIRM), or TYPE(SYNC_LEVEL) is specified and the mapped conversation is not in send state.
- TYPE(ABEND) is specified and the mapped conversation is not in send, defer, receive, confirm, or sync point state.
- TYPE(LOCAL) is specified and the mapped conversation is not in deallocate state.

Notes:

1. When the deallocation is deferred (see the TYPE parameter), the LU buffers the deallocation information to be sent to the remote LU until the local program issues a verb that causes the LU to flush its send buffer.

2. The TYPE(SYNC_LEVEL) parameter is intended to be used by the transaction program in order to deallocate the mapped conversation based on the synchronization level allocated to the mapped conversation.

   - If the synchronization level is NONE, the mapped conversation is unconditionally deallocated.
   - If the synchronization level is CONFIRM, the mapped conversation is deallocated when the remote program responds to the confirmation request by issuing MC_CONFIRMED. The mapped conversation remains allocated when the remote program responds to the confirmation request by issuing MC_SEND_ERROR.
   - If the synchronization level is SYNCPT, the mapped conversation is deallocated when the local program subsequently issues SYNCPT and all programs throughout the transaction, connected to conversations having the synchronization level of SYNCPT, respond to the sync point request by issuing SYNCPT. The mapped conversation remains allocated when the remote program responds to the sync point request by issuing MC_SEND_ERROR, or one or more programs respond by issuing BACKOUT.

3. The TYPE(FLUSH) parameter is intended to be used by the transaction program in order to unconditionally deallocate the mapped conversation regardless of its synchronization level. TYPE(FLUSH) is functionally equivalent to:

   - TYPE(SYNC_LEVEL) with a synchronization level of NONE.
   - TYPE(SYNC_LEVEL) with a synchronization level of SYNCPT, followed by the MC_FLUSH verb.

4. The TYPE(CONFIRM) parameter is intended to be used by the transaction program in order to conditionally deallocate the mapped conversation, depending on the remote program's response, when the synchronization level is CONFIRM or SYNCPT. TYPE(CONFIRM) is functionally equivalent to:

   - TYPE(SYNC_LEVEL) with a synchronization level of CONFIRM.
   - TYPE(SYNC_LEVEL) with a synchronization level of SYNCPT, followed by the MC_CONFIRM verb.

   The mapped conversation is deallocated when the remote program responds to the confirmation request by issuing MC_CONFIRMED. The mapped conversation remains allocated when the remote program responds to the confirmation request by issuing MC_SEND_ERROR.

5. The TYPE(ABEND) parameter is intended to be used by the transaction program in order to unconditionally deallocate the mapped conversation regardless of its synchronization level and its current state. Specifically, the parameter is intended to be used when the program detects an error condition that prevents further useful communications, that is, communications that would lead to successful completion of the transaction. The specific use and meaning of ABEND are program-defined.

6. The TYPE(LOCAL) parameter is intended to be used by the transaction program in order to complete the program's deallocation of the mapped conversation after receiving an indication that the mapped conversation has been deallocated from the session, an indication such as a DEALLOCATE_NORMAL or RESOURCE_FAILURE_RETRY return code.

7. The remote transaction program receives the deallocate notification by means of a return code or what-received indication, as follows:

   - DEALLOCATE_NORMAL return code: The local program specified either TYPE(FLUSH); TYPE(SYNC_LEVEL) and the synchronization level is NONE; or TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the local program subsequently issued MC_FLUSH.

   - CONFIRM_DEALLOCATE what-received indication: The local program specified either TYPE(CONFIRM); TYPE(SYNC_LEVEL) and the synchronization level is CONFIRM; or TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the local program subsequently issued MC_CONFIRM.

   - TAKE_SYNCPT_DEALLOCATE what-received indication: The local program specified TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the local program subsequently issued SYNCPT.

   - DEALLOCATE_ABEND return code: The local program specified TYPE(ABEND), with the following exception: If the remote program has issued MC_SEND_ERROR in receive state, a DEALLOCATE_NORMAL return code is reported instead of DEALLOCATE_ABEND.

8. MC_DEALLOCATE with TYPE(ABEND) resets or cancels posting. If posting is active and the mapped conversation has been posted, posting is reset. If posting is active and the mapped conversation has not been posted, posting is canceled (posting will not occur). See the MC_POST_ON_RECEIPT verb for more details about posting.

9. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

MC_FLUSH

Flushes the local LU's send buffer. The LU sends any information it has buffered to the remote LU. Information the LU buffers can come from MC_ALLOCATE, MC_DEALLOCATE, MC_SEND_DATA, MC_PREPARE_TO_RECEIVE, or MC_SEND_ERROR. Refer to the descriptions of these verbs for details of the information the LU buffers and when buffering occurs.

MC_FLUSH	Supplied Parameters: RESOURCE (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

State Changes:

Receive state is entered when the verb is issued in defer state following MC_PREPARE_TO_RECEIVE.

Reset state is entered when the verb is issued in defer state following MC_DEALLOCATE.

No state change occurs when the verb is issued in send state.

ABEND Conditions:

Parameter Check

- This verb is not supported.
- RESOURCE specifies an unassigned resource ID.

State Check

The mapped conversation is not in send or defer state.

Notes:

1. This verb is useful for optimization of processing between the local and remote programs. The LU normally buffers the data records from consecutive MC_SEND_DATAs until it has a sufficient amount for transmission. At that time it transmits the buffered data records. However, the local program can issue MC_FLUSH in order to cause the LU to transmit the buffered data records. In this way, the local program can minimize the delay in the remote program's processing of the data records.

2. This verb can be issued after MC_DEALLOCATE with TYPE(SYNC_LEVEL) when the synchronization level for the mapped conversation is SYNCPT. The effect to the remote program is the same as issuing MC_DEALLOCATE with TYPE(FLUSH). The mapped conversation is deallocated at the completion of the MC_FLUSH verb.

3. This verb can be issued after MC_PREPARE_TO_RECEIVE with TYPE(SYNC_LEVEL) when the synchronization level for the mapped conversation is SYNCPT. The effect to the remote program is the same as issuing MC_PREPARE_TO_RECEIVE with TYPE(FLUSH). The mapped conversation enters receive state at the completion of the MC_FLUSH verb.

4. The LU flushes its send buffer only when it has some information to transmit. If the LU has no information in its send buffer, nothing is transmitted to the remote LU.

5. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

MC_GET_ATTRIBUTES

Returns information pertaining to the specified mapped conversation.

MC_GET_ATTRIBUTES	Supplied Parameters: RESOURCE (variable)
	Returned Parameters: [OWN_FULLY_QUALIFIED_LU_NAME (variable)] [PARTNER_LU_NAME (variable)] [PARTNER_FULLY_QUALIFIED_LU_NAME (variable)] [MODE_NAME (variable)] [SYNC_LEVEL (variable)] [SECURITY_USER_ID (variable)] [SECURITY_PROFILE (variable)] [LUW_IDENTIFIER (variable)] [CONVERSATION_CORRELATOR (variable)]
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID of the mapped conversation of which the attributes are desired.

Returned Parameters:

OWN_FULLY_QUALIFIED_LU_NAME specifies the variable for returning the fully qualified name of the LU at which the local transaction program is located. If the local fully qualified LU name is not known, a null value is returned.

PARTNER_LU_NAME specifies the variable for returning the name of the LU at which the remote transaction program is located. This is a name by which the local LU knows the remote LU for the purpose of allocating a mapped conversation. Refer to the description of the LU_NAME parameter of MC_ALLOCATE for more details.

PARTNER_FULLY_QUALIFIED_LU_NAME specifies the variable for returning the fully qualified name of the LU at which the remote transaction program is located. If the partner's fully qualified LU name is not known, a null value is returned.

MODE_NAME specifies the variable for returning the mode name for the session on which the mapped conversation is allocated.

SYNC_LEVEL specifies the variable for returning the level of synchronization processing being used for the mapped conversation. The synchronization levels are:

- NONE
- CONFIRM

- SYNCPT

SECURITY_USER_ID specifies the variable for returning the user ID carried on the allocation request that initiated execution of the local program. A null value is returned if the allocation request did not contain a user ID.

SECURITY_PROFILE specifies the variable for returning the profile carried on the allocation request that initiated execution of the local program. A null value is returned if the allocation request did not contain a profile.

LUW_IDENTIFIER specifies the variable for returning the logical unit of work (LUW) identifier associated with the mapped conversation. The LUW identifier is created and maintained by the LU. The LU uses it to identify the most recent sync point and for accounting purposes. If no LUW identifier is used on the mapped conversation, a null value is returned.

CONVERSATION_CORRELATOR specifies the variable for returning the conversation correlator. The conversation correlator is created and maintained by the LU. The LU uses it during sync point resynchronization. If no conversation correlator is used on the mapped conversation, a null value is returned.

State Changes:

None

ABEND Conditions:

Parameter Check

- This verb is not supported.
- RESOURCE specifies an unassigned resource ID.
- SECURITY_USER_ID is specified and not supported.
- SECURITY_PROFILE is specified and not supported.
- LUW_IDENTIFIER is specified and not supported.
- CONVERSATION_CORRELATOR is specified and not supported.

State Check

None

Notes:

1. The program may issue this verb in order to obtain the attributes of the mapped conversation, including the one by which the program was started.

2. Specifying SECURITY_USER_ID or SECURITY_PROFILE returns the user ID or profile carried on the allocation request that initiated execution of the local program, regardless of which resource ID is supplied on the RESOURCE parameter.

3. The LU creates the LUW identifier for its use during sync point processing, and for accounting purposes. For sync point, the LUW identifier uniquely identifies the most recent synchronization point.

4. The LU creates the conversation correlator for its use during sync point resynchronization. For sync point resynchronization, the conversation correlator correlates the logical unit of work to the sync point states associated with the current instance of the local program.

MC_POST_ON_RECEIPT

Causes the LU to post the specified mapped conversation when information is available for the program to receive. The information can be data, mapped conversation status, or a request for confirmation or sync point. WAIT should be issued after MC_POST_ON_RECEIPT in order to wait for posting to occur. Alternatively, MC_TEST may be issued following MC_POST_ON_RECEIPT in order to determine when posting has occurred.

MC_POST_ON_RECEIPT	Supplied Parameters: RESOURCE (variable) LENGTH (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

LENGTH specifies the variable containing a length value, which is the maximum length data record that the program can receive. This parameter is used to determine when to post the mapped conversation for the receipt of a data record.

State Changes:

None

ABEND Conditions:

Parameter Check

- This verb is not supported.
- RESOURCE specifies an unassigned resource ID.

State Check

The mapped conversation is not in receive state.

Notes:

1. This verb is intended to be used in conjunction with MC_TEST or WAIT. The use of this verb and WAIT allows a program to perform synchronous receiving from multiple mapped conversations, where the program issues this verb for each of the mapped conversations and then issues WAIT (for each mapped conversation) to wait until information is available to be received on the mapped conversations. The use of this verb and MC_TEST allows a program to continue its processing and test the mapped conversations to determine when information is available to be received.

2. Posting occurs when the LU has any information that the program can receive, such as a data record, mapped conversation status, or a request for confirmation or sync point. Refer to the MC_RECEIVE_AND_WAIT verb for a description of the types of information a program can receive.

3. Posting is active for a mapped conversation when MC_POST_ON_RECEIPT has been issued for the mapped conversation and posting has not yet been reset or cancelled.

   Posting is reset when any of the following verbs is issued for the same mapped conversation as specified on MC_POST_ON_RECEIPT after the mapped conversation is posted:

   BACKOUT
   MC_DEALLOCATE with TYPE(ABEND)
   MC_RECEIVE_AND_WAIT
   MC_RECEIVE_IMMEDIATE

MC_SEND_ERROR
MC_TEST
WAIT

Posting is cancelled when any of the following verbs is issued for the same mapped conversation as specified on MC_POST_ON_RECEIPT before the mapped conversation is posted:

BACKOUT
MC_DEALLOCATE with TYPE(ABEND)
MC_RECEIVE_IMMEDIATE
MC_SEND_ERROR

In order for the program to activate posting again after posting has been reset or cancelled, the program issues another MC_POST_ON_RECEIPT.

4. Any number of MC_POST_ON_RECEIPTs may be issued for a given mapped conversation before posting is reset or cancelled. The last MC_POST_ON_RECEIPT issued for a mapped conversation is the one that determines when posting will occur for data. For example, if a program issues MC_POST_ON_RECEIPT with LENGTH(1000) in preparation to receive a 1000 byte data record, and then issues the verb again with LENGTH(500), posting will occur when 500 bytes of the data record are available.

5. MC_POST_ON_RECEIPT with LENGTH(0) has no special significance. It specifies that posting for a data record is to occur upon receipt of any amount of the data record of one byte or more. It is equivalent to MC_POST_ON_RECEIPT with LENGTH(1).

6. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

MC_PREPARE_TO_RECEIVE

Changes the mapped conversation from send to receive state in preparation to receive data. The change to receive state can be either completed as part of this verb, or deferred until the program issues an MC_FLUSH, MC_CONFIRM, or SYNCPT verb. When it is completed as part of this verb it includes the function of the MC_FLUSH or MC_CONFIRM verb.

MC_PREPARE_TO_RECEIVE	**Supplied Parameters:** RESOURCE (variable) [TYPE (SYNC_LEVEL) (FLUSH) (CONFIRM)] [LOCKS (SHORT) (LONG)]
	Returned Parameters: RETURN_CODE (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

TYPE specifies the type of prepare-to-receive to be performed for this mapped conversation.

- **SYNC_LEVEL** specifies to perform the prepare-to-receive based on the synchronization level allocated to this mapped conversation:
  - If SYNC_LEVEL(NONE), execute the function of the MC_FLUSH verb and enter receive state.
  - If SYNC_LEVEL(CONFIRM), execute the function of the MC_CONFIRM verb and if it is successful (as indicated by a return code of OK on this MC_PREPARE_TO_RECEIVE verb), enter receive state; if it is not successful, the state of the mapped conversation is determined by the return code.
  - If SYNC_LEVEL(SYNCPT), enter defer state until the program issues a SYNCPT, or the program issues an MC_CONFIRM or MC_FLUSH for this mapped conversation. If the SYNCPT or MC_CONFIRM is successful (as indicated by a return code of OK on that verb) or MC_FLUSH is issued, receive state is then entered for this mapped conversation; otherwise, the state of the mapped conversation is determined by the return code.
- **FLUSH** specifies to execute the function of the MC_FLUSH verb and enter receive state.
- **CONFIRM** specifies to execute the function of the MC_CONFIRM verb and, if it is successful (as indicated by a return code of OK on this MC_PREPARE_TO_RECEIVE verb), enter receive state; if it is not successful, the state of the mapped conversation is determined by the return code.

The execution of the MC_FLUSH or MC_CONFIRM function as part of this verb includes the flushing of the LU's send buffer. When, instead, defer state is entered, the LU defers flushing its send buffer until the program issues a subsequent verb for this mapped conversation.

LOCKS specifies when control is to be returned to the local program following execution of the CONFIRM function of this verb or following execution of an MC_CONFIRM verb issued subsequent to this verb. This

parameter is significant only when TYPE(CONFIRM) is also specified or when TYPE(SYNC_LEVEL) is also specified and the synchronization level for this mapped conversation is CONFIRM; or when TYPE(SYNC_LEVEL) is also specified, the synchronization level for this mapped conversation is SYNCPT, and a subsequent MC_CONFIRM is issued. Otherwise, this parameter has no meaning and is ignored.

- **SHORT** specifies to return control when an affirmative reply is received, as follows:
  - When the synchronization level is CONFIRM, return control from execution of this verb when an MC_CONFIRMED reply is received.
  - When the synchronization level is SYNCPT, return control immediately from execution of this verb; return control from execution of a subsequent MC_CONFIRM verb when a corresponding MC_CONFIRMED reply is received.
- **LONG** specifies to return control when information, such as data, is received from the remote program following an affirmative reply, as follows:
  - When the synchronization level is CONFIRM, return control from execution of this verb when information is received following an MC_CONFIRMED reply.
  - When the synchronization level is SYNCPT, return control immediately from execution of this verb; return control from execution of a subsequent MC_CONFIRM verb when information is received following a corresponding MC_CONFIRMED reply.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the local program. The return code indicates the result of verb execution. The TYPE parameter determines which of the following return codes can be returned to the program.

- If TYPE(FLUSH) is specified, or if TYPE(SYNC_LEVEL) is specified and the synchronization level allocated to this mapped conversation is NONE, the following return code is returned:
  - OK
- If TYPE(SYNC_LEVEL) is specified and the synchronization level allocated to this mapped conversation is CONFIRM, or TYPE(CONFIRM) is specified, one of the following return codes is returned:
  - OK
  - ALLOCATION_ERROR
  - DEALLOCATE_ABEND
  - FMH_DATA_NOT_SUPPORTED
  - MAPPING_NOT_SUPPORTED
  - MAP_NOT_FOUND
  - MAP_EXECUTION_FAILURE
  - PROG_ERROR_PURGING
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY
- If TYPE(SYNC_LEVEL) is specified and the synchronization level allocated to this mapped conversation is SYNCPT, the following return code is returned:
  - OK

State Changes (when RETURN_CODE indicates OK):

Defer state is entered when TYPE(SYNC_LEVEL) is specified and the synchronization level is SYNCPT.

Receive state is entered when TYPE(FLUSH) or TYPE(CONFIRM) is specified, or when TYPE(SYNC_LEVEL) is specified and the synchronization level is NONE or CONFIRM.

ABEND Conditions:

Parameter Check

- This verb is not supported.
- RESOURCE specifies an unassigned resource ID.
- TYPE(CONFIRM) is specified and the conversation is allocated with SYNC_LEVEL(NONE).
- LOCKS(LONG) is specified and not supported.

State Check

The mapped conversation is not in send state.

Notes:

1. The TYPE(SYNC_LEVEL) parameter is intended to be used by the transaction program in order to transfer send control to the remote program based on the synchronization level allocated to the mapped conversation.

   - If the synchronization level is NONE, send control is transferred to the remote program without any synchronizing acknowledgment.
   - If the synchronization level is CONFIRM, send control is transferred to the remote program with confirmation requested.
   - If the synchronization level is SYNCPT, transfer of send control is deferred. When the local program subsequently issues SYNCPT, send control is transferred to the remote program with sync point requested.

2. The TYPE(FLUSH) parameter is intended to be used by the transaction program in order to transfer send control to the remote program without any synchronizing acknowledgment. TYPE(FLUSH) is functionally equivalent to:

   - TYPE(SYNC_LEVEL) with a synchronization level of NONE.
   - TYPE(SYNC_LEVEL) with a synchronization level of SYNCPT, followed by the MC_FLUSH verb.

3. The TYPE(CONFIRM) parameter is intended to be used by the transaction program in order to transfer send control to the remote program with confirmation requested. TYPE(CONFIRM) is functionally equivalent to:

   - TYPE(SYNC_LEVEL) with a synchronization level of CONFIRM.
   - TYPE(SYNC_LEVEL) with a synchronization level of SYNCPT, followed by the MC_CONFIRM verb.

4. The remote transaction program receives send control by means of a what-received indication of SEND, CONFIRM_SEND, or TAKE_SYNCPT_SEND, as follows:

   - SEND: The local program specified either TYPE(FLUSH); TYPE(SYNC_LEVEL) and the synchronization level is NONE; or TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the local program subsequently issued MC_FLUSH.
   - CONFIRM_SEND: The local program specified either TYPE(CONFIRM); TYPE(SYNC_LEVEL) and the synchronization level is CONFIRM; or TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the local program subsequently issued MC_CONFIRM.

- TAKE_SYNCPT_SEND: The local program specified TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the local program subsequently issued SYNCPT.

5. If TYPE(SYNC_LEVEL) is specified and the synchronization level for the mapped conversation is SYNCPT, the LU buffers the SEND notification to be sent to the remote LU until the local program issues a verb that causes the LU to flush its send buffer.

6. The mapped conversation for the remote program enters the corresponding send state when it issues an MC_RECEIVE_AND_WAIT or MC_RECEIVE_IMMEDIATE verb and receives the SEND indication (on the WHAT_RECEIVED parameter). The remote program can then send data to the local program.

7. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

MC_RECEIVE_AND_WAIT

Waits for information to arrive on the specified mapped conversation and then receives the information. If information is already available, the program receives it without waiting. The information can be a data record, mapped conversation status, or a request for confirmation or sync point. Control is returned to the program with an indication of the type of information.

The program can issue this verb when the mapped 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 mapped 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.

MC_RECEIVE_AND_WAIT	Supplied Parameters: RESOURCE (variable)
	Supplied-and-Returned Parameters: LENGTH (variable)
	Returned Parameters: RETURN_CODE (variable) REQUEST_TO_SEND_RECEIVED (variable) DATA (variable) WHAT_RECEIVED (variable) [MAP_NAME (variable)]
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

Supplied-and-Returned Parameters:

LENGTH specifies the variable containing a length value that is the maximum amount of the data record the program is to receive. When control is returned to the program this variable contains the actual amount of the data record the program received, up to the maximum. If the program receives information other than data, this variable remains unchanged.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the program. The return code indicates the result of verb execution. The return codes that can be returned depend on the state of the mapped conversation at the time this verb is issued.

- If this verb is issued in send state, the following return codes can be returned:
  - OK
  - ALLOCATION_ERROR
  - BACKED_OUT
  - DEALLOCATE_ABEND
  - FMH_DATA_NOT_SUPPORTED
  - MAPPING_NOT_SUPPORTED
  - MAP_NOT_FOUND
  - MAP_EXECUTION_FAILURE
  - PROG_ERROR_PURGING

  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY

- If this verb is issued in receive state, the following return codes can be returned:

  - OK
  - ALLOCATION_ERROR
  - BACKED_OUT
  - DEALLOCATE_ABEND
  - DEALLOCATE_NORMAL
  - FMH_DATA_NOT_SUPPORTED
  - MAPPING_NOT_SUPPORTED
  - MAP_NOT_FOUND
  - MAP_EXECUTION_FAILURE
  - PROG_ERROR_NO_TRUNC
  - PROG_ERROR_PURGING
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received. The indication is either YES or NO.

- YES indicates a REQUEST_TO_SEND notification has been received from the remote transaction program. The remote program has issued MC_REQUEST_TO_SEND, requesting the local program to enter receive state and thereby place the remote program in send state

- NO indicates a REQUEST_TO_SEND notification has not been received.

DATA specifies the variable in which the program is to receive the data. When the program receives information other than data, as indicated by the WHAT_RECEIVED parameter, nothing is placed in this variable.

WHAT_RECEIVED specifies the variable in which is returned an indication of what the transaction program receives. The program should examine this variable only when RETURN_CODE indicates OK; otherwise, nothing is placed in this variable.

- DATA_COMPLETE indicates the program received a complete data record or the last remaining portion of the record.

- DATA_TRUNCATED indicates the program received less than a complete data record, and the LU discarded the remainder of the data record.

- DATA_INCOMPLETE indicates the program received less than a complete data record, and the LU retained the remainder of the data record. The program may receive the remainder of the data record by issuing another MC_RECEIVE_AND_WAIT (or possibly multiple MC_RECEIVE_AND_WAITs).

- FMH_DATA_COMPLETE indicates the program received a complete data record or the last remaining portion of the record, and the data record contains FM headers.

- FMH_DATA_TRUNCATED indicates the program received less than a complete data record containing FM headers, and the LU discarded the remainder of the data record.

- FMH_DATA_INCOMPLETE indicates the program received less than a complete data record containing FM headers, and the LU retained the remainder of the data record. The program may receive the remainder of the data record by issuing another MC_RECEIVE_AND_WAIT (or possibly multiple MC_RECEIVE_AND_WAITs).

- SEND indicates the remote program has entered receive state, placing the local program in send state. The local program may now issue MC_SEND_DATA.

- CONFIRM indicates the remote program has issued MC_CONFIRM, requesting the local program to respond by issuing MC_CONFIRMED. The program may respond, instead, by issuing a verb other than MC_CONFIRMED, such as MC_SEND_ERROR.

- CONFIRM_SEND indicates the remote program has issued MC_PREPARE_TO_RECEIVE with TYPE(CONFIRM); or with TYPE(SYNC_LEVEL), and either the synchronization level is CONFIRM, or it is SYNCPT and the remote program subsequently issued MC_CONFIRM. The local program may respond by issuing MC_CONFIRMED, or by issuing another verb such as MC_SEND_ERROR.

- CONFIRM_DEALLOCATE indicates the remote program has issued MC_DEALLOCATE with TYPE(CONFIRM); or with TYPE(SYNC_LEVEL), and either the synchronization level is CONFIRM, or it is SYNCPT and the remote program subsequently issued MC_CONFIRM. The local program may respond by issuing MC_CONFIRMED, or by issuing another verb such as MC_SEND_ERROR.

- TAKE_SYNCPT indicates the remote program has issued SYNCPT, requesting the local program to respond by issuing SYNCPT in order to perform the sync-point function on all protected resources throughout the transaction. Issuing the SYNCPT verb also causes an affirmative reply to be returned to the remote program if the sync-point function is successful. The program may respond, instead, by issuing a verb other than SYNCPT, such as BACKOUT or MC_SEND_ERROR.

- TAKE_SYNCPT_SEND indicates the remote program has issued MC_PREPARE_TO_RECEIVE with TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the remote program subsequently issued SYNCPT. The local program may respond by issuing SYNCPT, or by issuing another verb such as BACKOUT or MC_SEND_ERROR.

- TAKE_SYNCPT_DEALLOCATE indicates the remote program has issued MC_DEALLOCATE with TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the remote program subsequently issued SYNCPT. The local program may respond by issuing SYNCPT, or by issuing another verb such as BACKOUT or MC_SEND_ERROR.

MAP_NAME specifies the variable in which is returned the name of the format (such as the name of a DSECT or DECLARE) that defines the structure of the data record. A null value returned means the data record has not been mapped. That is, mapping of this data record is suppressed.

When the program receives information other than data, as indicated by the WHAT_RECEIVED parameter, nothing is placed in this variable.

State Changes (When RETURN CODE indicates OK):

Receive state is entered when the verb is issued in send state and WHAT_RECEIVED indicates DATA_COMPLETE, DATA_INCOMPLETE, FMH_DATA_COMPLETE, or FMH_DATA_INCOMPLETE.

Send state is entered when WHAT_RECEIVED indicates SEND.

Confirm state is entered when WHAT_RECEIVED indicates CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE.

Sync-point state is entered when WHAT_RECEIVED indicates TAKE_SYNCPT, TAKE_SYNCPT_SEND, or TAKE_SYNCPT_DEALLOCATE.

No state change occurs when the verb is issued in receive state and WHAT_RECEIVED indicates DATA_COMPLETE, DATA_INCOMPLETE, FMH_DATA_COMPLETE, or FMH_DATA_INCOMPLETE.

ABEND Conditions:

Parameter Check

- RESOURCE specifies an unassigned resource ID.
- MAP_NAME is specified and not supported.

State Check

The mapped conversation is not in send or receive state.

Notes:

1. When the program issues MC_RECEIVE_AND_WAIT in send state, the LU implicitly executes an MC_PREPARE_TO_RECEIVE with TYPE(FLUSH) before executing the MC_RECEIVE_AND_WAIT. Refer to the description of MC_PREPARE_TO_RECEIVE for details of its function.

2. The mapped conversation protocol boundary provides for the sending and receiving of data records. Unlike the logical records defined for the basic conversation protocol boundary, data records contain only data; they do not contain the logical record length field.

3. The MC_RECEIVE_AND_WAIT verb can receive only as much of the data record as specified by the LENGTH parameter. The WHAT_RECEIVED parameter indicates whether the program has received a complete or incomplete data record, as follows:

   - The WHAT_RECEIVED parameter indicates DATA_COMPLETE or FMH_DATA_COMPLETE when the program receives a complete data record or the last remaining portion of a data record. The length of the record or portion of the record is equal to or less than the length specified on the LENGTH parameter.

   - The WHAT_RECEIVED parameter indicates DATA_TRUNCATED, DATA_INCOMPLETE, FMH_DATA_TRUNCATED, or FMH_DATA_INCOMPLETE when the program receives a portion of a data record other than the last remaining portion. The data record is incomplete because the length of the record is greater than the length specified on the LENGTH parameter; the amount received equals the length specified.

4. Whether the LU discards or retains the remainder of an incompletely received data record depends on the product and the data-record format indicated by the format name returned on the MAP_NAME parameter. A product may imply by some or all of its format names (including the null value) that the remaining data is discarded, rather than retained.

5. MC_RECEIVE_AND_WAIT with LENGTH(0) has no special significance. The type of information available is indicated by the RETURN_CODE and WHAT_RECEIVED parameters, as usual. However, the program receives no data.

6. The program receives only one kind of information at a time. For example, it may receive data or a CONFIRM request, but it does not receive both at the same time. The RETURN_CODE and WHAT_RECEIVED parameters indicate to the program the kind of information the program receives.

7. MC_RECEIVE_AND_WAIT includes posting. If posting is already active when this verb is issued, this verb supersedes the prior MC_POST_ON_RECEIPT function. Posting is reset at the completion of this verb. See the MC_POST_ON_RECEIPT verb for more details about posting.

8. It is the responsibility of both sending and receiving installations to maintain the map-name definitions referenced by their application transaction programs.

9. The function of FM headers in the data record is significant only to the transaction programs; the sending and receiving LUs perform no FM-header related processing other than indicating that the data record contains FM headers. The presence of FM headers in the data record is specified by the remote transaction program by means of the FMH_DATA parameter of the MC_SEND_DATA that sent the data record.

10. The REQUEST_TO_SEND notification is usually received when the local transaction program is in send state, and reported to the

program on an MC_SEND_DATA verb or on an MC_SEND_ERROR verb issued in send state. However, the notification can be received when the program is in receive state under the following conditions:

- When the local program just entered receive state and the remote program issued MC_REQUEST_TO_SEND before it entered send state.
- When the remote program has just entered receive state by means of the MC_PREPARE_TO_RECEIVE verb (not MC_RECEIVE_AND_WAIT), and then issued MC_REQUEST_TO_SEND before the local program enters send state. This can occur because the REQUEST_TO_SEND is transmitted as an expedited request and can therefore arrive ahead of the request carrying the SEND indication. Potentially, the local program cannot distinguish this case from the first. This ambiguity is avoided when the remote program waits until it receives information from the local program before it issues the MC_REQUEST_TO_SEND.
- When the remote program issues the MC_REQUEST_TO_SEND in send state (see "Notes on Implementation Details" in Appendix A).

11. The REQUEST_TO_SEND notification is returned to the program in addition to (not in place of) the information indicated by the RETURN_CODE and WHAT_RECEIVED parameters.
12. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

MC_RECEIVE_IMMEDIATE

Receives any information that is available from the specified mapped conversation, but does not wait for information to arrive. The information (if any) can be data, mapped conversation status, or a request for confirmation or sync point. Control is returned to the program with an indication of whether any information was received and, if so, the type of information.

MC_RECEIVE_IMMEDIATE	Supplied Parameters: RESOURCE (variable)
	Supplied-and-Returned Parameters: LENGTH (variable)
	Returned Parameters: RETURN_CODE (variable) REQUEST_TO_SEND_RECEIVED (variable) DATA (variable) WHAT_RECEIVED (variable) [MAP_NAME (variable)]
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

Supplied-and-Returned Parameters:

LENGTH specifies the variable containing a length value that is the maximum amount of the data record the program is to receive. When control is returned to the program this variable contains the actual amount of the data record the program received, up to the maximum. If the program receives information other than data, or no information at all, this variable remains unchanged.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the program. The return code indicates the result of verb execution.

- OK
- ALLOCATION_ERROR
- BACKED_OUT
- DEALLOCATE_ABEND
- DEALLOCATE_NORMAL
- FMH_DATA_NOT_SUPPORTED
- MAPPING_NOT_SUPPORTED
- MAP_NOT_FOUND
- MAP_EXECUTION_FAILURE
- PROG_ERROR_NO_TRUNC
- PROG_ERROR_PURGING
- RESOURCE_FAILURE_NO_RETRY
- RESOURCE_FAILURE_RETRY
- UNSUCCESSFUL - There is nothing to receive.

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received. The indication is either YES or NO.

- YES indicates a REQUEST_TO_SEND notification has been received from the remote transaction program. The remote program has issued MC_REQUEST_TO_SEND, requesting the local program to enter receive state and thereby place the remote program in send state

- NO indicates a REQUEST_TO_SEND notification has not been received.

DATA specifies the variable in which the program is to receive the data. When the program receives information other than data, as indicated by the WHAT_RECEIVED parameter, nothing is placed in this variable.

WHAT_RECEIVED specifies the variable in which is returned an indication of what the transaction program received. The program should examine this variable only when RETURN_CODE indicates OK; otherwise, nothing is placed in this variable.

- DATA_COMPLETE indicates the program received a complete data record or the last remaining portion of the record.

- DATA_TRUNCATED indicates the program received less than a complete data record, and the LU discarded the remainder of the data record.

- DATA_INCOMPLETE indicates the program received less than a complete data record, and the LU retained the remainder of the data record. The program may receive the remainder of the data record by issuing another MC_RECEIVE_IMMEDIATE (or possibly multiple MC_RECEIVE_IMMEDIATEs).

- FMH_DATA_COMPLETE indicates the program received a complete data record or the last remaining portion of the record, and the data record contains FM headers.

- FMH_DATA_TRUNCATED indicates the program received less than a complete data record containing FM headers, and the LU discarded the remainder of the data record.

- FMH_DATA_INCOMPLETE indicates the program received less than a complete data record containing FM headers, and the LU retained the remainder of the data record. The program may receive the remainder of the data record by issuing another MC_RECEIVE_IMMEDIATE (or possibly multiple MC_RECEIVE_IMMEDIATEs).

- SEND indicates the remote program has entered receive state, placing the local program in send state. The local program may now issue MC_SEND_DATA.

- CONFIRM indicates the remote program has issued MC_CONFIRM, requesting the local program to respond by issuing MC_CONFIRMED. The program may respond, instead, by issuing a verb other than MC_CONFIRMED, such as MC_SEND_ERROR.

- CONFIRM_SEND indicates the remote program has issued MC_PREPARE_TO_RECEIVE with TYPE(CONFIRM); or with TYPE(SYNC_LEVEL), and either the synchronization level is CONFIRM, or it is SYNCPT and the remote program subsequently issued MC_CONFIRM. The local program may respond by issuing MC_CONFIRMED, or by issuing another verb such as MC_SEND_ERROR.

- CONFIRM_DEALLOCATE indicates the remote program has issued MC_DEALLOCATE with TYPE(CONFIRM); or with TYPE(SYNC_LEVEL), and either the synchronization level is CONFIRM, or it is SYNCPT and the remote program subsequently issued MC_CONFIRM. The local program may respond by issuing MC_CONFIRMED, or by issuing another verb such as MC_SEND_ERROR.

- TAKE_SYNCPT indicates the remote program has issued SYNCPT, requesting the local program to respond by issuing SYNCPT in order to perform the sync-point function on all protected resources throughout the transaction. Issuing the SYNCPT verb also causes an affirmative reply to be returned to the remote program if the

sync-point function is successful. The program may respond, instead, by issuing a verb other than SYNCPT, such as BACKOUT or MC_SEND_ERROR.

- TAKE_SYNCPT_SEND indicates the remote program has issued MC_PREPARE_TO_RECEIVE with TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the remote program subsequently issued SYNCPT. The local program may respond by issuing SYNCPT, or by issuing another verb such as BACKOUT or MC_SEND_ERROR.

- TAKE_SYNCPT_DEALLOCATE indicates the remote program has issued MC_DEALLOCATE with TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the remote program subsequently issued SYNCPT. The local program may respond by issuing SYNCPT, or by issuing another verb such as BACKOUT or MC_SEND_ERROR.

MAP_NAME specifies the variable in which is returned the name of the format (such as the name of a DSECT or DECLARE) that defines the structure of the data record. A null value returned means the data record has not been mapped. That is, mapping of this data record is suppressed.

When the program receives information other than data, as indicated by the WHAT_RECEIVED parameter, nothing is placed in this variable.

State Changes (When RETURN_CODE indicates OK):

Send state is entered when WHAT_RECEIVED indicates SEND.

Confirm state is entered when WHAT_RECEIVED indicates CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE.

Sync-point state is entered when WHAT_RECEIVED indicates TAKE_SYNCPT, TAKE_SYNCPT_SEND, or TAKE_SYNCPT_DEALLOCATE.

No state change occurs when WHAT_RECEIVED indicates DATA_COMPLETE, DATA_INCOMPLETE, FMH_DATA_COMPLETE, or FMH_DATA_INCOMPLETE.

ABEND Conditions:

Parameter Check

- This verb is not supported.
- RESOURCE specifies an unassigned resource ID.
- MAP_NAME is specified and not supported.

State Check

The mapped conversation is not in receive state.

Notes:

1. The mapped conversation protocol boundary provides for the sending and receiving of data records. Unlike the logical records defined for the basic conversation protocol boundary, data records contain only data; they do not contain the logical record length field.

2. The MC_RECEIVE_IMMEDIATE verb can receive only as much of the data record as specified by the LENGTH parameter. The WHAT_RECEIVED parameter indicates whether the program has received a complete or incomplete data record, as follows:

   - The WHAT_RECEIVED parameter indicates DATA_COMPLETE or FMH_DATA_COMPLETE when the program receives a complete data record or the last remaining portion of a data record. The length of the record or portion of the record is equal to or less than the length specified on the LENGTH parameter.

   - The WHAT_RECEIVED parameter indicates DATA_TRUNCATED, DATA_INCOMPLETE, FMH_DATA_TRUNCATED, or FMH_DATA_INCOMPLETE when the program receives a portion of a data record other than the last remaining portion. The data record is incomplete because:

- The length of the record is greater than the length specified on the LENGTH parameter; in this case the amount received equals the length specified.

- Only a portion of the data record is available, the portion being equal to or less than the length specified on the LENGTH parameter.

3. Whether the LU discards or retains the remainder of an incompletely received data record depends on the product and the data-record format indicated by the format name returned on the MAP_NAME parameter. A product may imply by some or all of its format names (including the null value) that the remaining data is discarded, rather than retained.

4. MC_RECEIVE_IMMEDIATE with LENGTH(0) has no special significance. The type of information available, if any, is indicated by the RETURN_CODE and WHAT_RECEIVED parameters, as usual. However, the program receives no data.

5. The program receives only one kind of information at a time. For example, it may receive data or a CONFIRM request, but it does not receive both at the same time. The RETURN_CODE and WHAT_RECEIVED parameters indicate to the program the kind of information the program receives, if any.

6. MC_RECEIVE_IMMEDIATE resets or cancels posting. If posting is active and the mapped conversation has been posted, posting is reset. If posting is active and the mapped conversation has not been posted, posting is cancelled (posting will not occur). See the MC_POST_ON_RECEIPT verb for more details about posting.

7. It is the responsibility of both sending and receiving installations to maintain the map-name definitions referenced by their application transaction programs.

8. The function of FM headers in the data record is significant only to the transaction programs; the sending and receiving LUs perform no FM-header related processing other than indicating that the data record contains FM headers. The presence of FM headers in the data record is specified by the remote transaction program by means of the FMH_DATA parameter of the MC_SEND_DATA that sent the data record.

9. The REQUEST_TO_SEND notification is usually received when the local transaction program is in send state, and reported to the program on an MC_SEND_DATA verb or on an MC_SEND_ERROR verb issued in send state. However, the notification can be received when the program is in receive state under the following conditions:

   - When the local program just entered receive state and the remote program issued MC_REQUEST_TO_SEND before it entered send state.

   - When the remote program has just entered receive state by means of the MC_PREPARE_TO_RECEIVE verb (not MC_RECEIVE_AND_WAIT), and then issued MC_REQUEST_TO_SEND before the local program enters send state. This can occur because the MC_REQUEST_TO_SEND is transmitted as an expedited request and can therefore arrive ahead of the request carrying the SEND indication. Potentially, the local program cannot distinguish this case from the first. This ambiguity is avoided when the remote program waits until it receives information from the local program before it issues the MC_REQUEST_TO_SEND.

   - When the remote program issues the MC_REQUEST_TO_SEND in send state (see "Notes on Implementation Details" in Appendix A).

10. The REQUEST_TO_SEND notification is returned to the program in addition to (not in place of) the information indicated by the RETURN_CODE and WHAT_RECEIVED parameters.

11. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

MC_REQUEST_TO_SEND

Notifies the remote program that the local program is requesting to enter send state for the mapped conversation. The mapped conversation will be changed to send state when the local program subsequently receives a SEND indication from the remote program.

MC_REQUEST_TO_SEND	Supplied Parameters: RESOURCE (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

State Changes:

None

ABEND Conditions:

Parameter Check

RESOURCE specifies an unassigned resource ID.

State Check

- The mapped conversation is not in receive, confirm, or sync-point state.

Notes:

1. The REQUEST_TO_SEND notification is indicated to the remote program by means of the REQUEST_TO_SEND_RECEIVED parameter. When the REQUEST_TO_SEND_RECEIVED parameter is set to YES, the remote program is requested to enter receive state and thereby place the local program in send state. A program enters receive state by means of the MC_RECEIVE_AND_WAIT or MC_PREPARE_TO_RECEIVE verb. The partner program enters the corresponding send state when it issues an MC_RECEIVE_AND_WAIT or MC_RECEIVE_IMMEDIATE verb and receives the SEND indication (on the WHAT_RECEIVED parameter).

2. The REQUEST_TO_SEND_RECEIVED indication of YES is normally returned to the remote program when it is in send state, that is, on an MC_SEND_DATA or on an MC_SEND_ERROR issued in send state. However, it can be returned on an MC_RECEIVE_AND_WAIT or MC_RECEIVE_IMMEDIATE verb; see the description of MC_RECEIVE_AND_WAIT or MC_RECEIVE_IMMEDIATE for details about when this can occur.

3. When the remote LU receives the REQUEST_TO_SEND notification, it retains the notification until the remote program issues a verb on which the notification can be indicated, that is, a verb with the REQUEST_TO_SEND_RECEIVED parameter. The remote LU will retain only one REQUEST_TO_SEND notification at a time (per mapped conversation); additional notifications are discarded until the retained notification is indicated to the remote program. It is therefore possible for the local program to issue the MC_REQUEST_TO_SEND verb more times than are indicated to the remote program.

4. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

MC_SEND_DATA

Sends one data record to the remote transaction program. The data record consists entirely of data. The program can specify data mapping as a function of this verb, and it can indicate whether the data record includes FM headers.

MC_SEND_DATA	**Supplied Parameters:** RESOURCE (variable) DATA (variable) LENGTH (variable) [MAP_NAME (NO) (YES (variable))] [FMH_DATA (NO) (YES)]
	Returned Parameters: RETURN_CODE (variable) REQUEST_TO_SEND_RECEIVED (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID of the mapped conversation on which the data record is to be sent.

DATA specifies the variable containing the data record to be sent. The data record consists entirely of data.[1] The length of the data record is given by the LENGTH parameter.

LENGTH specifies the variable containing the length of the data record to be sent. The length may be zero or greater. If zero, a null data record is sent.

MAP_NAME specifies whether the data record is to be mapped:

- **NO** specifies that data mapping is to be suppressed. The data record is sent as is, without being mapped.
- **YES** specifies that the data record is to be mapped using the map name contained in the variable. The map name is a non-null user-defined name that identifies the format of the data record and the mapping to be performed on the data record before it is sent.

FMH_DATA specifies whether the data record contains FM headers.

- **NO** specifies that FM headers are not present in the data record.
- **YES** specifies that the data record contains FM headers.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the local program. The return code indicates the result of verb execution.

---

[1] The data format for the basic conversation verb, SEND_DATA, consists of logical records, which include a length field. See the description of SEND_DATA for more details.

- OK
- ALLOCATION_ERROR
- BACKED_OUT
- DEALLOCATE_ABEND
- FMH_DATA_NOT_SUPPORTED
- MAPPING_NOT_SUPPORTED
- MAP_NOT_FOUND
- MAP_EXECUTION_FAILURE
- PROG_ERROR_PURGING
- RESOURCE_FAILURE_NO_RETRY
- RESOURCE_FAILURE_RETRY

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received. The indication is either YES or NO.

- YES indicates a REQUEST_TO_SEND notification has been received from the remote transaction program. The remote program has issued MC_REQUEST_TO_SEND, requesting the local program to enter receive state and thereby place the remote program in send state.

- NO indicates a REQUEST_TO_SEND notification has not been received.

State Changes (when RETURN CODE indicates OK):

None

ABEND Conditions:

Parameter Check

- RESOURCE specifies an unassigned resource ID.
- MAP_NAME(YES(variable)) is specified and not supported.
- FMH_DATA(YES) is specified and not supported.

State Check

The mapped conversation is not is send state.

Notes:

1. The mapped conversation protocol boundary provides for the sending and receiving of data records. Unlike the logical records defined for the basic conversation protocol boundary, data records contain only data; they do not contain the logical record length field.

2. The MC_SEND_DATA verb sends one complete data record. Thus, the sending program cannot truncate a data record.

3. The LU buffers the data to be sent to the remote LU until it accumulates from one or more MC_SEND_DATA verbs a sufficient amount for transmission, or until the local program issues a verb that causes the LU to flush its send buffer. The amount of data that is sufficient for transmission depends on the characteristics of the session allocated for the mapped conversation, and can vary from one session to another.

4. The MAP_NAME parameter is used to specify data mapping. The data mapping function uses the MAP_NAME parameter as follows:

   - MAP_NAME(NO) is used to generate a null (zero-length) value for the map name, which suppresses data mapping.

   - MAP_NAME(YES(variable)) is used to specify a non-null map name, which invokes data mapping.

   The data mapping may be performed by the local LU, remote LU, or both, depending on the data mapping function. When a mapped conversation is started, data mapping is initially suppressed until MAP_NAME(YES(variable)) is specified, at which time data mapping is invoked. During the remainder of the conversation data mapping

of each data record is either invoked or suppressed as the MAP_NAME parameter specifies.

The data mapping function underlying the mapped conversation protocol boundary includes the sending of the map name to the remote LU. The local LU sends the map name when data mapping is first invoked on the mapped conversation, and thereafter whenever the one to be sent differs from the one previously sent. This protocol for sending the map name and data applies independently in each direction on the mapped conversation.

5. The data mapping function underlying the mapped conversation protocol boundary may include mapping of the map name itself, depending on the mapping function. Consequently, the local program may specify a map name that differs from the map name the remote program receives. For example, the DATA parameter may specify a high-level-language data structure, which the local LU must serialize for transmission. Correspondingly, the remote LU may have to map the serialized data into a (possibly different) high-level-language data structure for the remote program. In this example, the local LU maps the program-specified map name to a second map name that describes the format of the serialized data, and sends the second map name together with the serialized data to the remote LU. The remote LU maps the second map name to a third map name that describes the structure of the data passed to the remote program.

6. It is the responsibility of both sending and receiving installations to maintain the map-name definitions referred to by their application transaction programs.

7. The function of FM headers in the data record is significant only to the transaction programs; the sending and receiving LUs perform no FM-header related processing other than indicating that the data record contains FM headers. The presence of FM headers in the data record is indicated to the remote transaction program by means of the WHAT_RECEIVED parameter of the MC_RECEIVE_AND_WAIT or MC_RECEIVE_IMMEDIATE verb that receives the data record.

8. When REQUEST_TO_SEND_RECEIVED indicates YES, the remote program is requesting the local program to enter receive state and thereby place the remote program in send state. A program enters receive state by means of the MC_PREPARE_TO_RECEIVE or MC_RECEIVE_AND_WAIT verb. The partner program enters the corresponding send state when it issues an MC_RECEIVE_AND_WAIT or MC_RECEIVE_IMMEDIATE verb and receives the SEND indication (on the WHAT_RECEIVED parameter).

9. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

MC_SEND_ERROR

Informs the remote transaction program that the local program detected an application error. If the mapped conversation is in send state, the LU flushes its send buffer.

Upon successful completion of this verb, the local program is in send state and the remote program is in receive state. Further action is defined by transaction program logic.

MC_SEND_ERROR	Supplied Parameters: RESOURCE (variable)
	Returned Parameters: RETURN_CODE (variable) REQUEST_TO_SEND_RECEIVED (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the local program. The return code indicates the result of verb execution. The return codes that can be returned depend on the state of the mapped conversation at the time this verb is issued:

- If this verb is issued in send state, the following return codes can be returned:
  - OK
  - ALLOCATION_ERROR
  - BACKED_OUT
  - DEALLOCATE_ABEND
  - PROG_ERROR_PURGING
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY
  - FMH_DATA_NOT_SUPPORTED
  - MAPPING_NOT_SUPPORTED
  - MAP_NOT_FOUND
  - MAP_EXECUTION_FAILURE
- If this verb is issued in receive state, the following return codes can be returned:
  - OK
  - DEALLOCATE_NORMAL
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY
- If this verb is issued in confirm state or sync-point state, the following return codes can be returned:
  - OK
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received. The indication is either YES or NO.

- YES indicates a REQUEST_TO_SEND notification has been received from the remote transaction program. The remote program has issued MC_REQUEST_TO_SEND, requesting the local program to enter receive state and thereby place the remote program in send state.

- NO indicates a REQUEST_TO_SEND notification has not been received.

State Changes (When RETURN CODE indicates OK):

Send state is entered when the verb is issued in receive, confirm, or sync-point state.

No state change occurs when the verb is issued in send state.

ABEND Conditions:

Parameter Check

RESOURCE specifies an unassigned resource ID.

State Check

The mapped conversation is not in send, receive, confirm, or sync-point state.

Notes:

1. The LU may send the error notification to the remote LU immediately, that is, during the processing of this verb, or the LU may defer sending the notification until a later time. The determination is made as follows:

   - If the local product does not support the MC_FLUSH verb (see "Notes on Implementation Details" in Appendix A), then the LU sends the error notification immediately.

   - If the local product does support the MC_FLUSH verb, then the LU may or may not send the notification immediately, depending on the product. If the LU defers sending the notification, it buffers the notification until it accumulates a sufficient amount of information for transmission, or until the local program issues a verb that causes the LU to flush its send buffer. The amount of information that is sufficient for transmission depends on the characteristics of the session allocated for the mapped conversation, and can vary from one session to another.

2. The local program can ensure that the remote program receives the error notification as soon as possible by issuing MC_FLUSH immediately after MC_SEND_ERROR.

3. MC_SEND_ERROR is reported to the remote transaction program as one of the following return codes:

   - PROG_ERROR_NO_TRUNC - The local program issued MC_SEND_ERROR in send state. No data truncation occurs at the mapped conversation protocol boundary.

   - PROG_ERROR_PURGING - The local program issued MC_SEND_ERROR in receive state and all data sent by the remote program and not yet received by the local program, if any, has been purged; or the local program issued MC_SEND_ERROR in confirm or sync-point state, in which case no purging has occurred.

4. When MC_SEND_ERROR is issued in receive state, purging of incoming information occurs. The incoming information that is purged includes the following return code indications:

   - ALLOCATION_ERROR
   - BACKED_OUT
   - DEALLOCATE_ABEND
   - FMH_DATA_NOT_SUPPORTED
   - MAPPING_NOT_SUPPORTED
   - MAP_NOT_FOUND
   - MAP_EXECUTION_FAILURE
   - PROG_ERROR_NO_TRUNC
   - PROG_ERROR_PURGING

The return code DEALLOCATE_NORMAL is reported instead of ALLOCATION_ERROR or DEALLOCATE_ABEND. The return code OK is reported instead of the other return codes. When the return code BACKED_OUT is purged, the remote LU resends the BACKED_OUT indication and the local program receives the return code on a subsequent verb.

The other kinds of incoming information that are purged are:

- Data, sent by means of the MC_SEND_DATA verb.
- Map name, sent by means of the MC_SEND_DATA verb.
- Confirmation request, sent by means of the MC_CONFIRM, MC_PREPARE_TO_RECEIVE, or MC_DEALLOCATE verb.
- Sync point request, sent by means of the SYNCPT, MC_PREPARE_TO_RECEIVE, or MC_DEALLOCATE verb.

If the confirmation or sync point request was sent in conjunction with the MC_DEALLOCATE verb (by means of its TYPE(CONFIRM) or TYPE(SYNC_LEVEL) parameter), the deallocation request is also purged.

Incoming information that is not purged is the REQUEST_TO_SEND indication. This indication is reported to the program when it issues a verb that includes the REQUEST_TO_SEND_RECEIVED parameter.

5. When REQUEST_TO_SEND_RECEIVED indicates YES, the remote program is requesting the local program to enter receive state and thereby place the remote program in send state. A program enters receive state by means of the MC_RECEIVE_AND_WAIT or MC_PREPARE_TO_RECEIVE verb. The partner program enters the corresponding send state when it issues an MC_RECEIVE_AND_WAIT or MC_RECEIVE_IMMEDIATE verb and receives the SEND indication (on the WHAT_RECEIVED parameter).

6. The program may use this verb for various application-level functions. For example, the program may issue this verb to inform the remote program of an error it detected in the data records it received, or to reject a confirmation or sync-point request.

7. MC_SEND_ERROR resets or cancels posting. If posting is active and the mapped conversation has been posted, posting is reset. If posting is active and the mapped conversation has not been posted, posting is canceled (posting will not occur). See the MC_POST_ON_RECEIPT verb for more details about posting.

8. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

MC_TEST

Tests the specified mapped conversation for a condition. The return code indicates the result of the test.

MC_TEST	**Supplied Parameters:** RESOURCE (variable) [TEST (POSTED) (REQUEST_TO_SEND_RECEIVED)]
	Returned Parameters: RETURN_CODE (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

TEST specifies the condition to be tested.

- **POSTED** specifies to test whether the mapped conversation has been posted. The return code indicates whether posting has occurred.
- **REQUEST_TO_SEND_RECEIVED** specifies to test whether REQUEST_TO_SEND notification has been received from the remote transaction program. The return code indicates whether the notification has been received.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the program. The return code indicates the result of the test. The TEST parameter determines which of the following return codes can be returned to the program.

- If TEST(POSTED) is specified, one of the following return codes is returned:
  - OK
    - DATA
    - NOT_DATA
  - POSTING_NOT_ACTIVE
  - UNSUCCESSFUL
  - ALLOCATION_ERROR
  - BACKED_OUT
  - DEALLOCATE_NORMAL
  - DEALLOCATE_ABEND
  - FMH_DATA_NOT_SUPPORTED
  - MAP_EXECUTION_FAILURE
  - MAP_NOT_FOUND
  - MAPPING_NOT_SUPPORTED
  - PROG_ERROR_NO_TRUNC
  - PROG_ERROR_PURGING
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY
- If TEST(REQUEST_TO_SEND_RECEIVED) is specified, one of the following return codes is returned:
  - OK
  - UNSUCCESSFUL

State Changes (When RETURN CODE indicates OK):

None

ABEND Conditions:

Parameter Check

- This verb is not supported.
- TEST(POSTED) is specified and not supported.
- TEST(REQUEST_TO_SEND_RECEIVED) is specified and not supported.
- RESOURCE specifies an unassigned resource ID.

State Check

- TEST(POSTED) is specified and the mapped conversation is not in receive state.
- TEST(REQUEST_TO_SEND_RECEIVED) is specified and the mapped conversation is not in send, defer, or receive state.

Notes:

1. The TEST(POSTED) parameter on this verb is intended to be used in conjunction with MC_POST_ON_RECEIPT. The use of MC_POST_ON_RECEIPT and this verb allows a program to continue its processing while waiting for information to become available, where the program issues MC_POST_ON_RECEIPT for one or more mapped conversations and then issues this verb for each mapped conversation to determine when information is available to be received.

2. For TEST(POSTED), the return code indicates whether posting has occurred, as follows:

   - OK indicates posting was active for the mapped conversation and it has been posted. Posting is now reset. The subcode of the OK return code indicates why the mapped conversation has been posted.

     - DATA indicates data is available for the program to receive.

     - NOT_DATA indicates information other than data, such as a SEND, CONFIRM, or TAKE_SYNCPT indication, is available for the program to receive.

     The program should issue MC_RECEIVE_AND_WAIT or MC_RECEIVE_IMMEDIATE in order to receive the information. The program may use the subcode to determine whether it needs to specify the DATA parameter on the MC_RECEIVE_AND_WAIT or MC_RECEIVE_IMMEDIATE verb.

   - POSTING_NOT_ACTIVE indicates posting is not active for the mapped conversation.

   - UNSUCCESSFUL indicates posting is active for the mapped conversation and it has not been posted. Posting remains active.

   The remaining return codes indicate posting was active for the mapped conversation and it has been posted for the reason indicated by the specific return code. Posting is now reset.

3. Posting is active for a mapped conversation when MC_POST_ON_RECEIPT has been issued for the mapped conversation and posting has not been reset or canceled (see the MC_POST_ON_RECEIPT verb).

4. The TEST(REQUEST_TO_SEND_RECEIVED) parameter specifies to test whether REQUEST_TO_SEND notification has been received from the remote transaction program. The return code indicates whether the notification has been received, as follows:

   - OK indicates REQUEST_TO_SEND has been received. The remote program has issued MC_REQUEST_TO_SEND, requesting the local program to enter receive state and thereby place the remote program in send state. A program enters receive state by means of the MC_RECEIVE_AND_WAIT or MC_PREPARE_TO_RECEIVE verb. The partner program enters the corresponding send

state when it issues an MC_RECEIVE_AND_WAIT or MC_RECEIVE_IMMEDIATE verb and receives the SEND indication (on the WHAT_RECEIVED parameter).

- UNSUCCESSFUL indicates REQUEST_TO_SEND has not been received.

5. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

TYPE-INDEPENDENT CONVERSATION VERBS

This section describes the subcategory of conversation verbs called type-independent conversation verbs. These verbs are intended for use on both mapped conversations and basic conversations. In particular, the BACKOUT, SYNCPT, and WAIT verbs can be issued against multiple conversations, which can consist of either mapped or basic conversations or both. The GET_TYPE verb is issued against a single conversation, either mapped or basic.

The detailed descriptions of the type-independent conversation verbs follow. References to verbs that can be either mapped or basic conversation verbs are shown with the "[MC_]" prefix in the verb name.

BACKOUT

Restores all protected resources to their status as of the last synchronization point. Protected resources are those currently allocated to the transaction with a synchronization level of SYNCPT. The last synchronization point is either the start of the transaction, or the completion of the last successful sync point function if one was executed since the start of the transaction. As part of the backout function, the LU flushes its send buffers for all protected resources that are in send or defer state.

BACKOUT	;

Parameters:

No parameters are defined for this verb.

State Changes:

The state of each protected resource at the completion of this verb is the same as it was immediately following the last synchronization point.

ABEND Conditions:

Parameter Check

This verb is not supported.

State Check

At least one protected resource is not in send, defer, receive, confirm, sync point, or backed-out state.

Notes:

1. The BACKOUT verb causes the local LU to restore all local protected resources to their status as of the last synchronization point, and to send a backed-out indication on all protected conversations. (A protected conversation is one that is allocated with a synchronization level of SYNCPT.)

2. Any program throughout the distributed transaction may initiate the backout function, that is, may be the first to issue BACKOUT since the last synchronization point. It does so when it determines that an error or exceptional condition exists that requires restoring all protected resources to their last synchronization point. The program can initiate the backout function as a response to a sync point request, or at other times unrelated to a sync point request. All other programs interconnected by protected conversations are informed, by means of the BACKED_OUT return code, that the backout function has been initiated.

3. A program must issue this verb whenever it receives a BACKED_OUT return code, in order to extend the backout function to all protected resources throughout the transaction.

4. BACKOUT resets or cancels posting. If posting is active and the resource has been posted, posting is reset. If posting is active and the resource has not been posted, posting is canceled (posting will not occur). See the [MC_]POST_ON_RECEIPT verb for details about posting of a conversation.

GET_TYPE

Returns the type of resource to which the specified resource ID is assigned.

GET_TYPE	Supplied Parameters: RESOURCE (variable)
	Returned Parameters: TYPE (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID of the resource of which the type is desired.

Returned Parameters:

TYPE specifies the variable for returning the type of resource that is allocated. The types are:

- BASIC_CONVERSATION
- MAPPED_CONVERSATION

State Changes:

None

ABEND Conditions:

Parameter Check

- This verb is not supported.
- RESOURCE specifies an unassigned resource ID.

State Check

None

Notes:

1. A program that can be processed at either the basic conversation protocol boundary or the mapped conversation protocol boundary issues this verb in order to determine which category of verbs, basic conversation or mapped conversation, it is to use for the resource.

SYNCPT

Advances all protected resources to the next synchronization point. Protected resources are those currently allocated to the transaction with a synchronization level of SYNCPT. As part of the sync point function, the LU flushes its send buffers for all protected resources that are in send or defer state.

SYNCPT	Returned Parameters: RETURN_CODE (variable) REQUEST_TO_SEND_RECEIVED (variable)
	;

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the program. The return code indicates the result of the sync point function.

- OK (sync point is successful)
- BACKED_OUT
- HEURISTIC_MIXED

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received. The indication is either YES or NO.

- YES indicates a REQUEST_TO_SEND notification has been received from one or more remote programs.
- NO indicates a REQUEST_TO_SEND notification has not been received.

State Changes (when RETURN_CODE indicates OK):

Reset state is entered when the verb is issued in the defer state entered by the preceding [MC_]DEALLOCATE verb.

Receive state is entered when the verb is issued in the defer state entered by the preceding [MC_]PREPARE_TO_RECEIVE verb, or when the verb is issued in the sync point state entered by receipt of TAKE_SYNCPT on the preceding [MC_]RECEIVE_AND_WAIT or [MC_]RECEIVE_IMMEDIATE verb.

Send state is entered when the verb is issued in the sync point state entered by receipt of TAKE_SYNCPT_SEND on the preceding [MC_]RECEIVE_AND_WAIT or [MC_]RECEIVE_IMMEDIATE verb.

Deallocate state is entered when the verb is issued in the sync point state entered by receipt of TAKE_SYNCPT_DEALLOCATE on the preceding [MC_]RECEIVE_AND_WAIT or [MC_]RECEIVE_IMMEDIATE verb.

No state change occurs when the verb is issued in send state.

ABEND Conditions:

Parameter Check

- This verb is not supported.

State Check

- A protected resource is not in send, defer, or sync point state.
- A protected resource is in send state, and the program started but did not finish sending a basic conversation logical record.

Notes:

1. The program may issue SYNCPT when all protected conversations are in send, defer, or sync point state, or a combination of these states; however, only one conversation can be in sync point state. (A protected conversation is one that is allocated with a synchronization level of SYNCPT.) The remote programs receive the sync point request by means of the WHAT_RECEIVED parameter of the [MC_]RECEIVE_AND_WAIT or [MC_]RECEIVE_IMMEDIATE verb, as follows:

   - On conversations for which the local program is in send state, the remote programs receive the TAKE_SYNCPT indication.
   - On conversations in defer state entered by means of a preceding [MC_]PREPARE_TO_RECEIVE verb, the remote programs receive the TAKE_SYNCPT_SEND indication.
   - On conversations in defer state entered by means of a preceding [MC_]DEALLOCATE verb, the remote programs receive the TAKE_SYNCPT_DEALLOCATE indication.

2. In a distributed transaction, one program (usually chosen during transaction design) is the initiator for sync point processing. The other programs each cooperate in propagating the sync point processing throughout the distributed transaction. The program initiating sync point processing issues SYNCPT, which causes its LU to send a sync point request on all of the protected conversations allocated to the program. Each program receiving the sync point request may issue SYNCPT, thereby propagating the request throughout the transaction. When all participating programs respond to the sync point request by issuing SYNCPT, their LUs and the initiating program's LU advance their respective local resources to the next synchronization point.

3. All protected resources, including conversations, allocated to the local transaction program must be in send, defer, or sync point state when the program issues SYNCPT. If one or more protected conversations are in receive state, the program may issue [MC_]REQUEST_TO_SEND on those conversations to request send control.

4. The return code indicates whether the sync point function was successful.

   - OK indicates all protected resources have been advanced to the next synchronization point.
   - BACKED_OUT indicates all protected resources are to be restored to their status as of the last synchronization point. The program must issue BACKOUT, which causes the backout function to be performed on all protected resources throughout the transaction.
   - HEURISTIC_MIXED indicates that some protected resources throughout the distributed transaction have been advanced to the next synchronization point and others have been restored to the previous synchronization point as a result of an error during the sync point processing. This mixed status of protected resources occurs when an LU operator intervenes in an attempt to recover from the error. See SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2 for more details.

5. Use of sync point ensures consistency of the protected resources involved in a distributed transaction. Consistency means that if the return code, OK, is returned to the transaction program that issued the first SYNCPT verb (called the initiator), OK will also have been returned to the dependent SYNCPT verbs issued by every other transaction program participating in the distributed logical unit of work.

   Similarly, consistency means that if the BACKED_OUT return code is received on any protected conversation in a distributed trans-

action, BACKED_OUT will be received on all protected conversations in the distributed transaction. Further, all protected local resources that share in the distributed logical unit of work will be backed out to the most recent point of successful commitment.

Of particular importance are updates to files or data bases. For example, take the case of a fund transfer from an account maintained at one node to an account maintained at another node; use of SYNCPT will ensure, except when heuristic decisions must be made, that the debit from one account will be credited to the other.

6. The processing of unprotected resources is the program's responsibility. If the sync point function is successful, the program should advance all unprotected resources associated with the transaction to a consistent state. If the sync point function is unsuccessful, the unprotected resources should be restored to a state consistent with the previous synchronization point.

7. When REQUEST_TO_SEND_RECEIVED indicates YES, one or more remote programs are requesting the local program to enter receive state and thereby place the remote programs in send state. For each resource on which a REQUEST_TO_SEND notification was received, the notification will also be reported to the local program on the next resource-specific verb it issues that has the REQUEST_TO_SEND_RECEIVED parameter.

8. References in this verb description to a program being in a particular state are only in terms of each resource.

Waits for posting to occur on any basic or mapped conversation from among a list of conversations. Posting of a conversation occurs when posting is active for the conversation and the LU has any information that the program can receive, such as data, conversation status, or a request for confirmation or sync point.

WAIT	**Supplied Parameters:** **RESOURCE_LIST (variable1 variable2 ... variablen)**
	Returned Parameters: **RETURN_CODE (variable)** **RESOURCE_POSTED (variable)**
	;

Supplied Parameters:

RESOURCE_LIST specifies the variables containing the resource IDs of the conversations for which posting is expected.

- **variable1 variable2 ... variablen** are the variables containing the individual resource IDs. One or more resource IDs may be specified.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the program. The return code indicates the result of verb execution. The type of conversation posted determines which of the return codes can be returned to the program.

- If a mapped conversation is posted, one of the following return codes is returned:
  - OK
    - DATA
    - NOT_DATA
  - POSTING_NOT_ACTIVE
  - ALLOCATION_ERROR
  - BACKED_OUT
  - DEALLOCATE_ABEND
  - DEALLOCATE_NORMAL
  - FMH_DATA_NOT_SUPPORTED
  - MAP_EXECUTION_FAILURE
  - MAP_NOT_FOUND
  - MAPPING_NOT_SUPPORTED
  - PROG_ERROR_NO_TRUNC
  - PROG_ERROR_PURGING
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY
- If a basic conversation is posted, one of the following return codes is returned:
  - OK
    - DATA
    - NOT_DATA
  - POSTING_NOT_ACTIVE
  - ALLOCATION_ERROR
  - BACKED_OUT
  - DEALLOCATE_ABEND_PROG
  - DEALLOCATE_ABEND_SVC
  - DEALLOCATE_ABEND_TIMER
  - DEALLOCATE_NORMAL
  - PROG_ERROR_NO_TRUNC
  - PROG_ERROR_PURGING
  - PROG_ERROR_TRUNC

- SVC_ERROR_NO_TRUNC
- SVC_ERROR_PURGING
- SVC_ERROR_TRUNC
- RESOURCE_FAILURE_NO_RETRY
- RESOURCE_FAILURE_RETRY

RESOURCE_POSTED specifies the variable in which the resource ID of the posted conversation is returned to the program.

State Changes (When RETURN_CODE indicates OK):

None

ABEND Conditions:

Parameter Check

- This verb is not supported.
- RESOURCE_LIST specifies an unassigned resource ID.

State Check

None

Notes:

1. This verb is intended to be used in conjunction with [MC_]POST_ON_RECEIPT. The use of [MC_]POST_ON_RECEIPT and this verb allows a program to perform synchronous receiving from multiple conversations, where the program issues [MC_]POST_ON_RECEIPT for each of the conversations and then issues this verb (for each conversation) to wait until information is available to be received on the conversations.

2. The RESOURCE_LIST parameter may specify any combination of basic and mapped conversations. Posting for each conversation may be active or not active. This verb waits for posting to occur only on the conversations for which posting is active. When a conversation is posted, the resource ID of the posted conversation is returned to the program by means of the RESOURCE_POSTED parameter.

3. The return code indicates whether posting has occurred, as follows:

   - OK indicates posting was active for a conversation and it has been posted. Posting is now reset for the conversation. The subcode of the OK return code indicates why the conversation has been posted.

     - DATA indicates data is available for the program to receive.

     - NOT_DATA indicates information other than data, such as a SEND, CONFIRM, or TAKE_SYNCPT indication, is available for the program to receive.

     The program should issue [MC_]RECEIVE_AND_WAIT or [MC_]RECEIVE_IMMEDIATE in order to receive the information. The program may use the subcode to determine whether it needs to specify the DATA parameter on the [MC_]RECEIVE_AND_WAIT or [MC_]RECEIVE_IMMEDIATE verb.

   - POSTING_NOT_ACTIVE indicates posting is not active for any and all of the conversations.

   The remaining return codes indicate posting was active for a conversation and it has been posted for the reason indicated by the specific return code. Posting is now reset for the conversation.

4. Posting is active for a conversation when [MC_]POST_ON_RECEIPT has been issued for the conversation and posting has not been reset or canceled (see the [MC_]POST_ON_RECEIPT verb).

BASIC CONVERSATION VERBS

This section describes the subcategory of conversation verbs called basic conversation verbs. These verbs are intended for use by LU services programs. The LU services programs can provide end-user services or protocol boundaries for end-user application transaction programs. Examples of LU services programs are:

- The LU services component programs that process mapped conversation verbs and control-operator verbs. These verbs define the LU 6.2 protocol boundary for mapped conversations and the control operator.

- SNA service transaction programs. These programs provide end-user protocol boundaries that are defined by the specific IBM product implementations of the service programs. Refer to the IBM product publications for a description of the SNA service programs and their protocol boundaries that each product provides. The names of some SNA service transaction programs that have general applicability are listed in "Appendix D. List of SNA Service Transaction Programs".

The detailed descriptions of the basic conversation verbs follow.

Note: Every conversation is either a basic or mapped conversation. The basic conversation verbs can be used for operations on both types. The mapped conversation verbs can be used for operations only on a mapped conversation. The capability to use basic conversation verbs on mapped conversations is provided for implementation of a mapped conversation LU services component program. Throughout the descriptions of the basic conversation verbs, references to a basic conversation or mapped conversation are made only when it is necessary to make a distinction between them. Otherwise, references are made simply to conversations.

ALLOCATE

Allocates a session between the local LU and a remote LU, and on that session allocates a basic or mapped conversation between the local program and a remote program. A resource ID is assigned to the conversation. This verb is issued prior to any verbs that refer to the conversation.

```
ALLOCATE

Supplied Parameters:

LU_NAME { OWN }
        { OTHER { variable } }

MODE_NAME { variable }

TPN { variable }

[ TYPE { BASIC_CONVERSATION }
       { MAPPED_CONVERSATION } ]

[                  { WHEN_SESSION_ALLOCATED }                    ]
[ RETURN_CONTROL   { DELAYED_ALLOCATION_PERMITTED }              ]
[                  { IMMEDIATE }                                 ]

[              { NONE }                                          ]
[ SYNC_LEVEL   { CONFIRM }                                       ]
[              { SYNCPT }                                        ]

[           { NONE }                                             ]
[ SECURITY  { SAME }                                             ]
[           { PGM { USER_ID { variable } PASSWORD { variable }   ]
[                   PROFILE { variable } } }                     ]

[ PIP { NO }                                                     ]
[     { YES { variable1 variable2 ... variablen } }              ]

Returned Parameters:

RESOURCE { variable }

RETURN_CODE { variable }

;
```

Supplied Parameters:

LU_NAME specifies the name of the remote LU at which the remote transaction program is located. This LU name is any name by which the local LU knows the remote LU for the purpose of allocating a conversation. The local LU transforms this locally-known LU name to an LU name used by the network, if the names are different.

- **OWN** specifies that the remote program is located at the same LU as the local program.
- **OTHER** specifies that the remote program is located at another LU. The specified variable contains the LU name.

MODE_NAME specifies the mode name designating the network properties for the session to be allocated for the conversation. The network properties include, for example, the class of service to be used, and whether data is to be enciphered or translated to ASCII before it is sent. The SNA-defined mode name, SNASVCMG, may be specified, but only by an LU services program.

TPN specifies the name of the remote transaction program to be connected at the other end of the conversation. A transaction program that has the appropriate privilege may specify the name of an SNA service transaction program. Privilege is an identification that a product or installation defines in order to differentiate LU services transaction programs from other programs, such as application transaction programs. (See "Appendix D. List of SNA Service Transaction Programs" for more details about SNA service transaction program names.)

TYPE specifies the type of conversation to be allocated.

- **BASIC_CONVERSATION** specifies to allocate a basic conversation.
- **MAPPED_CONVERSATION** specifies to allocate a mapped conversation. This argument is used in support of mapped conversation verbs. It may be specified only by a mapped conversation LU services program.

RETURN_CONTROL specifies when the local LU is to return control to the local program, in relation to the allocation of a session for the conversation. An allocation error resulting from the local LU's failure to obtain a session for the conversation is reported either on this verb or a subsequent verb, depending on the argument specified for this parameter. An allocation error resulting from the remote LU's rejection of the allocation request is reported on a subsequent verb.

- **WHEN_SESSION_ALLOCATED** specifies to allocate a session for the conversation before returning control to the program. An error in allocating a session is reported on this verb.
- **DELAYED_ALLOCATION_PERMITTED** specifies to allocate a session for the conversation after returning control to the program. An error in allocating a session is reported on a subsequent verb.
- **IMMEDIATE** specifies to allocate a session for the conversation if a session is immediately available, and return control to the program with a return code indicating whether a session is allocated.
  - A return code of OK indicates a session is immediately available and is allocated for the conversation. A session is immediately available when it is active, it is not allocated to another conversation, and the local LU is the contention winner for the session.
  - A return code of UNSUCCESSFUL indicates a session is not immediately available. Allocation is not performed.

  An error in allocating a session that is immediately available is reported on this verb.

SYNC_LEVEL specifies the synchronization level that the local and remote programs can use on this conversation.

- **NONE** specifies that the programs will not perform confirmation or sync point processing on this conversation. The programs will not issue any verbs and will not recognize any returned parameters relating to these synchronization functions.
- **CONFIRM** specifies that the programs can perform confirmation processing but not sync-point processing on this conversation. The programs may issue verbs and will recognize returned parameters relating to confirmation, but they will not issue any verbs and will not recognize any returned parameters relating to sync point.
- **SYNCPT** specifies that the programs can perform both confirmation and sync-point processing on this conversation. The programs may issue verbs and will recognize returned parameters relating to confirmation or sync point. For sync-point processing, a conversation allocated with this synchronization level is a protected resource.

SECURITY specifies access security information that the remote LU uses to verify the identity of the end-user and validate access to the remote program and its resources. The access security information consists of a user ID, a password, and a profile.

- **NONE** specifies to omit access security information on this allocation request.
- **SAME** specifies to use the user ID and profile (if present) from the allocation request that initiated execution of the local program. The password (if present) is not used; instead, the user ID is indicated as being already verified. If the allocation request that initiated execution of the local program contained no access security information, then access security information is omitted on this allocation request.
- **PGM** specifies to use the access security information that the local program provides on this parameter. The local program provides the information by means of the following arguments:
  - **USER_ID** specifies the variable containing the user ID. The remote LU uses this value and the password to verify the identity of the end-user making the allocation request. In addition, the remote LU may use the user ID for auditing or accounting purposes, or it may use the user ID, together with the profile (if present), to determine which remote programs the local program may access and which resources the remote program may access.
  - **PASSWORD** specifies the variable containing the password. The remote LU uses this value and the user ID to verify the identity of the end-user making the allocation request.
  - **PROFILE** specifies the variable containing the profile. The remote LU may use this value, in addition to or in place of the user ID, to determine which remote programs the local program may access, and which resources the remote program may access.

  Specifying a null value for any of the access security arguments is equivalent to omitting the argument.

PIP specifies program initialization parameters for the remote program.

- **NO** specifies that PIP data is not present.
- **YES** specifies that PIP data is present.
  - **variable1 variable2 ... variablen** contain the PIP data to be sent to the remote program. The PIP data consists of one or more subfields, each of which is specified by a separate variable; variables 1 through n correspond to subfields 1 through n. If a variable is omitted in the PIP parameter or it is of null value, the corresponding PIP subfield is made to be of 0 length. The number of PIP subfields must agree with the number of PIP variables specified on the remote program's PROC statement (see "Transaction Program Structure and Execution" in Chapter 3).

Returned Parameters:

RESOURCE specifies the variable in which the resource ID is to be returned. The length and actual format of the resource ID is product dependent. The resource ID is returned to the program when the return code is either OK or ALLOCATION_ERROR.

RETURN_CODE specifies the variable in which a return code is returned to the local program. The return code indicates the result of verb execution. The RETURN_CONTROL parameter determines which of the following return codes can be returned to the program.

- If RETURN_CONTROL(WHEN_SESSION_ALLOCATED) is specified, one of the following return codes is returned:

- OK
- ALLOCATION_ERROR (with one of the following subcodes)
  - ALLOCATION_FAILURE_NO_RETRY
  - ALLOCATION_FAILURE_RETRY
  - SYNC_LEVEL_NOT_SUPPORTED_BY_LU
- PARAMETER_ERROR (for the following reasons)
  - Invalid LU name
  - Invalid mode name

- If RETURN_CONTROL(DELAYED_ALLOCATION_PERMITTED) is specified, one of the following return codes is returned:

  - OK
  - PARAMETER_ERROR (for one of the following reasons)
    - Invalid LU name
    - Invalid mode name

- If RETURN_CONTROL(IMMEDIATE) is specified, one of the following return codes is returned:

  - OK
  - ALLOCATION_ERROR (with the following subcode)
    - SYNC_LEVEL_NOT_SUPPORTED_BY_LU
  - PARAMETER_ERROR (for one of the following reasons)
    - Invalid LU name
    - Invalid mode name
  - UNSUCCESSFUL (for the following reason)
    - Session not immediately available

State Changes (When RETURN_CODE indicates OK):

Send state is entered.

ABEND Conditions:

Parameter Check

- LU_NAME(OWN) is specified and not supported.
- MODE_NAME specifies SNASVCMG and the local program is not an LU services program.
- TPN specifies an SNA service transaction program name and the local program does not have the appropriate privilege to allocate a conversation to an SNA service program.
- TPN specifies a null (0 length) value.
- TYPE(BASIC_CONVERSATION) is specified and the local program does not have basic conversation support defined.
- TYPE(MAPPED_CONVERSATION) is specified and the local program is not a mapped conversation LU services program.
- RETURN_CONTROL(DELAYED_ALLOCATION_PERMITTED) is specified and not supported.
- RETURN_CONTROL(IMMEDIATE) is specified and not supported.
- SYNC_LEVEL(SYNCPT) is specified and not supported.
- SECURITY(SAME) is specified and not supported.
- SECURITY(PGM(USER_ID(variable) PASSWORD(variable))) is specified and not supported.
- SECURITY(PGM(PROFILE(variable))) is specified and not supported.
- PIP(YES(variable)) is specified and not supported.

State Check

None

Notes:

1. This verb is used by a transaction program to allocate a basic conversation. It is also used by an LU services component program to allocate either a basic conversation or a mapped conversation, depending on the function that the component program provides. For example, a component program that processes control operator verbs uses this verb to allocate a basic conversation, and a component program that processes mapped conversation verbs uses this verb to allocate a mapped conversation.

2. Depending on the product, the LU may send the allocation request to the remote LU as soon as it allocates a session for the conversation. Alternatively, the LU may buffer the allocation request until it accumulates from the PIP parameter of this verb or from one or more subsequent SEND_DATA verbs a sufficient amount of information for transmission, or until the local program issues a subsequent verb other than SEND_DATA that explicitly causes the LU to flush its send buffer. The amount of information that is sufficient for transmission depends on the characteristics of the session allocated for the conversation, and can vary from one session to another.

3. The local program can ensure that the remote program is connected as soon as possible by issuing FLUSH immediately after ALLOCATE.

4. Two LUs connected by a session may both attempt to allocate a conversation on the session at the same time. This is called contention. Contention is resolved by making one LU the contention winner of the session and the other LU the contention loser of the session. The contention-winner LU allocates a conversation on a session without asking permission from the contention-loser LU. Conversely, the contention-loser LU requests permission from the contention-winner LU to allocate a conversation on the session, and the contention-winner LU either grants or rejects the request.

5. If the program issues ALLOCATE with the parameter RETURN_CONTROL(DELAYED_ALLOCATION_PERMITTED), the LU delays allocation of the session until it flushes its send buffer. At that time the LU allocates the session and transmits the allocation request to the remote LU. The program is unaffected by the delayed allocation of the session, with one exception: When the LU allocates a contention-loser session, it does so by transmitting the allocation request and then waiting for information to arrive before returning control to the program. This can affect the sequence of the verbs that the program can issue.

   For example, suppose the program has the following sequence of verbs:

   ALLOCATE with RETURN_CONTROL(DELAYED_ALLOCATION_PERMITTED)

   PREPARE_TO_RECEIVE with TYPE(FLUSH)

   REQUEST_TO_SEND

   In this example, assume the program is using REQUEST_TO_SEND to prompt the remote program to begin sending information, instead of requesting send control. However, if the LU allocates a contention-loser session (and an allocation error or resource failure does not occur), control is not returned to the program after it issues the PREPARE_TO_RECEIVE until the remote program sends some information. If the remote program waits for the REQUEST_TO_SEND notification before sending any information, a deadlock condition occurs. This deadlock can be avoided by issuing the ALLOCATE with either RETURN_CONTROL (WHEN_SESSION_ALLOCATED) or RETURN_CONTROL (IMMEDIATE).

6. SYNC_LEVEL(SYNCPT) permits use of the SYNCPT and BACKOUT verbs and the Resynchronization transaction program (an SNA service transaction program), to aid in maintaining consistency across all protected resources within a distributed logical unit of work. The Resynchronization program performs sync point resynchronization, which maintains this consistency when session failure and reinitiation occurs. See SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2 for more details of sync point resynchronization.

7. Each LU indicates at session activation time whether it will accept LU security parameters on allocation requests the partner LU sends. If the remote LU will not accept any security parameters from the local LU, and the local program specifies SECURITY(SAME) or SECURITY(PGM(...)), the local LU downgrades the specification to SECURITY(NONE). Similarly, if the remote LU

will not accept the local LU's verification of the user ID and password, and the local program specifies SECURITY(SAME), the local LU downgrades the specification to SECURITY(NONE).

8. The remote program is connected to the other end of the conversation in receive state.

9. The program uses the resource ID, returned to the program on the RESOURCE parameter, on all subsequent basic conversation verbs it issues for this conversation.

10. References in this verb description to a program being in a particular state are only in terms of the allocated conversation.

CONFIRM

Sends a confirmation request to a remote transaction program and waits for a reply. This verb allows the local and remote programs to synchronize their processing with one another. The LU flushes its send buffer as a function of this verb.

CONFIRM	Supplied Parameters: RESOURCE (variable)
	Returned Parameters: RETURN_CODE (variable) REQUEST_TO_SEND_RECEIVED (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID. The conversation must be allocated with a synchronization level of CONFIRM or SYNCPT.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the local program. The return code indicates the result of verb execution.

- OK (remote program replied CONFIRMED)
- ALLOCATION_ERROR
- BACKED_OUT
- DEALLOCATE_ABEND_PROG
- DEALLOCATE_ABEND_SVC
- DEALLOCATE_ABEND_TIMER
- PROG_ERROR_PURGING
- RESOURCE_FAILURE_NO_RETRY
- RESOURCE_FAILURE_RETRY
- SVC_ERROR_PURGING

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received. The indication is either YES or NO.

- YES indicates a REQUEST_TO_SEND notification has been received from the remote transaction program. The remote program has issued REQUEST_TO_SEND, requesting the local program to enter receive state and thereby place the remote program in send state.
- NO indicates a REQUEST_TO_SEND notification has not been received.

State Changes (When RETURN CODE indicates OK):

Receive state is entered when the verb is issued in defer state following PREPARE_TO_RECEIVE.

Reset state is entered when the verb is issued in defer state following DEALLOCATE.

No state change occurs when the verb is issued in send state.

ABEND Conditions:

Parameter Check

- The conversation was allocated with SYNC_LEVEL(NONE).
- RESOURCE specifies an unassigned resource ID.

State Check

- The conversation is not in send or defer state.
- The conversation is in scnd state, and the program started but did not finish sending a logical record.

Notes:

1. The program may use this verb for various application-level functions. For example:

   - The program may issue this verb immediately following an ALLOCATE in order to determine whether the allocation of the conversation is successful before sending any data.

   - The program may issue this verb as a request for acknowledgement of data it sent to the remote program. The remote program may respond by issuing CONFIRMED as an indication that it received and processed the data without error, or by issuing SEND_ERROR as an indication that it encountered an error.

2. When REQUEST_TO_SEND_RECEIVED indicates YES, the remote program requests the local program to enter receive state and thereby place the remote program in send state. A program enters receive state by means of the PREPARE_TO_RECEIVE or RECEIVE_AND_WAIT verb. The partner program enters the corresponding send state when it issues a RECEIVE_AND_WAIT or RECEIVE_IMMEDIATE verb and receives the SEND indication (on the WHAT_RECEIVED parameter).

3. References in this verb description to a program being in a particular state are only in terms of the specified conversation.

CONFIRMED

Sends a confirmation reply to the remote transaction program. This verb allows the local and remote programs to synchronize their processing with one another. The local program can issue this verb when it receives a confirmation request (see the WHAT_RECEIVED parameter of the RECEIVE_AND_WAIT or RECEIVE_IMMEDIATE verb).

CONFIRMED	Supplied Parameters: RESOURCE (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

State Changes:

Receive state is entered when CONFIRM was received on the preceding RECEIVE_AND_WAIT or RECEIVE_IMMEDIATE.

Send state is entered when CONFIRM_SEND was received on the preceding RECEIVE_AND_WAIT or RECEIVE_IMMEDIATE.

Deallocate state is entered when CONFIRM_DEALLOCATE was received on the preceding RECEIVE_AND_WAIT or RECEIVE_IMMEDIATE.

ABEND Conditions:

Parameter Check

RESOURCE specifies an unassigned resource ID.

State Check

The conversation is not in confirm state.

Notes:

1. The program can issue this verb only as a reply to a confirmation request; the verb cannot be issued at any other time.
2. The program may use this verb for various application-level functions. For example, the remote program may send data followed by a confirmation request. When the local program receives the confirmation request, it may issue this verb as an indication that it received and processed the data without error.
3. References in this verb description to a program being in a particular state are only in terms of the specified conversation.

DEALLOCATE

Deallocates the specified conversation from the transaction program. The deallocation can be either completed as part of this verb, or deferred until the program issues a FLUSH, CONFIRM, or SYNCPT verb. When it is completed as part of this verb it can include the function of the FLUSH or CONFIRM verb. The resource ID becomes unassigned when deallocation is complete.

```
DEALLOCATE

Supplied Parameters:

RESOURCE ( variable )

[      ( SYNC_LEVEL )
       ( FLUSH )
       ( CONFIRM )
 TYPE  ( ABEND_PROG )
       ( ABEND_SVC )
       ( ABEND_TIMER )
       ( LOCAL )      ]

[ LOG_DATA ( NO )
           ( YES ( variable ) ) ]

Returned Parameters:

RETURN_CODE ( variable )

;
```

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID of the conversation to be deallocated.

TYPE specifies the type of deallocation to be performed.

- **SYNC_LEVEL** specifies to perform deallocation based on the synchronization level allocated to this conversation:
  - If SYNC_LEVEL(NONE), execute the function of the FLUSH verb and deallocate the conversation normally.
  - If SYNC_LEVEL(CONFIRM), execute the function of the CONFIRM verb and if it is successful (as indicated by a return code of OK on this DEALLOCATE verb), deallocate the conversation normally; if it is not successful, the state of the conversation is determined by the return code.
  - If SYNC_LEVEL(SYNCPT), defer the deallocation until the program issues a SYNCPT, or the program issues a CONFIRM or FLUSH for this conversation. If the SYNCPT or CONFIRM is successful (as indicated by a return code of OK on that verb) or FLUSH is issued, the conversation is then deallocated normally; otherwise, the state of the conversation is determined by the return code.
- **FLUSH** specifies to execute the function of the FLUSH verb and deallocate the conversation normally.
- **CONFIRM** specifies to execute the function of the CONFIRM verb and if it is successful (as indicated by a return code of OK on this DEALLOCATE verb), deallocate the conversation normally; if it is not successful, the state of the conversation is determined by the return code.
- **ABEND_PROG, ABEND_SVC, or ABEND_TIMER** specifies to execute the function of the FLUSH verb when the conversation is in send or defer state, and deallocate the conversation abnormally.

Logical-record truncation can occur when the conversation is in send state; data purging can occur when it is in receive state.

- **LOCAL** specifies to deallocate the conversation locally. This type of deallocation must be specified if, and only if, the conversation is in deallocate state. Deallocate state is entered when the program receives on a previously issued verb a return code indicating the conversation has been deallocated (see "Return Codes" on page 4-99).

The execution of the FLUSH or CONFIRM function as part of this verb includes the flushing of the LU's send buffer. When, instead, the deallocation is deferred, the LU also defers flushing its send buffer until the program issues a subsequent verb for this conversation.

LOG_DATA specifies whether product-unique error information is to be placed in the system error logs of the LUs supporting this conversation. This parameter can be specified only when TYPE(ABEND_PROG), TYPE(ABEND_SVC), or TYPE(ABEND_TIMER) is also specified.

- **NO** specifies that no error information is to be placed in the system error logs.

- **YES** specifies that product-unique error information is to be placed in the system error logs of the local and remote LUs. The specified variable contains the product-unique error information, in the format of the Error Log GDS variable. See SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2 for a definition of the Error Log GDS variable.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the local program. The return code indicates the result of verb execution. The TYPE parameter determines which of the following return codes can be returned to the program.

- If TYPE(SYNC_LEVEL) is specified and the synchronization level allocated to this conversation is NONE; or TYPE(FLUSH), TYPE(ABEND_PROG), TYPE(ABEND_SVC), TYPE(ABEND_TIMER), or TYPE(LOCAL) is specified; the following return code is returned:

  - OK (deallocation is complete)

- If TYPE(SYNC_LEVEL) is specified and the synchronization level allocated to this conversation is CONFIRM, or TYPE(CONFIRM) is specified, one of the following return codes is returned:

  - OK (deallocation is complete)
  - ALLOCATION_ERROR
  - DEALLOCATE_ABEND_PROG
  - DEALLOCATE_ABEND_SVC
  - DEALLOCATE_ABEND_TIMER
  - PROG_ERROR_PURGING
  - SVC_ERROR_PURGING
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY

- If TYPE(SYNC_LEVEL) is specified and the synchronization level allocated to this conversation is SYNCPT, the following return code is returned:

  - OK (deallocation is deferred)

State Changes (when RETURN CODE indicates OK):

Defer state is entered when TYPE(SYNC_LEVEL) is specified and the synchronization level is SYNCPT.

Reset state is entered when TYPE(FLUSH), TYPE(CONFIRM), TYPE(LOCAL), TYPE(ABEND_PROG), TYPE(ABEND_SVC), or TYPE(ABEND_TIMER) is specified; or when TYPE(SYNC_LEVEL) is specified and the synchronization level is NONE or CONFIRM.

ABEND Conditions:

Parameter Check

- RESOURCE specifies an unassigned resource ID.
- TYPE(CONFIRM) is specified and the conversation is allocated with SYNC_LEVEL(NONE).
- TYPE(ABEND_SVC) or TYPE(ABEND_TIMER) is specified and not supported.
- LOG_DATA is specified and not supported.

State Check

- TYPE(FLUSH), TYPE(CONFIRM), or TYPE(SYNC_LEVEL) is specified and the conversation is not in send state.
- TYPE(FLUSH), TYPE(CONFIRM), or TYPE(SYNC_LEVEL) is specified, the conversation is in send state, and the program started but did not finish sending a logical record.
- TYPE(ABEND_PROG), TYPE(ABEND_SVC), or TYPE(ABEND_TIMER) is specified and the conversation is not in send, defer, receive, confirm, or sync-point state.
- TYPE(LOCAL) is specified and the conversation is not in deallocate state.

Notes:

1. When the deallocation is deferred (see the TYPE parameter), the LU buffers the deallocation information to be sent to the remote LU until the local program issues a verb that causes the LU to flush its send buffer.

2. The TYPE(SYNC_LEVEL) parameter is intended to be used by the transaction program in order to deallocate the conversation based on the synchronization level allocated to the conversation.

   - If the synchronization level is NONE, the conversation is unconditionally deallocated.

   - If the synchronization level is CONFIRM, the conversation is deallocated when the remote program responds to the confirmation request by issuing CONFIRMED. The conversation remains allocated when the remote program responds to the confirmation request by issuing SEND_ERROR.

   - If the synchronization level is SYNCPT, the conversation is deallocated when the local program subsequently issues SYNCPT and all programs throughout the transaction, connected to conversations having the synchronization level of SYNCPT, respond to the sync point request by issuing SYNCPT. The conversation remains allocated when the remote program responds to the sync point request by issuing SEND_ERROR, or one or more programs respond by issuing BACKOUT.

3. The TYPE(FLUSH) parameter is intended to be used by the transaction program in order to unconditionally deallocate the conversation regardless of its synchronization level. TYPE(FLUSH) is functionally equivalent to:

   - TYPE(SYNC_LEVEL) with a synchronization level of NONE.

   - TYPE(SYNC_LEVEL) with a synchronization level of SYNCPT, followed by the FLUSH verb.

4. The TYPE(CONFIRM) parameter is intended to be used by the transaction program in order to conditionally deallocate the conversation, depending on the remote program's response, when the synchronization level is CONFIRM or SYNCPT. TYPE(CONFIRM) is functionally equivalent to:

   - TYPE(SYNC_LEVEL) with a synchronization level of CONFIRM.

   - TYPE(SYNC_LEVEL) with a synchronization level of SYNCPT, followed by the CONFIRM verb.

The conversation is deallocated when the remote program responds to the confirmation request by issuing CONFIRMED. The conversation remains allocated when the remote program responds to the confirmation request by issuing SEND_ERROR.

5. The TYPE(ABEND_PROG), TYPE(ABEND_SVC), and TYPE(ABEND_TIMER) parameters are intended to be used in order to unconditionally deallocate the conversation regardless of its synchronization level and its current state. Specifically:

   - The TYPE(ABEND_PROG) parameter is intended to be used by a transaction program when it detects an error condition that prevents further useful communications, that is, communications that would lead to successful completion of the transaction. The specific use and meaning of ABEND_PROG are program-defined.

   - The TYPE(ABEND_SVC) parameter is intended to be used by an LU services component, such as one that processes mapped conversation verbs, when it detects an error condition caused by its peer LU services component in the remote LU. An example is a format error in control information sent by the peer LU services component. The specific use and meaning of ABEND_SVC are product-defined.

   - The TYPE(ABEND_TIMER) parameter is intended to be used by an LU services component, such as one that processes mapped conversation verbs, when it detects or is informed of a condition that requires the conversation to be deallocated without further communications. For example, too much time elapses without receiving any information, or an operator prematurely ends program execution. The specific conditions and the means by which the LU services component detects or is informed of the conditions are product-defined. The specific use and meaning of ABEND_TIMER are also product-defined.

6. The TYPE(LOCAL) parameter is intended to be used by the transaction program in order to complete the program's deallocation of the conversation after receiving an indication that the conversation has been deallocated from the session, an indication such as a DEALLOCATE_NORMAL or RESOURCE_FAILURE_RETRY return code.

7. The remote transaction program receives the deallocate notification by means of a return code or what-received indication, as follows:

   - DEALLOCATE_NORMAL return code: The local program specified either TYPE(FLUSH); TYPE(SYNC_LEVEL) and the synchronization level is NONE; or TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the local program subsequently issued FLUSH.

   - CONFIRM_DEALLOCATE what-received indication: The local program specified either TYPE(CONFIRM); TYPE(SYNC_LEVEL) and the synchronization level is CONFIRM; or TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the local program subsequently issued CONFIRM.

   - TAKE_SYNCPT_DEALLOCATE what-received indication: The local program specified TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the local program subsequently issued SYNCPT.

   - DEALLOCATE_ABEND_PROG, DEALLOCATE_ABEND_SVC, or DEALLOCATE_ABEND_TIMER return code: The local program specified, respectively, TYPE(ABEND_PROG), TYPE(ABEND_SVC), or TYPE(ABEND_TIMER), with the following exception: If the remote program has issued a SEND_ERROR in receive state, a DEALLOCATE_NORMAL return code is reported instead of one of the DEALLOCATE_ABEND return codes.

8. DEALLOCATE with TYPE(ABEND_PROG), TYPE(ABEND_SVC), or TYPE(ABEND_TIMER) resets or cancels posting. If posting is active and the conversation has been posted, posting is reset. If posting is active and the conversation has not been posted, post-

ing is canceled (posting will not occur). See the POST_ON_RECEIPT verb for more details about posting.

9. References in this verb description to a program being in a particular state are only in terms of the specified conversation.

FLUSH

Flushes the local LU's send buffer. The LU sends any information it has buffered to the remote LU. Information the LU buffers can come from ALLOCATE, DEALLOCATE, SEND_DATA, PREPARE_TO_RECEIVE, or SEND_ERROR. Refer to the descriptions of these verbs for details of the information the LU buffers and when buffering occurs.

FLUSH	Supplied Parameters: RESOURCE (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

State Changes:

Receive state is entered when the verb is issued in defer state following PREPARE_TO_RECEIVE.

Reset state is entered when the verb is issued in defer state following DEALLOCATE.

No state change occurs when the verb is issued in send state.

ABEND Conditions:

Parameter Check

- This verb is not supported.
- RESOURCE specifies an unassigned resource ID.

State Check

The conversation is not in send or defer state.

Notes:

1. This verb is useful for optimization of processing between the local and remote programs. The LU normally buffers the data from consecutive SEND_DATAs until it has a sufficient amount for transmission. At that time it transmits the buffered data. However, the local program can issue FLUSH in order to cause the LU to transmit the buffered data. In this way, the local program can minimize the delay in the remote program's processing of the data.

2. This verb can be issued after DEALLOCATE with TYPE(SYNC_LEVEL) when the synchronization level for the conversation is SYNCPT. The effect to the remote program is the same as issuing DEALLOCATE with TYPE(FLUSH). The conversation is deallocated at the completion of the FLUSH verb.

3. This verb can be issued after PREPARE_TO_RECEIVE with TYPE(SYNC_LEVEL) when the synchronization level for the conversation is SYNCPT. The effect to the remote program is the same as issuing PREPARE_TO_RECEIVE with TYPE(FLUSH). The conversation enters receive state at the completion of the FLUSH verb.

4. The LU flushes its send buffer only when it has some information to transmit. If the LU has no information in its send buffer, nothing is transmitted to the remote LU.

5. References in this verb description to a program being in a particular state are only in terms of the specified conversation.

GET_ATTRIBUTES

Returns information pertaining to the specified conversation.

GET_ATTRIBUTES	**Supplied Parameters:** RESOURCE (variable)
	Returned Parameters: [OWN_FULLY_QUALIFIED_LU_NAME (variable)] [PARTNER_LU_NAME (variable)] [PARTNER_FULLY_QUALIFIED_LU_NAME (variable)] [MODE_NAME (variable)] [SYNC_LEVEL (variable)] [SECURITY_USER_ID (variable)] [SECURITY_PROFILE (variable)] [LUW_IDENTIFIER (variable)] [CONVERSATION_CORRELATOR (variable)]
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID of the conversation of which the attributes are desired.

Returned Parameters:

OWN_FULLY_QUALIFIED_LU_NAME specifies the variable for returning the fully qualified name of the LU at which the local transaction program is located. If the local fully qualified LU name is not known, a null value is returned.

PARTNER_LU_NAME specifies the variable for returning the name of the LU at which the remote transaction program is located. This is a name by which the local LU knows the remote LU for the purpose of allocating a conversation. Refer to the description of the LU_NAME parameter of ALLOCATE for more details.

PARTNER_FULLY_QUALIFIED_LU_NAME specifies the variable for returning the fully qualified name of the LU at which the remote transaction program is located. If the partner's fully qualified LU name is not known, a null value is returned.

MODE_NAME specifies the variable for returning the mode name for the session on which the conversation is allocated.

SYNC_LEVEL specifies the variable for returning the level of synchronization processing being used for the conversation. The synchronization levels are:

- NONE
- CONFIRM

- SYNCPT

SECURITY_USER_ID specifies the variable for returning the user ID carried on the allocation request that initiated execution of the local program. A null value is returned if the allocation request did not contain a user ID.

SECURITY_PROFILE specifies the variable for returning the profile carried on the allocation request that initiated execution of the local program. A null value is returned if the allocation request did not contain a profile.

LUW_IDENTIFIER specifies the variable for returning the logical unit of work (LUW) identifier associated with the conversation. The LUW identifier is created and maintained by the LU. The LU uses it to identify the most recent sync point and for accounting purposes. and for accounting purposes. If no LUW identifier is used on the conversation, a null value is returned.

CONVERSATION_CORRELATOR specifies the variable for returning the conversation correlator. The conversation correlator is created and maintained by the LU. The LU uses it during sync point resynchronization. If no conversation correlator is used on the conversation, a null value is returned.

State Changes:

None

ABEND Conditions:

Parameter Check

- RESOURCE specifies an unassigned resource ID.
- SECURITY_USER_ID is specified and not supported.
- SECURITY_PROFILE is specified and not supported.
- LUW_IDENTIFIER is specified and not supported.
- CONVERSATION_CORRELATOR is specified and not supported.

State Check

None

Notes:

1. The program issues this verb in order to obtain the attributes of the conversation, including the one by which the program was started.

2. Specifying SECURITY_USER_ID or SECURITY_PROFILE returns the user ID or profile carried on the allocation request that initiated execution of the local program, regardless of which resource ID is supplied on the RESOURCE parameter.

3. The LU creates the LUW identifier for its use during sync point processing, and for accounting purposes. For sync point, the LUW identifier uniquely identifies the most recent synchronization point.

4. The LU creates the conversation correlator for its use during sync point resynchronization. For sync point resynchronization, the conversation correlator correlates the logical unit of work to the sync point states associated with the current instance of the local program.

POST_ON_RECEIPT

Causes the LU to post the specified conversation when information is available for the program to receive. The information can be data, conversation status, or a request for confirmation or sync point. WAIT should be issued after POST_ON_RECEIPT in order to wait for posting to occur. Alternatively, TEST may be issued following POST_ON_RECEIPT in order to determine when posting has occurred.

POST_ON_RECEIPT	Supplied Parameters: RESOURCE (variable) [FILL (LL)] [(BUFFER)] LENGTH (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

FILL specifies whether posting for data is to occur in terms of the logical-record format of the data.

- **LL** specifies to post when a complete or truncated logical record is received, or when a part of a logical record is received that is at least equal in length to that specified on the LENGTH parameter, whichever occurs first.
- **BUFFER** specifies to post when data (independent of its logical-record format) is available that is at least equal in length to that specified by the LENGTH parameter, or when the end of data is available, whichever occurs first.

The specification and effect of FILL(LL) versus FILL(BUFFER) is relevant only at the time the verb is issued. The specification does not depend on past use, and has no bearing on subsequent use, of this parameter on any verbs to which it applies (POST_ON_RECEIPT, RECEIVE_IMMEDIATE, and RECEIVE_AND_WAIT).

Posting also occurs independent of the FILL specification when information other than data is received, such as conversation status (a SEND, PROG_ERROR_TRUNC, or DEALLOCATE_NORMAL indication, for example), or a confirmation or sync-point request.

LENGTH specifies the variable containing a length value, which is the maximum length of data that the program can receive. This parameter is used along with FILL to determine when to post the conversation for the receipt of data.

State Changes:

None

ABEND Conditions:

Parameter Check

- This verb is not supported.
- RESOURCE specifies an unassigned resource ID.

State Check

The conversation is not in receive state.

Notes:

1. This verb is intended to be used in conjunction with TEST or WAIT. The use of this verb and WAIT allows a program to perform synchronous receiving from multiple conversations, where the program issues this verb for each of the conversations and then issues WAIT (for each conversation) to wait until information is available to be received on the conversations. The use of this verb and TEST allows a program to continue its processing and test the conversations to determine when information is available to be received.

2. Posting occurs when the LU has any information that the program can receive, such as data, conversation status, or a request for confirmation or sync point. Refer to the RECEIVE_AND_WAIT verb for a description of the types of information a program can receive.

3. Posting is active for a conversation when POST_ON_RECEIPT has been issued for the conversation and posting has not yet been reset or cancelled.

   Posting is reset when any of the following verbs is issued for the same conversation as specified on POST_ON_RECEIPT <u>after</u> the conversation is posted:

   BACKOUT

   DEALLOCATE with TYPE(ABEND_PROG), TYPE(ABEND_SVC), or TYPE(ABEND_TIMER)

   RECEIVE_AND_WAIT

   RECEIVE_IMMEDIATE

   SEND_ERROR

   TEST

   WAIT

   Posting is cancelled when any of the following verbs is issued for the same conversation as specified on POST_ON_RECEIPT <u>before</u> the conversation is posted:

   BACKOUT

   DEALLOCATE with TYPE(ABEND_PROG), TYPE(ABEND_SVC), or TYPE(ABEND_TIMER)

   RECEIVE_IMMEDIATE

   SEND_ERROR

   In order for the program to activate posting again after posting has been reset or cancelled, the program issues another POST_ON_RECEIPT.

4. Any number of POST_ON_RECEIPTs may be issued for a given conversation before posting is reset or cancelled. The last POST_ON_RECEIPT issued for a conversation is the one that determines when posting will occur for data. For example, if a program issues POST_ON_RECEIPT with FILL(BUFFER) and LENGTH(1000) in preparation to receive 1000 bytes of data, and then issues the verb again with LENGTH(500), posting will occur when 500 bytes of data are available; or if the program issues the verb again with FILL(LL), posting will occur in terms of logical records.

5. POST_ON_RECEIPT with LENGTH(0) has no special significance. It specifies that posting for data is to occur upon receipt of any amount of data of one byte or more. It is equivalent to POST_ON_RECEIPT with LENGTH(1).

6. When FILL(BUFFER) is specified, posting for data occurs independent of its logical record format. The conversation is posted when an amount of data is available that is equal to, or less than, the length specified on the LENGTH parameter. Posting for less data can occur only when the end of the data is available. The end of data occurs when it is followed by an indication of a change in the state of the conversation, that is, a change to send, confirm, sync-point, or deallocate state. See RECEIVE_AND_WAIT for additional information.

7. References in this verb description to a program being in a particular state are only in terms of the specified conversation.

PREPARE_TO_RECEIVE

Changes the conversation from send to receive state in preparation to receive data. The change to receive state can be either completed as part of this verb, or deferred until the program issues a FLUSH, CONFIRM, or SYNCPT verb. When it is completed as part of this verb it includes the function of the FLUSH or CONFIRM verb.

PREPARE_TO_RECEIVE	**Supplied Parameters:** RESOURCE (variable) [TYPE (SYNC_LEVEL) / (FLUSH) / (CONFIRM)] [LOCKS (SHORT) / (LONG)]
	Returned Parameters: RETURN_CODE (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

TYPE specifies the type of prepare-to-receive to be performed for this conversation.

- **SYNC_LEVEL** specifies to perform the prepare-to-receive based on the synchronization level allocated to this conversation:
  - If SYNC_LEVEL(NONE), execute the function of the FLUSH verb and enter receive state.
  - If SYNC_LEVEL(CONFIRM), execute the function of the CONFIRM verb and if it is successful (as indicated by a return code of OK on this PREPARE_TO_RECEIVE verb), enter receive state; if it is not successful, the state of the conversation is determined by the return code.
  - If SYNC_LEVEL(SYNCPT), enter defer state until the program issues a SYNCPT, or the program issues a CONFIRM or FLUSH for this conversation. If the SYNCPT or CONFIRM is successful (as indicated by a return code of OK on that verb) or FLUSH is issued, receive state is then entered for this conversation; otherwise, the state of the conversation is determined by the return code.
- **FLUSH** specifies to execute the function of the FLUSH verb and enter receive state.
- **CONFIRM** specifies to execute the function of the CONFIRM verb and if it is successful (as indicated by a return code of OK on this PREPARE_TO_RECEIVE verb), enter receive state; if it is not successful, the state of the conversation is determined by the return code.

The execution of the FLUSH or CONFIRM function as part of this verb includes the flushing of the LU's send buffer. When, instead, defer state is entered, the LU defers flushing its send buffer until the program issues a subsequent verb for this conversation.

LOCKS specifies when control is to be returned to the local program following execution of the CONFIRM function of this verb or following execution of a CONFIRM verb issued subsequent to this verb. This

parameter is significant only when TYPE(CONFIRM) is also specified, or when TYPE(SYNC_LEVEL) is also specified and the synchronization level for this conversation is CONFIRM; or when TYPE(SYNC_LEVEL) is also specified, the synchronization level for this conversation is SYNCPT, and a subsequent CONFIRM is issued. Otherwise, this parameter has no meaning and is ignored.

- **SHORT** specifies to return control when an affirmative reply is received, as follows:
  - When the synchronization level is CONFIRM, return control from execution of this verb when a CONFIRMED reply is received.
  - When the synchronization level is SYNCPT, return control immediately from execution of this verb; return control from execution of a subsequent CONFIRM or SYNCPT verb when a corresponding CONFIRMED or SYNCPT reply is received.
- **LONG** specifies to return control when information, such as data, is received from the remote program following an affirmative reply, as follows:
  - When the synchronization level is CONFIRM, return control from execution of this verb when information is received following a CONFIRMED reply.
  - When the synchronization level is SYNCPT, return control immediately from execution of this verb; return control from execution of a subsequent CONFIRM or SYNCPT verb when information is received following a corresponding CONFIRMED or SYNCPT reply.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the local program. The return code indicates the result of verb execution. The TYPE parameter determines which of the following return codes can be returned to the program.

- If TYPE(FLUSH) is specified, or if TYPE(SYNC_LEVEL) is specified and the synchronization level allocated to this conversation is NONE, the following return code is returned:
  - OK
- If TYPE(SYNC_LEVEL) is specified and the synchronization level allocated to this conversation is CONFIRM, or TYPE(CONFIRM) is specified, one of the following return codes is returned:
  - OK
  - ALLOCATION_ERROR
  - DEALLOCATE_ABEND_PROG
  - DEALLOCATE_ABEND_SVC
  - DEALLOCATE_ABEND_TIMER
  - PROG_ERROR_PURGING
  - SVC_ERROR_PURGING
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY
- If TYPE(SYNC_LEVEL) is specified and the synchronization level allocated to this conversation is SYNCPT, the following return code is returned:
  - OK

State Changes (When RETURN_CODE indicates OK):

Defer state is entered when TYPE(SYNC_LEVEL) is specified and the synchronization level is SYNCPT.

Receive state is entered when TYPE(FLUSH) or TYPE(CONFIRM) is specified, or when TYPE(SYNC_LEVEL) is specified and the synchronization level is NONE or CONFIRM.

ABEND Conditions:

Parameter Check

- This verb is not supported.
- RESOURCE specifies an unassigned resource ID.
- TYPE(CONFIRM) is specified and the conversation is allocated with SYNC_LEVEL(NONE).
- LOCKS(LONG) is specified and not supported.

State Check

- The conversation is not in send state.
- The conversation is in send state, and the program started but did not finish sending a logical record.

Notes:

1. The TYPE(SYNC_LEVEL) parameter is intended to be used by the transaction program in order to transfer send control to the remote program based on the synchronization level allocated to the conversation.

   - If the synchronization level is NONE, send control is transferred to the remote program without any synchronizing acknowledgment.
   - If the synchronization level is CONFIRM, send control is transferred to the remote program with confirmation requested.
   - If the synchronization level is SYNCPT, transfer of send control is deferred. When the local program subsequently issues SYNCPT, send control is transferred to the remote program with sync point requested.

2. The TYPE(FLUSH) parameter is intended to be used by the transaction program in order to transfer send control to the remote program without any synchronizing acknowledgment. TYPE(FLUSH) is functionally equivalent to:

   - TYPE(SYNC_LEVEL) with a synchronization level of NONE.
   - TYPE(SYNC_LEVEL) with a synchronization level of SYNCPT, followed by the FLUSH verb.

3. The TYPE(CONFIRM) parameter is intended to be used by the transaction program in order to transfer send control to the remote program with confirmation requested. TYPE(CONFIRM) is functionally equivalent to:

   - TYPE(SYNC_LEVEL) with a synchronization level of CONFIRM.
   - TYPE(SYNC_LEVEL) with a synchronization level of SYNCPT, followed by the CONFIRM verb.

4. The remote transaction program receives send control by means of a what-received indication of SEND, CONFIRM_SEND, or TAKE_SYNCPT_SEND, as follows:

   - SEND: The local program specified either TYPE(FLUSH); TYPE(SYNC_LEVEL) and the synchronization level is NONE; or TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the local program subsequently issued FLUSH.
   - CONFIRM_SEND: The local program specified either TYPE(CONFIRM); TYPE(SYNC_LEVEL) and the synchronization level is CONFIRM; or TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the local program subsequently issued CONFIRM.
   - TAKE_SYNCPT_SEND: The local program specified TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the local program subsequently issued SYNCPT.

5. If TYPE(SYNC_LEVEL) is specified and the synchronization level for the conversation is SYNCPT, the LU buffers the SEND notification to be sent to the remote program until the local program issues a verb that causes the LU to flush its send buffer.

6. The conversation for the remote program enters the corresponding send state when it issues a RECEIVE_AND_WAIT or RECEIVE_IMMEDIATE verb and receives the SEND indication (on the WHAT_RECEIVED parameter.) The remote program can then send data to the local program.

7. References in this verb description to a program being in a particular state are only in terms of the specified conversation.

RECEIVE_AND_WAIT

Waits for information to arrive on the specified conversation and then receives the information. If information is already available, the program receives it without waiting. The information can be data, conversation status, or a request for confirmation or sync point. Control is returned to the program with an indication of the type of information.

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.

RECEIVE_AND_WAIT	Supplied Parameters: RESOURCE (variable) [FILL (LL)] [(BUFFER)]
	Supplied-and-Returned Parameters: LENGTH (variable)
	Returned Parameters: RETURN_CODE (variable) REQUEST_TO_SEND_RECEIVED (variable) DATA (variable) WHAT_RECEIVED (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

FILL specifies whether the program is to receive data in terms of the logical-record format of the data.

- **LL** specifies the program is to receive one complete or truncated logical record, or a portion of the logical record that is equal to the length specified by the LENGTH parameter.
- **BUFFER** specifies the program is to receive data independent of its logical-record format. The amount of data received will be equal to, or less than, the length specified by the LENGTH parameter. The amount is less than the specified length when the program receives the end of the data.

The specification and effect of FILL(LL) versus FILL(BUFFER) is relevant only at the time the verb is issued. The specification does not depend on past use, and has no bearing on subsequent use, of this parameter on any verbs to which it applies (POST_ON_RECEIPT, RECEIVE_AND_WAIT, and RECEIVE_IMMEDIATE).

Supplied-and-Returned Parameters:

LENGTH specifies the variable containing a length value that is the maximum amount of data the program is to receive. When control is returned to the program this variable contains the actual amount of data the program received up to the maximum. If the program receives information other than data, this variable remains unchanged.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the program. The return code indicates the result of verb execution. The return codes that can be returned depend on the state of the conversation at the time this verb is issued.

- If this verb is issued in send state, the following return codes can be returned:

  - OK
  - ALLOCATION_ERROR
  - BACKED_OUT
  - DEALLOCATE_ABEND_PROG
  - DEALLOCATE_ABEND_SVC
  - DEALLOCATE_ABEND_TIMER
  - PROG_ERROR_PURGING
  - SVC_ERROR_PURGING
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY

- If this verb is issued in receive state, the following return codes can be returned:

  - OK
  - ALLOCATION_ERROR
  - BACKED_OUT
  - DEALLOCATE_ABEND_PROG
  - DEALLOCATE_ABEND_SVC
  - DEALLOCATE_ABEND_TIMER
  - DEALLOCATE_NORMAL
  - PROG_ERROR_NO_TRUNC
  - PROG_ERROR_PURGING
  - PROG_ERROR_TRUNC
  - SVC_ERROR_NO_TRUNC
  - SVC_ERROR_PURGING
  - SVC_ERROR_TRUNC
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received. The indication is either YES or NO.

- YES indicates a REQUEST_TO_SEND notification has been received from the remote transaction program. The remote program has issued REQUEST_TO_SEND, requesting the local program to enter receive state and thereby place the remote program in send state.

- NO indicates a REQUEST_TO_SEND notification has not been received.

DATA specifies the variable in which the program is to receive the data. When the program receives information other than data, as indicated by the WHAT_RECEIVED parameter, nothing is placed in this variable.

WHAT_RECEIVED specifies the variable in which is returned an indication of what the transaction program received. The program should examine this variable only when RETURN_CODE indicates OK; otherwise, nothing is placed in this variable.

- DATA is indicated when FILL(BUFFER) is specified and data (independent of its logical-record format) is received by the program.

- DATA_COMPLETE is indicated when FILL(LL) is specified and a complete logical record, or the last remaining portion thereof, is received by the program.

- DATA_INCOMPLETE is indicated when FILL(LL) is specified and less than a complete logical record is received by the program. The transaction program issues another RECEIVE_AND_WAIT (or possibly multiple RECEIVE_AND_WAITs) to receive the remainder of the data.

- LL_TRUNCATED is indicated when FILL(LL) is specified and the 2-byte LL field of a logical record has been truncated after the first byte. The LU discards the truncated LL field; it is not received by the program.

- SEND indicates the remote program has entered receive state, placing the local program in send state. The local program may now issue SEND_DATA.

- CONFIRM indicates the remote program has issued CONFIRM, requesting the local program to respond by issuing CONFIRMED. The program may respond, instead, by issuing a verb other than CONFIRMED, such as SEND_ERROR.

- CONFIRM_SEND indicates the remote program has issued PREPARE_TO_RECEIVE with TYPE(CONFIRM); or with TYPE(SYNC_LEVEL), and either the synchronization level is CONFIRM, or it is SYNCPT and the remote program subsequently issued CONFIRM. The local program may respond by issuing CONFIRMED, or by issuing another verb such as SEND_ERROR.

- CONFIRM_DEALLOCATE indicates the remote program has issued DEALLOCATE with TYPE(CONFIRM); or with TYPE(SYNC_LEVEL), and either the synchronization level is CONFIRM, or it is SYNCPT and the remote program subsequently issued CONFIRM. The local program may respond by issuing CONFIRMED, or by issuing another verb such as SEND_ERROR.

- TAKE_SYNCPT indicates the remote program has issued SYNCPT, requesting the local program to respond by issuing SYNCPT in order to perform the sync-point function on all protected resources throughout the transaction. Issuing the SYNCPT verb also causes an affirmative reply to be returned to the remote program if the sync-point function is successful. The program may respond, instead, by issuing a verb other than SYNCPT, such as BACKOUT or SEND_ERROR.

- TAKE_SYNCPT_SEND indicates the remote program has issued PREPARE_TO_RECEIVE with TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the remote program subsequently issued SYNCPT. The local program may respond by issuing SYNCPT, or by issuing another verb such as BACKOUT or SEND_ERROR.

- TAKE_SYNCPT_DEALLOCATE indicates the remote program has issued DEALLOCATE with TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the remote program subsequently issued SYNCPT. The local program may respond by issuing SYNCPT, or by issuing another verb such as BACKOUT or SEND_ERROR.

State Changes (when RETURN_CODE indicates OK):

Receive state is entered when the verb is issued in send state and WHAT_RECEIVED indicates DATA, DATA_COMPLETE, DATA_INCOMPLETE, or LL_TRUNCATED.

Send state is entered when WHAT_RECEIVED indicates SEND.

Confirm state is entered when WHAT_RECEIVED indicates CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE.

Sync-point state is entered when WHAT_RECEIVED indicates TAKE_SYNCPT, TAKE_SYNCPT_SEND, or TAKE_SYNCPT_DEALLOCATE.

No state change occurs when the verb is issued in receive state and WHAT_RECEIVED indicates DATA, DATA_COMPLETE, DATA_INCOMPLETE, or LL_TRUNCATED.

ABEND Conditions:

Parameter Check

RESOURCE specifies an unassigned resource ID.

State Check

- The conversation is not in send or receive state.
- The conversation is in send state, and the program started but did not finish sending a logical record.

Notes:

1. When the program issues RECEIVE_AND_WAIT in send state, the LU implicitly executes a PREPARE_TO_RECEIVE with TYPE(FLUSH) before executing the RECEIVE_AND_WAIT. Refer to the description of PREPARE_TO_RECEIVE for details of its function.

2. When FILL(LL) is specified, the program is to receive a logical record and there are the following possibilities:

   - The program receives a complete logical record or the last remaining portion of a complete record. The length of the record or portion of the record is equal to or less than the length specified on the LENGTH parameter. The WHAT_RECEIVED parameter indicates DATA_COMPLETE.

   - The program receives an incomplete logical record. The logical record is incomplete because:

     - The length of the logical record is greater than the length specified on the LENGTH parameter; in this case the amount received equals the length specified.

     - Only a portion of the logical record is available because it has been truncated, the portion being equal to or less than the length specified on the LENGTH parameter.

     The WHAT_RECEIVED parameter indicates DATA_INCOMPLETE. The program issues another RECEIVE_AND_WAIT (or possibly multiple RECEIVE_AND_WAITs) to receive the remainder of the logical record.

   - The program receives no part of the logical record because it was truncated after the first byte of the LL field. The WHAT_RECEIVED parameter indicates LL_TRUNCATED.

   Refer to the SEND_DATA verb for a definition of complete and incomplete logical records.

3. When FILL(BUFFER) is specified, the program is to receive data independent of its logical-record format. The program receives an amount of data equal to, or less than, the length specified on the LENGTH parameter. The program can receive less data only when it receives the end of the data. The end of data occurs when it is followed by an indication of a change in the state of the conversation, that is, a change to send, confirm, sync-point, or deallocate state. The program is responsible for tracking the logical-record format of the data.

4. RECEIVE_AND_WAIT with LENGTH(0) has no special significance. The type of information available is indicated by the RETURN_CODE and WHAT_RECEIVED parameters, as usual. If data is available and FILL(LL) is specified, the WHAT_RECEIVED parameter indicates DATA_INCOMPLETE. If data is available and FILL(BUFFER) is specified, the WHAT_RECEIVED parameter indicates DATA. In either case, however, the program receives no data.

5. The program receives only one kind of information at a time. For example, it may receive data or a CONFIRM request, but it does not receive both at the same time. Also, if the remote program truncates a logical record, the local program receives the indication of the truncation on the RECEIVE_AND_WAIT it issues after receiving all of the truncated record. The RETURN_CODE and WHAT_RECEIVED parameters indicate to the program the kind of information the program receives.

6. RECEIVE_AND_WAIT includes posting. If posting is already active when this verb is issued, this verb supersedes the prior POST_ON_RECEIPT function. Posting is reset at the completion of this verb. See the POST_ON_RECEIPT verb for more details about posting.

7. The REQUEST_TO_SEND notification is usually received when the local transaction program is in send state, and reported to the program on a SEND_DATA verb or on a SEND_ERROR verb issued in send state. However, the notification can be received when the program is in receive state under the following conditions:

   - When the local program just entered receive state and the remote program issued REQUEST_TO_SEND before it entered send state.

   - When the remote program has just entered receive state by means of the PREPARE_TO_RECEIVE verb (not RECEIVE_AND_WAIT), and then issued REQUEST_TO_SEND before the local program enters send state. This can occur because the REQUEST_TO_SEND is transmitted as an expedited request and can therefore arrive ahead of the request carrying the SEND indication. Potentially, the local program cannot distinguish this condition from the first. This ambiguity is avoided when the remote program waits until it receives information from the local program before it issues the REQUEST_TO_SEND.

   - When the remote program issues the REQUEST_TO_SEND in send state (see "Notes on Implementation Details" in Appendix A).

8. The REQUEST_TO_SEND notification is returned to the program in addition to (not in place of) the information indicated by the RETURN_CODE and WHAT_RECEIVED parameters.

9. References in this verb description to the program being in a particular state are only in terms of the specified conversation.

RECEIVE_IMMEDIATE

Receives any information that is available from the specified conversation, but does not wait for information to arrive. The information (if any) can be data, conversation status, or a request for confirmation or sync point. Control is returned to the program with an indication of whether any information was received and, if so, the type of information.

RECEIVE_IMMEDIATE	**Supplied Parameters:** RESOURCE (variable) [FILL (LL) (BUFFER)]
	Supplied-and-Returned Parameters: LENGTH (variable)
	Returned Parameters: RETURN_CODE (variable) REQUEST_TO_SEND_RECEIVED (variable) DATA (variable) WHAT_RECEIVED (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

FILL specifies whether the program is to receive data in terms of the logical-record format of the data.

- **LL** specifies the program is to receive one logical record, or whatever portion of the logical record is available, up to the length specified by the LENGTH parameter.
- **BUFFER** specifies the program is to receive data independent of its logical-record format, up to the length specified by the LENGTH parameter.

The specification and effect of FILL(LL) versus FILL(BUFFER) is relevant only at the time the verb is issued. The specification does not depend on past use, and has no bearing on subsequent use, of this parameter on any verbs to which it applies (POST_ON_RECEIPT, RECEIVE_IMMEDIATE, and RECEIVE_AND_WAIT).

Supplied-and-Returned Parameters:

LENGTH specifies the variable containing a length value that is the maximum amount of data the program is to receive. When control is returned to the program this variable contains the actual amount of data the program received up to the maximum. If the program receives information other than data, or no information at all, this variable remains unchanged.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the program. The return code indicates the result of verb execution.

- OK
- ALLOCATION_ERROR

- BACKED_OUT
- DEALLOCATE_ABEND_PROG
- DEALLOCATE_ABEND_SVC
- DEALLOCATE_ABEND_TIMER
- DEALLOCATE_NORMAL
- PROG_ERROR_NO_TRUNC
- PROG_ERROR_PURGING
- PROG_ERROR_TRUNC
- SVC_ERROR_NO_TRUNC
- SVC_ERROR_PURGING
- SVC_ERROR_TRUNC
- RESOURCE_FAILURE_NO_RETRY
- RESOURCE_FAILURE_RETRY
- UNSUCCESSFUL - There is nothing to receive.

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received. The indication is either YES or NO.

- YES indicates a REQUEST_TO_SEND notification has been received from the remote program. The remote program has issued REQUEST_TO_SEND, requesting the local program to enter receive state and thereby place the remote program in send state.

- NO indicates a REQUEST_TO_SEND notification has not been received.

DATA specifies the variable in which the program is to receive the data. When the program receives information other than data, as indicated by the WHAT_RECEIVED parameter, nothing is placed in this variable.

WHAT_RECEIVED specifies the variable in which is returned an indication of what the transaction program received. The program should examine this variable only when RETURN_CODE indicates OK; otherwise, nothing is placed in this variable.

- DATA is indicated when FILL(BUFFER) is specified and data (independent of its logical-record format) is received by the program.

- DATA_COMPLETE is indicated when FILL(LL) is specified and a complete logical record, or the last remaining portion thereof, is received by the program.

- DATA_INCOMPLETE is indicated when FILL(LL) is specified and less than a complete logical record is received by the program. The transaction program issues another RECEIVE_IMMEDIATE (or possibly multiple RECEIVE_IMMEDIATEs) to receive the remainder of the logical record.

- LL_TRUNCATED is indicated when FILL(LL) is specified and the 2-byte LL field of a logical record has been truncated after the first byte. The LU discards the truncated LL field; it is not received by the program.

- SEND indicates the remote program has entered receive state, placing the local program in send state. The local program may now issue SEND_DATA.

- CONFIRM indicates the remote program has issued CONFIRM, requesting the local program to respond by issuing CONFIRMED. The program may respond, instead, by issuing a verb other than CONFIRMED, such as SEND_ERROR.

- CONFIRM_SEND indicates the remote program has issued PREPARE_TO_RECEIVE with TYPE(CONFIRM); or with TYPE(SYNC_LEVEL), and either the synchronization level is CONFIRM, or it is SYNCPT and the remote program subsequently issued CONFIRM. The local program may respond by issuing CONFIRMED, or by issuing another verb such as SEND_ERROR.

- CONFIRM_DEALLOCATE indicates the remote program has issued DEALLOCATE with TYPE(CONFIRM); or with TYPE(SYNC_LEVEL), and either the synchronization level is CONFIRM, or it is SYNCPT and the

remote program subsequently issued CONFIRM. The local program may respond by issuing CONFIRMED, or by issuing another verb such as SEND_ERROR.

- TAKE_SYNCPT indicates the remote program has issued SYNCPT, requesting the local program to respond by issuing SYNCPT in order to perform the sync-point function on all protected resources throughout the transaction. Issuing the SYNCPT verb also causes an affirmative reply to be returned to the remote program if the sync-point function is successful. The program may respond, instead, by issuing a verb other than SYNCPT, such as BACKOUT or SEND_ERROR.

- TAKE_SYNCPT_SEND indicates the remote program has issued PREPARE_TO_RECEIVE with TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the remote program subsequently issued SYNCPT. The local program may respond by issuing SYNCPT, or by issuing another verb such as BACKOUT or SEND_ERROR.

- TAKE_SYNCPT_DEALLOCATE indicates the remote program has issued DEALLOCATE with TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the remote program subsequently issued SYNCPT. The local program may respond by issuing SYNCPT, or by issuing another verb such as BACKOUT or SEND_ERROR.

State Changes (When RETURN CODE indicates OK):

Send state is entered when WHAT_RECEIVED indicates SEND.

Confirm state is entered when WHAT_RECEIVED indicates CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE.

Sync-point state is entered when WHAT_RECEIVED indicates TAKE_SYNCPT, TAKE_SYNCPT_SEND, or TAKE_SYNCPT_DEALLOCATE.

No state change occurs when WHAT_RECEIVED indicates DATA, DATA_COMPLETE, DATA_INCOMPLETE, or LL_TRUNCATED.

ABEND Conditions:

Parameter Check

- This verb is not supported.
- RESOURCE specifies an unassigned resource ID.

State Check

The conversation is not in receive state.

Notes:

1. When FILL(LL) is specified, the program is to receive a logical record and there are the following possibilities:

   - The program receives a complete logical record or the last remaining portion of a complete record. The length of the record or portion of the record is equal to or less than the length specified on the LENGTH parameter. The WHAT_RECEIVED parameter indicates DATA_COMPLETE.

   - The program receives an incomplete logical record. The logical record is incomplete because:

     - The length of the logical record is greater than the length specified on the LENGTH parameter; in this case the amount received equals the length specified.

     - Only a portion of the logical record is available (possibly because it has been truncated), the portion being equal to or less than the length specified on the LENGTH parameter.

     The WHAT_RECEIVED parameter indicates DATA_INCOMPLETE. The program issues another RECEIVE_IMMEDIATE (or possibly multi-

ple RECEIVE_IMMEDIATEs) to receive the remainder of the logical record.

- The program receives no part of the logical record because it was truncated after the first byte of the LL field. The WHAT_RECEIVED parameter indicates LL_TRUNCATED.

Refer to the SEND_DATA verb for a definition of complete and incomplete logical records.

2. When FILL(BUFFER) is specified, the program is to receive data independent of its logical-record format. The program receives whatever data is available, up to the amount specified on the LENGTH parameter. The program is responsible for tracking the logical-record format of the data.

3. RECEIVE_IMMEDIATE with LENGTH(0) has no special significance. The type of information available, if any, is indicated by the RETURN_CODE and WHAT_RECEIVED parameters, as usual. If data is available and FILL(LL) is specified, the WHAT_RECEIVED parameter indicates DATA_INCOMPLETE. If data is available and FILL(BUFFER) is specified, the WHAT_RECEIVED parameter indicates DATA. In either case, however, the program receives no data.

4. The program receives only one kind of information at a time. For example, it may receive data or a CONFIRM request, but it does not receive both at the same time. Also, if the remote program truncates a logical record, the local program receives the indication of the truncation on the RECEIVE_IMMEDIATE it issues after receiving all of the truncated record. The RETURN_CODE and WHAT_RECEIVED parameters indicate to the program the kind of information the program receives, if any.

5. RECEIVE_IMMEDIATE resets or cancels posting. If posting is active and the conversation has been posted, posting is reset. If posting is active and the conversation has not been posted, posting is canceled (posting will not occur). See the POST_ON_RECEIPT verb for more details about posting.

6. The REQUEST_TO_SEND notification is usually received when the local transaction program is in send state, and reported to the program on a SEND_DATA verb or on a SEND_ERROR verb issued in send state. However, the notification can be received when the program is in receive state under the following conditions:

   - When the local program just entered receive state and the remote program issued REQUEST_TO_SEND before it entered send state.

   - When the remote program has just entered receive state by means of the PREPARE_TO_RECEIVE verb (not RECEIVE_AND_WAIT), and then issued REQUEST_TO_SEND before the local program enters send state. This can occur because the REQUEST_TO_SEND is transmitted as an expedited request and can therefore arrive ahead of the request carrying the SEND indication. Potentially, the local program cannot distinguish this case from the first. This ambiguity is avoided when the remote program waits until it receives information from the local program before it issues the REQUEST_TO_SEND.

   - When the remote program issues the REQUEST_TO_SEND in send state (see "Notes on Implementation Details" in Appendix A).

7. The REQUEST_TO_SEND notification is returned to the program in addition to (not in place of) the information indicated by the RETURN_CODE and WHAT_RECEIVED parameters.

8. References in this verb description to a program being in a particular state are only in terms of the specified conversation.

REQUEST_TO_SEND

Notifies the remote program that the local program is requesting to enter send state for the conversation. The conversation will be changed to send state when the local program subsequently receives a SEND indication from the remote program.

REQUEST_TO_SEND	Supplied Parameters: RESOURCE (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

State Changes:

None

ABEND Conditions:

Parameter Check

- RESOURCE specifies an unassigned resource ID.

State Check

- The conversation is not in receive, confirm, or sync-point state.

Notes:

1. The REQUEST_TO_SEND notification is indicated to the remote program by means of the REQUEST_TO_SEND_RECEIVED parameter. When the REQUEST_TO_SEND_RECEIVED parameter is set to YES, the remote program is requested to enter receive state and thereby place the local program in send state. A program enters receive state by means of the RECEIVE_AND_WAIT or PREPARE_TO_RECEIVE verb. The partner program enters the corresponding send state when it issues a RECEIVE_AND_WAIT or RECEIVE_IMMEDIATE verb and receives the SEND indication (on the WHAT_RECEIVED parameter).

2. The REQUEST_TO_SEND_RECEIVED indication of YES is normally returned to the remote program when it is in send state, that is, on a SEND_DATA verb or on a SEND_ERROR verb issued in send state. However, it can be returned on a RECEIVE_AND_WAIT or RECEIVE_IMMEDIATE verb; see the description of RECEIVE_AND_WAIT or RECEIVE_IMMEDIATE for details about when this can occur.

3. When the remote LU receives the REQUEST_TO_SEND notification, it retains the notification until the remote program issues a verb on which the notification can be indicated, that is, a verb with the REQUEST_TO_SEND_RECEIVED parameter. The remote LU will retain only one REQUEST_TO_SEND notification at a time (per conversation); additional notifications are discarded until the retained notification is indicated to the remote program. It is therefore possible for the local program to issue the REQUEST_TO_SEND verb more times than are indicated to the remote program.

4. References in this verb description to a program being in a particular state are only in terms of the specified conversation.

SEND_DATA

Sends data to the remote transaction program. The data format consists of logical records. The amount of data is specified independently of the data format.

SEND_DATA	Supplied Parameters: RESOURCE (variable) DATA (variable) LENGTH (variable)
	Returned Parameters: RETURN_CODE (variable) REQUEST_TO_SEND_RECEIVED (variable)
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID of the conversation on which the data is to be sent.

DATA specifies the variable containing the data to be sent. The data consists of logical records. Each logical record consists of a two-byte length field (denoted as LL) followed by a data field; the length of the data field can range from zero to 32765 bytes. The two-byte length field contains the 15-bit binary length of the record, and a high-order bit that is not examined by the LU (it is used, for example, by the LU's mapped conversation component in support of the mapped conversation verbs). The length of the record includes the two-byte length field, that is, it equals the length of the data field plus two. Thus, logical-record length values of hex[2] 0000, 0001, 8000, and 8001, are invalid.

LENGTH specifies the variable containing the length of the data to be sent. This data length is not related in any way to the length of a logical record. It is used only to determine the length of the data located at the variable specified by the DATA parameter.

The data length may be zero or greater. If zero, no data is sent for this issuance of the verb, and the DATA parameter is not significant. However, the other parameters are significant and retain their meaning as described.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the local program. The return code indicates the result of verb execution.

- OK
- ALLOCATION_ERROR
- PROG_ERROR_PURGING
- SVC_ERROR_PURGING
- DEALLOCATE_ABEND_PROG
- DEALLOCATE_ABEND_SVC
- DEALLOCATE_ABEND_TIMER
- BACKED_OUT
- RESOURCE_FAILURE_NO_RETRY
- RESOURCE_FAILURE_RETRY

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received. The indication is either YES or NO.

---

[2] Hex (meaning hexadecimal) refers to the base-16 numbering system.

- YES indicates a REQUEST_TO_SEND notification has been received from the remote transaction program. The remote program has issued REQUEST_TO_SEND, requesting the local program to enter receive state and thereby place the remote program in send state.
- NO indicates a REQUEST_TO_SEND notification has not been received.

State Changes (when RETURN CODE indicates OK):

None

ABEND Conditions:

Parameter Check

- RESOURCE specifies an unassigned resource ID.
- DATA contains an invalid logical record length (LL) value of hex 0000, 0001, 8000, or 8001.

State Check

The conversation is not in send state.

Notes:

1. The data sent by the program consists of logical records. The logical records are independent of the length of data as specified by the LENGTH parameter. That is, the data may consist of one or more complete records, the beginning of a record, the middle of a record, or the end of a record. The following combinations of these are also possible:
   - One or more complete records, followed by the beginning of a record.
   - The end of a record, followed by one or more complete records.
   - The end of a record, followed by one or more complete records, followed by the beginning of a record.
   - The end of a record, followed by the beginning of a record.
2. The program must finish sending a logical record before issuing any of the following verbs:

   ```
   CONFIRM
   DEALLOCATE with TYPE(FLUSH), TYPE(CONFIRM), or
     TYPE(SYNC_LEVEL)
   PREPARE_TO_RECEIVE
   RECEIVE_AND_WAIT
   SYNCPT
   ```

   A program finishes sending a logical record when it sends a complete record or when it truncates an incomplete record.
3. A complete logical record contains the two-byte LL field and all bytes of the data field, as determined by the logical-record length. (If the data field is of zero length, the complete logical record contains only the two-byte length field.) An incomplete logical record consists of any amount of data less than a complete record. It can consist of only the first byte of the LL field, the two-byte LL field plus all of the data field except the last byte, or any amount in between. A logical record is incomplete until the last byte of the data field is sent, or until the second byte of the LL field is sent if the data field is of zero length.
4. A program can truncate an incomplete logical record by issuing the SEND_ERROR verb. SEND_ERROR causes the LU to flush its send buffer, which includes sending the truncated record. The LU then treats the first two bytes of data specified in the next SEND_DATA as the LL field. Issuing DEALLOCATE with TYPE(ABEND_PROG),

TYPE(ABEND_SVC), or TYPE(ABEND_TIMER) also truncates an incomplete logical record.

5. The LU buffers the data to be sent to the remote LU until it accumulates from one or more SEND_DATAs a sufficient amount for transmission, or until the local program issues a verb that causes the LU to flush its send buffer. The amount of data that is sufficient for transmission depends on the characteristics of the session allocated for the conversation, and can vary from one session to another.

6. When REQUEST_TO_SEND_RECEIVED indicates YES, the remote program is requesting the local program to enter receive state and thereby place the remote program in send state. A program enters receive state by means of the RECEIVE_AND_WAIT or PREPARE_TO_RECEIVE verb. The partner program enters the corresponding send state when it issues a RECEIVE_AND_WAIT or RECEIVE_IMMEDIATE verb and receives the SEND indication (on the WHAT_RECEIVED parameter).

7. References in this verb description to a program being in a particular state are only in terms of the specified conversation.

SEND_ERROR

Informs the remote transaction program that the local program detected an error. If the conversation is in send state, the LU flushes its send buffer.

Upon successful completion of this verb, the local program is in send state and the remote program is in receive state. Further action is defined by transaction program logic.

SEND_ERROR	**Supplied Parameters:** **RESOURCE (variable)** [**TYPE (PROG)** / **(SVC)**] [**LOG_DATA (NO)** / **(YES (variable))**]
	Returned Parameters: **RETURN_CODE (variable)** **REQUEST_TO_SEND_RECEIVED (variable)**
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

TYPE specifies the level of error—application or service—being reported. This parameter is intended to distinguish between errors to be reported to end-user application transaction programs and errors to be reported to LU services transaction programs.

- **PROG** specifies an end-user application program error is being reported. For instance, this error type is used by the LU services component for mapped conversation verbs in its processing of the MC_SEND_ERROR verb. The corresponding component at the remote LU will pass the error return code on to the remote end-user application program.
- **SVC** specifies an LU services error is being reported. For instance, this error type is used by the LU services component for mapped conversation verbs to report errors detected within the LU services layer.

LOG_DATA specifies whether product-unique error information is to be placed in the system error logs of the LUs supporting this conversation.

- **NO** specifies that no error information is to be placed in the system error logs.
- **YES** specifies that product-unique error information is to be placed in the system error logs of the local and remote LUs. The specified variable contains the product-unique error information, in the format of the Error Log GDS variable. See SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2 for a definition of the Error Log GDS variable.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the local program. The return code indicates the result of verb execution. The return codes that can be returned depend on the state of the conversation at the time this verb is issued:

- If the SEND_ERROR is issued in send state, the following return codes can be returned:

  - OK
  - ALLOCATION_ERROR
  - DEALLOCATE_ABEND_PROG
  - DEALLOCATE_ABEND_SVC
  - DEALLOCATE_ABEND_TIMER
  - PROG_ERROR_PURGING
  - SVC_ERROR_PURGING
  - BACKED_OUT
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY

- If the SEND_ERROR is issued in receive state, the following return codes can be returned:

  - OK
  - DEALLOCATE_NORMAL
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY

- If the SEND_ERROR is issued in confirm state or sync-point state, the following return codes can be returned:

  - OK
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received. The indication is either YES or NO.

- YES indicates a REQUEST_TO_SEND notification has been received from the remote transaction program. The remote program has issued REQUEST_TO_SEND, requesting the local program to enter receive state and thereby place the remote program in send state.

- NO indicates a REQUEST_TO_SEND notification has not been received.

State Changes (When RETURN CODE indicates OK):

Send state is entered when the verb is issued in receive, confirm, or sync-point state.

No state change occurs when the verb is issued in send state.

ABEND Conditions:

Parameter Check

- LOG_DATA is specified and not supported.
- RESOURCE specifies an unassigned resource ID.

State Check

The conversation is not in send, receive, confirm, or sync-point state.

Notes:

1. The LU may send the error notification to the remote LU immediately, that is, during the processing of this verb, or the LU may defer sending the notification until a later time. The determination is made as follows:

   - If the local product does not support the FLUSH verb (see "Notes on Implementation Details" in Appendix A), then the LU sends the error notification immediately.

   - If the local product does support the FLUSH verb, then the LU may or may not send the notification immediately, depending on the product. If the LU defers sending the notification, it

buffers the notification until it accumulates a sufficient amount of information for transmission, or until the local program issues a verb that causes the LU to flush its send buffer. The amount of information that is sufficient for transmission depends on the characteristics of the session allocated for the conversation, and can vary from one session to another. Transmission of the information may begin immediately, if the LOG_DATA parameter is specified with sufficient log data, or transmission may not begin until sufficient data from subsequent SEND_DATA verbs is also buffered.

2. The local program can ensure that the remote program receives the error notification as soon as possible by issuing FLUSH immediately after SEND_ERROR.

3. SEND_ERROR is reported to the remote transaction program as one of the following return codes (based on the TYPE parameter):

   - PROG_ERROR_TRUNC or SVC_ERROR_TRUNC - The local program issued SEND_ERROR in send state after sending an incomplete logical record (see SEND_DATA). The record has been truncated.

   - PROG_ERROR_NO_TRUNC or SVC_ERROR_NO_TRUNC - The local program issued SEND_ERROR in send state after sending a complete logical record (see SEND_DATA) or prior to sending any record. No truncation has occurred.

   - PROG_ERROR_PURGING or SVC_ERROR_PURGING - The local program issued SEND_ERROR in receive state and all information sent by the remote program and not yet received by the local program, if any, has been purged; or the local program issued SEND_ERROR in confirm or sync-point state, in which case no purging has occurred.

4. When SEND_ERROR is issued in receive state, purging of incoming information occurs. The incoming information that is purged includes the following return code indications:

   - ALLOCATION_ERROR
   - BACKED_OUT
   - DEALLOCATE_ABEND_PROG
   - DEALLOCATE_ABEND_SVC
   - DEALLOCATE_ABEND_TIMER
   - PROG_ERROR_NO_TRUNC
   - PROG_ERROR_PURGING
   - PROG_ERROR_TRUNC
   - SVC_ERROR_NO_TRUNC
   - SVC_ERROR_PURGING
   - SVC_ERROR_TRUNC

   The return code DEALLOCATE_NORMAL is reported instead of ALLOCATION_ERROR, DEALLOCATE_ABEND_PROG, DEALLOCATE_ABEND_SVC, or DEALLOCATE_ABEND_TIMER. The return code OK is reported instead of the other return codes. When the return code BACKED_OUT is purged, the remote LU resends the BACKED_OUT indication and the local program receives the return code on a subsequent verb.

   The other kinds of incoming information that are purged are:

   - Data, sent by means of the SEND_DATA verb.

   - Confirmation request, sent by means of the CONFIRM, PREPARE_TO_RECEIVE, or DEALLOCATE verb.

   - Sync point request, sent by means of the SYNCPT, PREPARE_TO_RECEIVE, or DEALLOCATE verb.

   If the confirmation or sync point request was sent in conjunction with the DEALLOCATE verb (by means of its TYPE(CONFIRM) or TYPE(SYNC_LEVEL) parameter), the deallocation request is also purged.

Incoming information that is not purged is the REQUEST_TO_SEND indication. This indication is reported to the program when it issues a verb that includes the REQUEST_TO_SEND_RECEIVED parameter.

5. When REQUEST_TO_SEND_RECEIVED indicates YES, the remote program is requesting the local program to enter receive state and thereby place the remote program in send state. A program enters receive state by means of the RECEIVE_AND_WAIT or PREPARE_TO_RECEIVE verb. The partner program enters the corresponding send state when it issues a RECEIVE_AND_WAIT or RECEIVE_IMMEDIATE verb and receives the SEND indication (on the WHAT_RECEIVED parameter).

6. The program may use this verb for various application-level functions. For example, the program may issue this verb to truncate an incomplete logical record it is sending, to inform the remote program of an error it detected in data it received, or to reject a confirmation or sync-point request.

7. SEND_ERROR resets or cancels posting. If posting is active and the conversation has been posted, posting is reset. If posting is active and the conversation has not been posted, posting is canceled (posting will not occur). See the POST_ON_RECEIPT verb for more details about posting.

8. References in this verb description to a program being in a particular state are only in terms of the specified conversation.

TEST

Tests the specified conversation for a condition. The return code indicates the result of the test.

TEST	**Supplied Parameters:** **RESOURCE (variable)** [**TEST (POSTED)** **(REQUEST_TO_SEND_RECEIVED)**]
	Returned Parameters: **RETURN_CODE (variable)**
	;

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

TEST specifies the condition to be tested.

- **POSTED** specifies to test whether the conversation has been posted. The return code indicates whether posting has occurred.
- **REQUEST_TO_SEND_RECEIVED** specifies to test whether REQUEST_TO_SEND notification has been received from the remote transaction program. The return code indicates whether the notification has been received.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the program. The return code indicates the result of the test. The TEST parameter determines which of the following return codes can be returned to the program.

- If TEST(POSTED) is specified, one of the following return codes is returned:
  - OK
    - DATA
    - NOT_DATA
  - POSTING_NOT_ACTIVE
  - UNSUCCESSFUL
  - ALLOCATION_ERROR
  - BACKED_OUT
  - DEALLOCATE_ABEND_PROG
  - DEALLOCATE_ABEND_SVC
  - DEALLOCATE_ABEND_TIMER
  - DEALLOCATE_NORMAL
  - PROG_ERROR_NO_TRUNC
  - PROG_ERROR_PURGING
  - PROG_ERROR_TRUNC
  - SVC_ERROR_NO_TRUNC
  - SVC_ERROR_PURGING
  - SVC_ERROR_TRUNC
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY
- If TEST(REQUEST_TO_SEND_RECEIVED) is specified, one of the following return codes is returned:
  - OK
  - UNSUCCESSFUL

State Changes (when RETURN_CODE indicates OK):

None

ABEND Conditions:

Parameter Check

- This verb is not supported.
- TEST(POSTED) is specified and not supported.
- TEST(REQUEST_TO_SEND_RECEIVED) is specified and not supported.
- RESOURCE specifies an unassigned resource ID.

State Check

- TEST(POSTED) is specified and the conversation is not in receive state.
- TEST(REQUEST_TO_SEND_RECEIVED) is specified and the conversation is not in send, defer, or receive state.

Notes:

1. The TEST(POSTED) parameter on this verb is intended to be used in conjunction with POST_ON_RECEIPT. The use of POST_ON_RECEIPT and this verb allows a program to continue its processing while waiting for information to become available, where the program issues POST_ON_RECEIPT for one or more conversations and then issues this verb for each conversation to determine when information is available to be received.

2. For TEST(POSTED), the return code indicates whether posting has occurred, as follows:

   - OK indicates posting was active for the conversation and it has been posted. Posting is now reset. The subcode of the OK return code indicates why the conversation has been posted.

     - DATA indicates data is available for the program to receive.

     - NOT_DATA indicates information other than data, such as a SEND, CONFIRM, or TAKE_SYNCPT indication, is available for the program to receive.

     The program should issue RECEIVE_AND_WAIT or RECEIVE_IMMEDIATE in order to receive the information. The program may use the subcode to determine whether it needs to specify the DATA parameter on the RECEIVE_AND_WAIT or RECEIVE_IMMEDIATE verb.

   - POSTING_NOT_ACTIVE indicates posting is not active for the conversation.

   - UNSUCCESSFUL indicates posting is active for the conversation and it has not been posted. Posting remains active.

   The remaining return codes indicate posting was active for the conversation and it has been posted for the reason indicated by the specific return code. Posting is now reset.

3. Posting is active for a conversation when POST_ON_RECEIPT has been issued for the conversation and posting has not been reset or canceled (see the POST_ON_RECEIPT verb).

4. The TEST(REQUEST_TO_SEND_RECEIVED) parameter specifies to test whether REQUEST_TO_SEND notification has been received from the remote transaction program. The return code indicates whether the notification has been received, as follows:

   - OK indicates REQUEST_TO_SEND has been received. The remote program has issued REQUEST_TO_SEND, requesting the local program to enter receive state and thereby place the remote program in send state. A program enters receive state by means

of the RECEIVE_AND_WAIT or PREPARE_TO_RECEIVE verb. The partner program enters the corresponding send state when it issues a RECEIVE_AND_WAIT or RECEIVE_IMMEDIATE verb and receives the SEND indication (on the WHAT_RECEIVED parameter).

- UNSUCCESSFUL indicates REQUEST_TO_SEND has not been received.

5. References in this verb description to a program being in a particular state are only in terms of the specified conversation.

CONVERSATION STATES

The verbs that a program may issue for a particular conversation depend on the state of the conversation. As the program issues verbs, the state of the conversation can change. The change in the state of the conversation is a result of the function of the verb, a result of a verb issued by the remote program, or a result of network errors.

The state of a conversation is defined in terms of the local program's view of the local end of the conversation. The local end of the conversation is the end to which the local program is connected. The states of other conversations allocated to the program can be different. For example, one conversation can be in receive state and another in send state, concurrently. The following conversation states are defined at the conversation protocol boundary (where the prefix [MC_] in a verb name means the verb can be either a mapped or basic conversation verb):

Reset is the state in which the program can allocate the conversation.

Send is the state in which the program can send data, request confirmation, or request sync point.

Defer is the state, entered by the [MC_]PREPARE_TO_RECEIVE or [MC_]DEALLOCATE verb, in which the program can request sync point or confirmation, or simply flush the LU's send buffer, in order to complete the transition to receive or reset state.

Receive is the state in which the program can receive information from the remote program.

Confirm is the state in which the program can reply to a confirmation request.

Sync point is the state in which the program can respond to a sync point request.

Backed out is the state in which the program can respond to a backed out indication.

Deallocate is the state in which the program can deallocate the conversation locally.

The state of the conversation determines the verbs that a program is allowed to issue. Figure 4-1 on page 4-98 correlates the verbs, and parameters if applicable, to the conversation states. For each verb and state, a "yes," "no," or "n/a" is indicated. The "yes" means the program is allowed to issue the verb when the conversation is in that state. The "no" means the program cannot issue the verb when the conversation is in that state because the verb is disallowed in that state. A verb issued for a conversation in a disallowed state is treated as a state-check ABEND condition (see "ABEND Conditions" in Chapter 3). The individual verb descriptions list the applicable state-check ABEND conditions. The "n/a" means the state is not applicable either because it cannot exist at the time the verb is issued or because it is not relevant to the verb.

The SYNCPT and BACKOUT verbs apply to all protected resources at the time the verbs are issued, and only to protected resources. A conversation is protected when it is allocated with a synchronization level of SYNCPT. Therefore, the correlation of the SYNCPT and BACKOUT verbs to conversation states applies only to protected conversations. The states of unprotected conversations — those allocated with a synchronization level other than SYNCPT — are not relevant to SYNCPT and BACKOUT.

Verb	Conversation States							
	Reset	Send	Defer	Re-ceive	Con-firm	Sync point	Backed out	Deal-locate
[MC_]ALLOCATE	yes	n/a	n/a	n/a	n/a	n/a	n/a	n/a
[MC_]CONFIRM	n/a	yes	yes	no	no	no	no	no
[MC_]CONFIRMED	n/a	no	no	no	yes	no	no	no
[MC_]DEALLOCATE with TYPE(FLUSH), TYPE(CONFIRM), or TYPE(SYNC_LEVEL)	n/a	yes	no	no	no	no	no	no
[MC_]DEALLOCATE with TYPE(ABEND_PROG), TYPE(ABEND_SVC), or TYPE(ABEND_TIMER)	n/a	yes	yes	yes	yes	yes	no	no
[MC_]DEALLOCATE with TYPE(LOCAL)	n/a	no	no	no	no	no	no	yes
[MC_]FLUSH	n/a	yes	yes	no	no	no	no	no
[MC_]GET_ATTRIBUTES	n/a	yes	yes	yes	yes	yes	yes	yes
[MC_]POST_ON_RECEIPT	n/a	no	no	yes	no	no	no	no
[MC_]PREPARE_TO_RECEIVE	n/a	yes	no	no	no	no	no	no
[MC_]RECEIVE_AND_WAIT	n/a	yes	no	yes	no	no	no	no
[MC_]RECEIVE_IMMEDIATE	n/a	no	no	yes	no	no	no	no
[MC_]REQUEST_TO_SEND	n/a	no	no	yes	yes	yes	no	no
[MC_]SEND_DATA	n/a	yes	no	no	no	no	no	no
[MC_]SEND_ERROR	n/a	yes	no	yes	yes	yes	no	no
[MC_]TEST with TEST(POSTED)	n/a	no	no	yes	no	no	no	no
[MC_]TEST with TEST(REQUEST_TO_SEND_RECEIVED)	n/a	yes	yes	yes	no	no	no	no
BACKOUT	n/a	yes	yes	yes	yes	yes	yes	n/a
GET_TYPE	n/a	yes	yes	yes	yes	yes	yes	yes
SYNCPT	n/a	yes	yes	no	no	yes	no	n/a
WAIT	n/a	no	no	yes	no	no	no	no

Figure 4-1. Correlation of Conversation Verbs to the Conversation States Allowing Their Issuance

A conversation enters a particular state when the program issues a verb that causes a state transition or when the program receives a return code that indicates a state transition. The specific state transitions are defined in the individual verb descriptions under the heading "State Changes," and in the return code descriptions under "Return Codes" on page 4-99.

RETURN CODES

Some verbs have a parameter called RETURN_CODE used to pass a return code back to the program at the completion of the LU's execution of a verb. The return code indicates the result of verb execution, including any state changes to the specified conversation. See "Conversation States" on page 4-97 for a definition of the states.

Some of the return codes indicate results of the local LU's processing of a verb; these return codes are returned on the verb that invoked the local processing. Other return codes indicate results of processing invoked at the remote end of the conversation and, depending on the verb, can be returned on the verb that invoked the remote processing or on a subsequent verb. Still other return codes report events that originate at the remote end of the conversation. In any case, only one code is returned at a time. Other verb-specific information may be passed back in verb-unique parameters. See each specific verb for a description of any verb-unique parameters.

Some of the return codes have the suffix "RETRY" or "NO_RETRY" in their name. RETRY means that the condition indicated by the return code may not be permanent, and the program may attempt to allocate the conversation again. Whether the retry attempt succeeds depends on the duration of the condition. In general, the program should limit the number of times it attempts to retry without success, after which it should consider the condition permanent. NO_RETRY means that the condition is most likely permanent, and, in general, no attempt should be made to allocate the conversation again until the condition is corrected.

The return codes are described below. Each description includes the meaning of the return code, the origin of the condition indicated by the return code, when the return code can be reported to the program, and the state of the conversation when control is returned to the program. The individual verb descriptions list the applicable return codes.

ALLOCATION_ERROR indicates the local program issued an MC_ALLOCATE or ALLOCATE verb and allocation of the specified conversation could not be completed. The ALLOCATION_ERROR indication together with one of the following subcodes form the complete return code that is returned to the program; the subcode identifies the specific error. (The remote LU and remote program referred to in the following subcode definitions are the LU named on the LU_NAME parameter and the program named on the TPN parameter, respectively, of the verb.) When ALLOCATION_ERROR (with any subcode) is returned to the program, the conversation is in deallocate state.

- **ALLOCATION_FAILURE_NO_RETRY** indicates the conversation cannot be allocated on a session because of a condition that is not temporary. For example, the session to be used for the conversation cannot be activated because the current (LU,mode) session limit for the specified (LU-name,mode-name) pair is 0, or because of a system definition error or a session-activation protocol error; or the session was deactivated because of a session protocol error before the conversation could be allocated. The program should not retry the allocation request until the condition is corrected. This return code is returned on the MC_ALLOCATE or ALLOCATE verb when the program specifies (by means of the RETURN_CONTROL parameter) that the local LU is to attempt to allocate a session before returning control to the program; otherwise, it is returned on a subsequent verb.

- **ALLOCATION_FAILURE_RETRY** indicates the conversation cannot be allocated on a session because of a condition that may be temporary. For example, the session to be used for the conversation cannot be activated because of a temporary lack of resources at the local LU or remote LU; or the session was deactivated because of session outage before the conversation could be allocated. The condition may be temporary, and the program can retry the allocation request. This return code is returned on the MC_ALLOCATE or ALLOCATE verb when the program

specifies (by means of the RETURN_CONTROL parameter) that the local LU is to attempt to allocate a session before returning control to the program; otherwise, it is returned on a subsequent verb.

- **CONVERSATION_TYPE_MISMATCH** indicates the remote LU rejected the allocation request because the local program issued an MC_ALLOCATE or ALLOCATE verb and the remote program does not support the respective mapped- or basic-conversation protocol boundary, or the local program issued an MC_ALLOCATE verb and the remote LU does not support mapped conversations. This return code is returned on a subsequent verb.

- **PIP_NOT_ALLOWED** indicates the remote LU rejected the allocation request because the local program specified program initialization parameters (by means of the PIP(YES) parameter) and either the remote LU does not support PIP data, or the remote program has no PIP variables defined (see "Transaction Program Structure and Execution" in Chapter 3). This return code is returned on a subsequent verb.

- **PIP_NOT_SPECIFIED_CORRECTLY** indicates the remote LU rejected the allocation request because the remote program has one or more PIP variables defined and the local program specified no program initialization parameters (by means of the PIP(NO) parameter), or it specified program initialization parameters (by means of the PIP(YES) parameter) that do not correspond in number to those defined for the remote program. This return code is returned on a subsequent verb.

- **SECURITY_NOT_VALID** indicates the remote LU rejected the allocation request because the access security information (specified by means of the SECURITY parameter) is invalid. This return code is returned on a subsequent verb.

- **SYNC_LEVEL_NOT_SUPPORTED_BY_LU** indicates the local LU rejected the allocation request because the local program specified a synchronization level (by means of the SYNC_LEVEL parameter) that the remote LU does not support. This return code is returned on the MC_ALLOCATE or ALLOCATE verb when the program specifies (by means of the RETURN_CONTROL parameter) that the local LU is to attempt to allocate a session before returning control to the program; otherwise, it is returned on a subsequent verb.

- **SYNC_LEVEL_NOT_SUPPORTED_BY_PGM** indicates the remote LU rejected the allocation request because the local program specified a synchronization level (by means of the SYNC_LEVEL parameter) that the remote program does not support. This return code is returned on a subsequent verb.

- **TPN_NOT_RECOGNIZED** indicates the remote LU rejected the allocation request because the local program specified a remote program name that the remote LU does not recognize. This return code is returned on a subsequent verb.

- **TRANS_PGM_NOT_AVAIL_NO_RETRY** indicates the remote LU rejected the allocation request because the local program specified a remote program that the remote LU recognizes but cannot start. The condition is not temporary, and the program should not retry the allocation request. This return code is returned on a subsequent verb.

- **TRANS_PGM_NOT_AVAIL_RETRY** indicates the remote LU rejected the allocation request because the local program specified a remote program that the remote LU recognizes but currently cannot start. The condition may be temporary, and the program can retry the allocation request. This return code is returned on a subsequent verb.

With one exception, the subcodes of ALLOCATION_ERROR are not explicitly listed in the individual verb descriptions, as any of them can be returned as part of the ALLOCATION_ERROR return code. The exception to this is for the MC_ALLOCATE and ALLOCATE verbs,

on which only certain subcodes can be returned; therefore, the applicable subcodes are listed for these verbs.

BACKED_OUT indicates the remote program issued a BACKOUT, or the local or remote LU has done so, in order to restore all protected resources to their status as of the last synchronization point. This return code can be reported to the local program on a verb it issues in send, defer, or receive state. The conversation is in backed-out state.

DEALLOCATE_ABEND indicates the remote program issued an MC_DEALLOCATE verb specifying the TYPE(ABEND) parameter, or the remote LU has done so because of a remote program ABEND condition. If the conversation for the remote program was in receive state when the verb was issued, information sent by the local program and not yet received by the remote program is purged. This return code can be reported to the local program on a verb it issues in send, defer, or receive state. The conversation is in deallocate state.

DEALLOCATE_ABEND_PROG indicates the remote program issued a DEALLOCATE verb specifying the TYPE(ABEND_PROG) parameter, or the remote LU has done so because of a remote program ABEND condition. If the conversation for the remote program was in receive state when the verb was issued, information sent by the local program and not yet received by the remote program is purged. This return code can be reported to the local program on a verb it issues in send, defer, or receive state. The conversation is in deallocate state.

DEALLOCATE_ABEND_SVC indicates the remote program issued a DEALLOCATE verb specifying the TYPE(ABEND_SVC) parameter. If the conversation for the remote program was in receive state when the verb was issued, information sent by the local program and not yet received by the remote program is purged. This return code can be reported to the local program on a verb it issues in send, defer, or receive state. The conversation is in deallocate state.

DEALLOCATE_ABEND_TIMER indicates the remote program issued a DEALLOCATE verb specifying the TYPE(ABEND_TIMER) parameter. If the conversation for the remote program was in receive state when the verb was issued, information sent by the local program and not yet received by the remote program is purged. This return code can be reported to the local program on a verb it issues in send, defer, or receive state. The conversation is in deallocate state.

DEALLOCATE_NORMAL indicates the remote program issued an MC_DEALLOCATE or DEALLOCATE verb specifying the TYPE(SYNC_LEVEL) or TYPE(FLUSH) parameter; if TYPE(SYNC_LEVEL), either the synchronization level is NONE or it is SYNCPT and the remote program subsequently issued an MC_FLUSH or FLUSH verb. This return code is reported to the local program on a verb it issues in receive state. The conversation is in deallocate state.

FMH_DATA_NOT_SUPPORTED indicates the local program issued an MC_SEND_DATA specifying that the data record contains FM headers (by means of the FMH_DATA parameter), and either the remote LU or remote program does not support FM header data. This return code is reported on a subsequent verb. All information sent by the local program on the MC_SEND_DATA verb and subsequent verbs prior to the reporting of the FMH_DATA_NOT_SUPPORTED return code is purged. The conversation is in send state.

HEURISTIC_MIXED indicates the local program issued a SYNCPT and an error occurred during the sync point processing within the distributed transaction. As a result of the error and subsequent LU operator intervention, at least one LU advanced its local protected resources to the next synchronization point and at least one LU restored its local protected resources to the previous synchronization point.

MAP_EXECUTION_FAILURE indicates the local program issued an MC_SEND_DATA specifying the data record is to be mapped (by means of the MAP_NAME parameter), and the local LU or the remote LU

could not map the data record based on the map name. This return code is returned on the MC_SEND_DATA verb when the map execution failed at the local LU. Otherwise, the remote LU rejects the data, and the return code is reported on a subsequent verb. All information sent by the local program on that MC_SEND_DATA verb and subsequent verbs prior to the reporting of the MAP_EXECUTION_FAILURE return code is purged. The conversation is in send state.

MAP_NOT_FOUND indicates the local program issued an MC_SEND_DATA specifying the data record is to be mapped (by means of the MAP_NAME parameter), and the map name is unknown to the local LU or the remote LU. This return code is reported on the MC_SEND_DATA verb when the map name is unknown to the local LU. Otherwise, the remote LU rejects the data, and the return code is reported on a subsequent verb. All information sent by the local program on that MC_SEND_DATA verb and subsequent verbs prior to the reporting of the MAP_NOT_FOUND return code is purged. The conversation is in send state.

MAPPING_NOT_SUPPORTED indicates the local program issued an MC_SEND_DATA specifying the data record is to be mapped (by means of the MAP_NAME parameter), and either the remote LU or remote program does not support data mapping. This return code is reported on a subsequent verb. All information sent by the local program on that MC_SEND_DATA verb and subsequent verbs prior to the reporting of the MAPPING_NOT_SUPPORTED return code is purged. The conversation is in send state.

OK indicates the verb issued by the local program executed successfully. That is, the function defined for the verb, up to the point at which control is returned to the program, was performed as specified. The state of the conversation is as defined for the verb.

For some verbs, the OK indication together with one of the following subcodes form the complete return code that is returned to the program; the subcode provides additional information.

- **DATA** indicates data is available for the program to receive.
- **NOT_DATA** indicates information other than data is available for the program to receive.

PARAMETER_ERROR indicates the local program issued a verb specifying a parameter containing an invalid argument. The source of the argument is considered to be outside the program definition, such as an LU name supplied by a terminal operator and used as the argument of LU_NAME on MC_ALLOCATE or ALLOCATE. Contrast this definition with the definition of the ABEND condition, Parameter Check, in the section "ABEND Conditions" in Chapter 3. This return code is returned on the verb specifying the invalid argument. The state of the conversation remains unchanged.

POSTING_NOT_ACTIVE indicates the local program issued a verb that determines whether a resource has been posted, and posting is not active for any of the specified resources.

PROG_ERROR_NO_TRUNC indicates one of the following:

- The remote program issued an MC_SEND_ERROR verb and the conversation for the remote program was in send state. No truncation occurs at the mapped conversation protocol boundary. This return code is reported to the local program on an MC_RECEIVE_AND_WAIT or MC_RECEIVE_IMMEDIATE verb it issues prior to receiving any data records or after receiving one or more data records.
- The remote program issued a SEND_ERROR verb specifying the TYPE(PROG) parameter, the conversation for the remote program was in send state, and the verb did not truncate a logical record. No truncation occurs at the basic conversation protocol boundary when a program issues SEND_ERROR before sending any logical records or after sending a complete logical

record. This return code is reported to the local program on a RECEIVE_AND_WAIT or RECEIVE_IMMEDIATE verb it issues prior to receiving any logical records or after receiving one or more complete logical records.

The conversation remains in receive state.

PROG_ERROR_PURGING indicates the remote program issued an MC_SEND_ERROR verb or it issued a SEND_ERROR verb specifying the TYPE(PROG) parameter, and the conversation for the remote program was in receive, confirm, or sync point state. The verb may have caused information to be purged. Purging occurs when a program issues MC_SEND_ERROR or SEND_ERROR in receive state before receiving all the information sent by its partner program, that is, all the information sent prior to the reporting of the PROG_ERROR_PURGING return code to the partner program. The purging can occur at the local LU, remote LU, or both. No purging occurs when a program issues the verb in confirm state or sync point state, or in receive state after receiving all the information sent by its partner program. This return code is normally reported to the local program on a verb it issues after sending some information to the remote program. However, the return code can be reported on a verb the program issues prior to sending any information, depending on the verb and when it is issued. The conversation is in receive state.

PROG_ERROR_TRUNC indicates the remote program issued a SEND_ERROR verb specifying the TYPE(PROG) parameter, the conversation for the remote program was in send state, and the verb truncated a logical record. Truncation occurs at the basic conversation protocol boundary when a program begins sending a logical record and then issues SEND_ERROR before sending the complete logical record. This return code is reported to the local program on a RECEIVE_AND_WAIT or RECEIVE_IMMEDIATE verb it issues after receiving the truncated logical record. The conversation remains in receive state.

SVC_ERROR_NO_TRUNC indicates the remote program issued a SEND_ERROR verb specifying the TYPE(SVC) parameter. Otherwise, this return code, as it applies to the basic conversation protocol boundary, has the same meaning as PROG_ERROR_NO_TRUNC. The conversation remains in receive state.

SVC_ERROR_PURGING indicates the remote program issued a SEND_ERROR verb specifying the TYPE(SVC) parameter. Otherwise, this return code, as it applies to the basic conversation protocol boundary, has the same meaning as PROG_ERROR_PURGING. The conversation is in receive state.

SVC_ERROR_TRUNC indicates the remote program issued a SEND_ERROR specifying the TYPE(SVC) parameter. Otherwise, this return code has the same meaning as PROG_ERROR_TRUNC. The conversation remains in receive state.

RESOURCE_FAILURE_NO_RETRY indicates a failure occurred that caused the conversation to be prematurely terminated. For example, the session being used for the conversation was deactivated because of a session protocol error, or the conversation was deallocated because of a protocol error between the mapped conversation components of the LUs. The condition is not temporary, and the program should not retry the transaction until the condition is corrected. This return code can be reported to the local program on a verb it issues in any state other than reset or deallocate. The conversation is in deallocate state.

RESOURCE_FAILURE_RETRY indicates a failure occurred that caused the conversation to be prematurely terminated. For example, the session being used for the conversation was deactivated because of a session outage, such as a line failure, a modem failure, or a crypto engine failure. The condition may be temporary, and the program can retry the transaction. This return code can be reported to the local program on a verb it issues in any state other than reset or deallocate. The conversation is in deallocate state.

UNSUCCESSFUL indicates the verb issued by the local program did not execute successfully. This return code is returned on the unsuccessful verb. The state of the conversation remains unchanged.

Figure 4-2 on page 4-105 shows the correlation of the return codes to the verbs on which they can be returned. The "X" in the figure means the return code can be returned on the corresponding verb. A verb without any "X"s beside it means no return codes are defined for the verb.

Verbs	Return Codes																							
	ALLOCATION ERROR	BACKED OUT	DEALLOCATE ABEND	DEALLOCATE ABEND PROG	DEALLOCATE ABEND SVC	DEALLOCATE ABEND TIMER	DEALLOCATE NORMAL	FMH DATA NOT SUPPORTED	HEURISTIC MIXED	MAP EXECUTION FAILURE	MAP NOT FOUND	MAPPING NOT SUPPORTED	OK	PARAMETER ERROR	POSTING NOT ACTIVE	PROG ERROR NO TRUNC	PROG ERROR PURGING	PROG ERROR TRUNC	RESOURCE FAILURE NO RETRY	RESOURCE FAILURE RETRY	SVC ERROR NO TRUNC	SVC ERROR PURGING	SVC ERROR TRUNC	UNSUCCESSFUL
MC_ALLOCATE	X												X	X										X
MC_CONFIRM	X	X	X					X		X	X	X	X				X		X	X				
MC_CONFIRMED																								
MC_DEALLOCATE	X		X					X		X	X	X	X				X		X	X				
MC_FLUSH																								
MC_GET_ATTRIBUTES																								
MC_POST_ON_RECEIPT																								
MC_PREPARE_TO_RECEIVE	X		X					X		X	X	X	X				X		X	X				
MC_RECEIVE_AND_WAIT	X	X	X				X	X		X	X	X	X			X	X		X	X				
MC_RECEIVE_IMMEDIATE	X	X	X				X	X		X	X	X	X			X	X		X	X				X
MC_REQUEST_TO_SEND																								
MC_SEND_DATA	X	X	X					X		X	X	X	X				X		X	X				
MC_SEND_ERROR	X	X	X				X	X		X	X	X	X				X		X	X				
MC_TEST	X	X	X				X	X		X	X	X	X		X	X	X		X	X				X
BACKOUT																								
GET_TYPE																								
SYNCPT		X							X				X											
WAIT	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X	
ALLOCATE	X												X	X										X
CONFIRM	X	X		X	X	X							X				X		X	X		X		
CONFIRMED																								
DEALLOCATE	X			X	X	X							X				X		X	X		X		
FLUSH																								
GET_ATTRIBUTES																								
POST_ON_RECEIPT																								
PREPARE_TO_RECEIVE	X			X	X	X							X				X		X	X		X		
RECEIVE_AND_WAIT	X	X		X	X	X	X						X			X	X	X	X	X	X	X	X	
RECEIVE_IMMEDIATE	X	X		X	X	X	X						X			X	X	X	X	X	X	X	X	X
REQUEST_TO_SEND																								
SEND_DATA	X	X		X	X	X							X				X		X	X		X		
SEND_ERROR	X	X		X	X	X	X						X				X		X	X		X		
TEST	X	X		X	X	X	X						X		X	X	X	X	X	X	X	X	X	X

Figure 4-2. Correlation of Return Codes to Verbs

This page intentionally left blank

CHAPTER 5. CONTROL-OPERATOR VERBS

This chapter describes the category of verbs called control-operator verbs. The control-operator verbs define the protocol boundary for a control-operator transaction program. In particular, the control-operator protocol boundary is intended for use by transaction programs that assist the control operator in performing functions related to the control of an LU.

Preceding the detailed descriptions of the control-operator verbs is a discussion of LU-LU sessions and functional subcategories of the control-operator verbs. Following the verb descriptions is a description of the return codes that apply to the control-operator verbs.

LU-LU SESSIONS

The following characteristics of LU-LU sessions are relevant to the control-operator verbs:

- The means of connecting LUs — single session or parallel sessions
- The contention-winner polarities of LU-LU sessions

SINGLE AND PARALLEL SESSIONS

Two LUs may connect to each other by means of one LU-LU session, called a single session, or multiple LU-LU sessions, called parallel sessions. The means of connection, single session or parallel sessions, depends on the products implementing the LUs. LU 6.2 products that provide an application programming interface (API), for user-written programs, equivalent to the conversation verbs support both single-session and parallel-session connections. Products that are not user programmable, or that are but do not provide an API equivalent to the conversation verbs, may support only single-session connections, and typically do so; however, they may also support parallel-session connections.

- When parallel-session support is available to both LUs, they can connect to each other using a single session or parallel sessions.
- When parallel-session support is unavailable to either LU, they can connect to each other using only a single session.

The session activation request that one LU sends to another indicates whether the session is a single session or a parallel session.

An LU for which parallel session support is available establishes the means of connection to a partner LU at the time the partner LU is defined (see the DEFINE_REMOTE_LU verb). Two LUs cannot be connected by both single and parallel sessions at the same time.

For single-session connections, an implied limit of 1 is imposed on the number of active sessions between the two LUs. That is, another session cannot be activated until the active session is deactivated.

For parallel-session connections, the sessions can be partitioned into groups. Each group has a limit on the number of active sessions within that group, which is agreed to by both LUs. Additional sessions within a group can be activated up to its limit.

Each single session or group of parallel sessions has associated with it a set of similar network properties and a corresponding mode name. The mode name serves as an identifier of the set of network properties. It allows a transaction program to select the set of network properties to be used for a conversation.

The set of similar network properties includes, for example, the highest synchronization level for conversations on the sessions, the class of service for the sessions, whether the sessions provide transmission security by means of session- or link-level cryptography, and the session routing and delay characteristics. The correlation of mode names to the sets of network properties is established at the time the mode name is defined for the partner LU (see the DEFINE_MODE verb).

The similarity of network properties among a group of sessions does not imply that the network properties must be identical for all sessions in the group. For example, the sessions within a group can be activated on different physical transmission facilities and yet have equivalent class of service, or they can have different physical security characteristics and still have equivalent level of transmission security.

CONTENTION-WINNER POLARITY

For each single or parallel LU-LU session, only one LU is the contention winner of the session; the other LU is the contention loser of the session. This contention-winner polarity of LU-LU sessions determines how contention is resolved when the two LUs attempt to allocate a conversation on the session at the same time.[1] The contention-winner LU allocates a conversation on a session without asking permission from the contention-loser LU. Conversely, the contention-loser LU requests permission from the contention-winner LU to allocate a conversation on the session, and the contention-winner LU either grants or rejects the request. The contention-winner polarity of a session is established at session activation time.

For single sessions, the LU initiating the session activation can request that it be the contention winner or loser. The LU responding to the session activation can accept the requested polarity or change the polarity, depending on the requested polarity. If the initiating LU requests that it be the contention winner, the responding LU can accept the polarity or change the polarity making it the contention winner. If the initiating LU requests that it be the contention loser, the responding LU always accepts the polarity.

For parallel sessions, each mode-name group of sessions can be partitioned based on contention-winner polarities. A number of sessions in the group can be designated as the minimum number of contention-winner sessions for one LU, and all or part of the remaining sessions can be designated as the minimum number of contention-winner sessions for the other LU. This partitioning allows two LUs to divide a group of parallel sessions between them such that each LU is assured of being the contention winner of a minimum number of the sessions.

The LU initiating the activation of a parallel session designates that it be the contention winner or contention loser. Details of how the initiating LU determines whether it is to be the contention winner or loser of a session are given in the notes in the descriptions of the CHANGE_SESSION_LIMIT and INITIALIZE_SESSION_LIMIT verbs. The LU responding to the activation of a parallel session always accepts the designated polarity.

Sessions can be activated by means of certain control-operator verbs, and as a result of allocation requests. The control-operator verbs that can cause sessions to be activated are CHANGE_SESSION_LIMIT, INITIALIZE_SESSION_LIMIT, and ACTIVATE_SESSION. Refer to the descriptions of these verbs for more details about session activation.

Note: The contention-winner polarity of sessions assigned the SNA-defined mode name, SNASVCMG, is not negotiable. That is, the LU responding to the activation of an SNASVCMG session always accepts the designated polarity.

---

[1] Specific details are given in SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2.

VERB DESCRIPTIONS

The control-operator verbs are divided into the following subcategories:

- Change number of sessions verbs
- Session control verbs
- LU definition verbs

The detailed descriptions of the control-operator verbs follow.

CHANGE NUMBER OF SESSIONS VERBS

This subcategory of control-operator verbs consists of four verbs called the change-number-of-sessions, or CNOS, verbs. The CNOS verbs change the (LU,mode) session limit, which controls the number of LU-LU sessions per mode name that are available between two LUs for allocation to conversations. The CNOS verbs apply to both single- and parallel-session connections.

For single sessions and SNASVCMG sessions, the CNOS verbs change the (LU,mode) session limit only at the local LU. The remote LU is not involved in processing the change. The CNOS verbs that a control-operator transaction program may issue for a single session or SNASVCMG session are:

INITIALIZE_SESSION_LIMIT
RESET_SESSION_LIMIT

For parallel sessions, the CNOS verbs change the (LU,mode) session limit as well as other CNOS parameters of the LUs, and both LUs are involved in processing the changes. These other CNOS parameters control the minimum number of contention-winner LU-LU sessions for each LU, control which LU is responsible for selecting and deactivating LU-LU sessions when the (LU,mode) session limit is decreased or reset, and control the draining of allocation requests when the (LU,mode) session limit is reset.

The two LUs cooperate in the execution of the CNOS verbs by means of a CNOS request and CNOS reply. The LU executing the control-operator transaction program sends a CNOS request to the partner LU. The partner LU invokes an SNA service transaction program called the "CNOS service transaction program" (see "Appendix D. List of SNA Service Transaction Programs"), which causes the partner LU to process the CNOS request and send back a CNOS reply. The CNOS request and reply are sent on a basic conversation, referred to in this chapter as the "CNOS conversation." The CNOS conversation is normally allocated on an SNASVCMG session. However, if an SNASVCMG session is not active, because of session outage for example, the CNOS conversation may be allocated on another active session.

The LU that sends the CNOS request is referred to as the source LU; the LU that receives the CNOS request is referred to as the target LU. The execution of a CNOS verb by the two LUs is considered a CNOS transaction. The role of the LU as a source LU or target LU lasts for the duration of the CNOS transaction.

The CNOS verbs that a control-operator transaction program may issue for parallel sessions are:

CHANGE_SESSION_LIMIT
INITIALIZE_SESSION_LIMIT
RESET_SESSION_LIMIT

Only a transaction program that has CNOS privilege may issue these verbs. The program is designated to have CNOS privilege when it is defined to the local LU (see the DEFINE_TP verb).

The CNOS verb that the CNOS service transaction program issues for parallel sessions is:

PROCESS_SESSION_LIMIT

The detailed descriptions of the CNOS verbs follows.

CHANGE_SESSION_LIMIT

Changes the (LU,mode) session limit and contention-winner polarities for parallel-session connections. The verb applies to the group of sessions with the specified mode name between this (source) LU and the specified (target) LU. The new (LU,mode) session limit and contention-winner polarities are enforced until changed by a subsequent CNOS verb. As a consequence of changing the (LU,mode) session limit and contention-winner polarities, LU-LU sessions with the specified mode name may be activated or deactivated to conform to the new session limit and polarities.

CHANGE_SESSION_LIMIT	Supplied Parameters: LU_NAME (variable) MODE_NAME (variable) LU_MODE_SESSION_LIMIT (variable) MIN_CONWINNERS_SOURCE (variable) MIN_CONWINNERS_TARGET (variable) [RESPONSIBLE (SOURCE)] [(TARGET)]
	Returned Parameters: RETURN_CODE (variable)
	;

Supplied Parameters:

LU_NAME specifies the name of the target LU to which the change in session limit and polarities applies. The LU name is a name that is valid as the LU_NAME parameter of the ALLOCATE verb (see "ALLOCATE" in Chapter 4).

MODE_NAME specifies the mode name for which the session limit and polarities are to be changed. The specified mode name cannot be the SNA-defined mode name, SNASVCMG.

LU_MODE_SESSION_LIMIT specifies the (LU,mode) session limit, that is, the maximum number of sessions to be allowed, between the source LU and target LU, for the specified mode name.

The specified session limit must be greater than 0. The target LU can negotiate this parameter to a value greater than 0 and less than the specified session limit. The specified session limit, or the negotiated session limit if it is negotiated, becomes the new session limit.

The value specified on this parameter must be greater than or equal to the sum of the values specified on the MIN_CONWINNERS_SOURCE and MIN_CONWINNERS_TARGET parameters.

MIN_CONWINNERS_SOURCE specifies the number of sessions of which the source LU is designated to be the contention winner. The specified number must be 0 or greater. The specified number, or the negotiated number if it is negotiated, becomes the new minimum number of contention-winner sessions for the source LU. The sum of this number and the target LU's new minimum number of contention-winner sessions cannot exceed the new session limit.

When the specified number is greater than 1/2 the new session limit (rounded downward), the target LU can negotiate this parameter to a number greater than or equal to 1/2 the new session limit and less than the specified number. When the specified number is less than or

equal to 1/2 the new session limit, the target LU cannot negotiate this parameter.

MIN_CONWINNERS_TARGET specifies the number of sessions of which the target LU is designated to be the contention winner. The specified number must be 0 or greater. The specified number, or the negotiated number if it is negotiated, becomes the new minimum number of contention-winner sessions for the target LU. The sum of this number and the source LU's new minimum number of contention-winner sessions cannot exceed the new session limit.

The target LU can negotiate this parameter to a number less than or equal to the new session limit minus the new minimum number of contention-winner sessions for the source LU.

RESPONSIBLE specifies which LU is responsible for selecting and deactivating sessions as a result of a change that decreases the session limit or the maximum number of contention-winner sessions for the source or target LU; see note 4 for details. If no sessions need to be deactivated, this parameter is ignored.

- **SOURCE** specifies that the source LU is responsible. The target LU cannot negotiate this argument.
- **TARGET** specifies that the target LU is responsible. The target LU can negotiate this argument to SOURCE, in which case the source LU becomes responsible.

The responsible LU can deactivate a session when both LUs are finished using the session. This verb will not terminate active conversations.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the program. The return code indicates the result of verb execution.

- OK (with one of the following subcodes)
  - AS_SPECIFIED
  - AS_NEGOTIATED
- ALLOCATION_ERROR
- COMMAND_RACE_REJECT
- LU_MODE_SESSION_LIMIT_ZERO
- LU_SESSION_LIMIT_EXCEEDED
- PARAMETER_ERROR (for one of the following reasons)
  - Invalid LU name
  - Invalid mode name
- REQUEST_EXCEEDS_MAX_ALLOWED
- RESOURCE_FAILURE_NO_RETRY
- UNRECOGNIZED_MODE_NAME

ABEND Conditions:

Parameter Check

- This verb is not supported.
- The program issuing this verb does not have CNOS privilege.
- MODE_NAME specifies the SNA-defined mode name, SNASVCMG.
- LU_MODE_SESSION_LIMIT specifies 0.
- MIN_CONWINNERS_TARGET is specified and not supported.
- The sum of MIN_CONWINNERS_SOURCE and MIN_CONWINNERS_TARGET specifies a number greater than LU_MODE_SESSION_LIMIT.
- RESPONSIBLE(TARGET) is specified and not supported.

Notes:

1. This verb applies only to parallel-session connections.
2. All of the parallel sessions between two LUs can be partitioned into groups, with all the sessions in a group having the same mode name. This verb is used to change the limits on the number of active sessions that can exist concurrently within a mode-name group between the source LU and target LU. The limits imposed on

the number of active parallel sessions within a mode-name group are:

a. The number of active sessions cannot exceed the (LU,mode) session limit.

b. The number of active contention-winner sessions for the source LU cannot exceed the (LU,mode) session limit minus the minimum number of contention-winner sessions for the target LU.

c. The number of active contention-winner sessions for the target LU cannot exceed the (LU,mode) session limit minus the new minimum number of contention-winner sessions for the source LU.

As a result of issuing this verb, sessions may be activated, deactivated, or both to conform to the new limits. The next two notes describe the conditions under which sessions may be activated or deactivated.

3. The source or target LU may activate parallel sessions, up to the limits given in note 2, as follows:

   - It may activate new contention-winner sessions when the number of its active contention-winner sessions is less than the new (LU,mode) session limit minus the new minimum number of contention-winner sessions for its partner LU.

   - It may activate new contention-loser sessions when the number of its active contention-loser sessions is less than the new (LU,mode) session limit minus its own new minimum number of contention-winner sessions.

   The LU may activate contention-winner or -loser sessions in response to allocation requests or by means of the ACTIVATE_SESSION verb. Also, it may activate contention-winner sessions automatically, after completion of this verb, up to the lesser of the its new minimum number of contention-winner sessions and its automatic-activation limit currently in effect (see the DEFINE_MODE verb).

4. Parallel sessions are deactivated when this verb causes one or more of the limits given in note 2 to be exceeded. The LU responsible for selecting and deactivating sessions is designated by the RESPONSIBLE parameter. The responsible LU deactivates sessions until all three limits given in note 2 are met. When all three limits are met, no more sessions are deactivated.

   The responsible LU deactivates sessions that are not allocated to conversations. If a session to be deactivated is currently allocated to a conversation, the responsible LU waits until the conversation is deallocated and then deactivates the session.

   The responsible LU can deactivate only those sessions that are in excess of an LU's minimum number of contention-winner sessions. If the number of currently active contention-winner sessions for the source or target LU is less than or equal to its minimum number of contention-winner sessions, none of its contention-winner sessions are deactivated.

INITIALIZE_SESSION_LIMIT

Establishes the initial (LU,mode) session limit for single- or parallel-session connections, and the contention-winner polarities for parallel-session connections. The verb applies to the group of sessions with the specified mode name between the source LU and the target LU. The new (LU,mode) session limit and contention-winner polarities are enforced until changed by a subsequent CNOS verb. As a consequence of initializing the session limit, one or more LU-LU sessions with the specified mode name may be activated.

INITIALIZE_SESSION_LIMIT	Supplied Parameters: LU_NAME (variable) MODE_NAME (variable) ('SNASVCMG') LU_MODE_SESSION_LIMIT (variable) MIN_CONWINNERS_SOURCE (variable) MIN_CONWINNERS_TARGET (variable)
	Returned Parameters: RETURN_CODE (variable)
	;

Supplied Parameters:

LU_NAME specifies the name of the target LU to which the initialization of session limit and polarities applies. The LU name is a name that is valid as the LU_NAME parameter of the ALLOCATE verb (see "ALLOCATE" in Chapter 4).

MODE_NAME specifies the mode name for which the session limit and polarities are to be initialized.

- **variable** contains the mode name.
- **'SNASVCMG'** specifies the SNA-defined mode name, which is used for exchanging the CNOS request and reply when the source LU and target LU are connected by parallel sessions.

LU_MODE_SESSION_LIMIT specifies the (LU,mode) session limit for parallel-session connections, that is, the maximum number of parallel sessions to be allowed, between the source LU and target LU, for the specified mode name.

The specified session limit must be greater than 0. The target LU can negotiate this parameter to a value greater than 0 and less than the specified session limit. The specified session limit, or the negotiated session limit if it is negotiated, becomes the new session limit.

The value specified on this parameter must be greater than or equal to the sum of the values specified on the MIN_CONWINNERS_SOURCE and MIN_CONWINNERS_TARGET parameters.

For single-session connections, the specified (LU,mode) session limit must be 1.

For the SNASVCMG mode name, the specified (LU,mode) session limit must be 2.

MIN_CONWINNERS_SOURCE specifies the number of parallel sessions of which the source LU is designated to be the contention winner. The specified number must be 0 or greater. The specified number, or the negotiated number if it is negotiated, becomes the new minimum number of contention-winner sessions for the source LU. The sum of this num-

ber and the target LU's new minimum number of contention-winner sessions cannot exceed the new session limit.

When the specified number is greater than 1/2 the new session limit (rounded downward), the target LU can negotiate this parameter to a number greater than or equal to 1/2 the new session limit and less than the specified number. When the specified number is less than or equal to 1/2 the new session limit, the target LU cannot negotiate this parameter.

For single-session connections, the specified minimum number of contention-winner sessions for the source LU may be 0, or it may be 1 if the value specified on the MIN_CONWINNER_TARGET parameter is 0.

For the SNASVCMG mode name, the specified minimum number of contention-winner sessions for the source LU must be 1.

MIN_CONWINNERS_TARGET specifies the number of parallel sessions of which the target LU is designated to be the contention winner. The specified number must be 0 or greater. The specified number, or the negotiated number if it is negotiated, becomes the new minimum number of contention-winner sessions for the target LU. The sum of this number and the source LU's new minimum number of contention-winner sessions cannot exceed the new session limit.

The target LU can negotiate this parameter to a number less than or equal to the new session limit minus the new minimum number of contention-winner sessions for the source LU.

For single-session connections, the specified minimum number of contention-winner sessions for the target LU may be 0, or it may be 1 if the value specified on the MIN_CONWINNER_SOURCE parameter is 0.

For the SNASVCMG mode name, the specified minimum number of contention-winner sessions for the target LU must be 1.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the program. The return code indicates the result of verb execution.

- OK (with one of the following subcodes)
  - AS_SPECIFIED
  - AS_NEGOTIATED
- ALLOCATION_ERROR
- COMMAND_RACE_REJECT
- LU_MODE_SESSION_LIMIT_CLOSED
- LU_MODE_SESSION_LIMIT_NOT_ZERO
- LU_SESSION_LIMIT_EXCEEDED
- PARAMETER_ERROR (for one of the following reasons)
  - Invalid LU name
  - Invalid mode name
- REQUEST_EXCEEDS_MAX_ALLOWED
- RESOURCE_FAILURE_NO_RETRY
- UNRECOGNIZED_MODE_NAME

ABEND Conditions:

Parameter Check

- The program issuing this verb does not have CNOS privilege.
- LU_MODE_SESSION_LIMIT specifies a value greater than 1 for a single-session connection.
- MODE_NAME('SNASVCMG') is specified and LU_MODE_SESSION_LIMIT, MIN_CONWINNERS_SOURCE, and MIN_CONWINNERS_TARGET do not specify 2, 1, and 1, respectively.
- LU_MODE_SESSION_LIMIT specifies 0.
- MIN_CONWINNERS_TARGET is specified and not supported.
- The sum of MIN_CONWINNERS_SOURCE and MIN_CONWINNERS_TARGET specifies a number greater than LU_MODE_SESSION_LIMIT.

Notes:

1. The (LU,mode) session limit for a single-session connection is initialized only locally at the source LU; a CNOS request and reply are not exchanged between the two LUs. Thus, the INITIALIZE_SESSION_LIMIT verb must be issued at both LUs before either LU can activate the corresponding session. From each LU's perspective, each is the source LU for the processing of this verb.

2. For a single-session connection, the contention-winner polarity for the session is determined from the MIN_CONWINNER_SOURCE and MIN_CONWINNER_TARGET parameters on the verbs issued at both LUs. The LU activating the session:

   - Requests that it be the contention winner when the verb issued at that LU specifies MIN_CONWINNERS_TARGET(0).
   - Requests that it be the contention loser when the verb specifies MIN_CONWINNERS_TARGET(1).

   The partner LU:

   - Accepts the requested contention-winner polarity when the verb issued at that LU specifies MIN_CONWINNERS_SOURCE(0).
   - Negotiates the polarity so that it is the contention winner when the verb specifies MIN_CONWINNERS_SOURCE(1).

3. For a single-session connection, the LU may have more than one mode defined at a time to a partner LU. Each (LU,mode) session limit can be either 0 or 1, and more than one (LU,mode) session limit can be 1, concurrently, for the partner LU. This permits the LU to activate a session for any of the modes that have an (LU,mode) session limit of 1; however, only one session can be active at a time.

4. For a parallel-session connection to a target LU, the (LU,mode) session limit and contention-winner polarities for the SNASVCMG mode name and target LU must be initialized before (LU,mode) session limit and contention-winner polarities can be initialized for any other mode name for the target LU. The (LU,mode) session limit and contention-winner polarities for the SNASVCMG mode name and target LU are initialized only at the source LU; a CNOS request and reply are not exchanged between the two LUs.

   The (LU,mode) session limit and contention-winner polarities for the SNASVCMG mode name must be initialized at both the source LU and target LU before either LU can activate the corresponding sessions; a CNOS request and reply are not exchanged between the two LUs. From each LU's perspective, each is the source LU for the processing of INITIALIZE_SESSION_LIMIT with MODE_NAME('SNASVCMG').

5. All the parallel sessions between two LUs can be partitioned into groups, with all the sessions in a group having the same mode name. This verb can be used to initialize the limits on the number of active parallel sessions that can exist concurrently within a mode-name group between the source and target LUs. The limits imposed on the number of active parallel sessions within a mode-name group are:

   a. The number of active sessions cannot exceed the (LU,mode) session limit.

   b. The number of active contention-winner sessions for the source LU cannot exceed the (LU,mode) session limit minus the minimum number of contention-winner sessions for the target LU.

   c. The number of active contention-winner sessions for the target LU cannot exceed the (LU,mode) session limit minus the new minimum number of contention-winner sessions for the source LU.

As a result of issuing this verb, parallel sessions may be activated to conform to the new limits.

6. For single- and parallel-session connections, an LU may activate contention-winner or -loser sessions in response to allocation requests or by means of the ACTIVATE_SESSION verb. Also, it may activate contention-winner sessions automatically, after completion of this verb, up to the lesser of the its new minimum number of contention-winner sessions and its automatic-activation limit currently in effect.

RESET_SESSION_LIMIT

Resets to 0 the (LU,mode) session limit for single or parallel-session connections, and the contention-winner polarities for the parallel-session connections. The verb applies to the group of sessions with the specified mode name, or all mode names, between the source LU and the target LU. The reset (LU,mode) session limit and contention-winner polarities are enforced until initialized by a subsequent INITIALIZE_SESSION_LIMIT verb. As a consequence of resetting the session limit and polarities, all active sessions with the specified mode name, or all mode names, are deactivated.

RESET_SESSION_LIMIT	**Supplied Parameters:** LU_NAME (variable) [MODE_NAME (ALL) (ONE (variable)) (ONE ('SNASVCMG'))] [RESPONSIBLE (SOURCE) (TARGET)] [DRAIN_SOURCE (NO) (YES)] [DRAIN_TARGET (NO) (YES)] [FORCE (NO) (YES)]
	Returned Parameters: RETURN_CODE (variable)
	;

Supplied Parameters:

LU_NAME specifies the name of the target LU to which the resetting of the session limit and polarities applies. The LU name is a name that is valid as the LU_NAME parameter of the ALLOCATE verb (see "ALLOCATE" in Chapter 4).

MODE_NAME specifies the mode name for which the session limit and polarities are to be reset to 0.

- **ALL** specifies that the session limit and polarities for all mode names that apply to the target LU are to be reset to 0, except for the SNA-defined mode name, SNASVCMG, which remains unchanged.
- **ONE(variable)** specifies that the session limit and polarities for only the specified mode name are to be reset to 0.
- **ONE('SNASVCMG')** specifies the SNA-defined mode name, which is used for exchanging the CNOS request and reply when the source LU and target LU are connected by parallel sessions.

RESPONSIBLE specifies which LU is responsible for deactivating the sessions as a result of resetting the session limit for parallel-session connections. This parameter is not applicable to single-session connections or the SNASVCMG sessions.

- **SOURCE** specifies that the source LU is responsible. The target LU cannot negotiate this argument.

- **TARGET** specifies that the target LU is responsible. The target LU can negotiate this argument to SOURCE, in which case the source LU becomes responsible.

The parameters DRAIN_SOURCE and DRAIN_TARGET determine when the responsible LU can deactivate the sessions:

- If an LU is to drain its allocation requests, it continues to allocate conversations to active sessions. The responsible LU deactivates a session only when the conversation allocated to the session is deallocated and no request is awaiting allocation to any session with the specified mode name. The allocation of an awaiting request takes precedence over the deactivation of a session.

- If an LU is not to drain its allocation requests, the responsible LU deactivates a session as soon as the conversation allocated to the session is deallocated. If no conversation is allocated to the session, the responsible LU deactivates the session immediately.

In no case, however, does this verb force deallocation of active conversations.

The RESPONSIBLE and MODE_NAME parameters are interrelated, as follows:

- If MODE_NAME(ALL) is specified, RESPONSIBLE is ignored for those mode names for which the session limit is currently 0.

- If MODE_NAME(ONE(variable)) is specified and the current session limit for that mode name is 0, RESPONSIBLE is ignored.

DRAIN_SOURCE specifies whether the source LU can drain its allocation requests. For parallel-session connections, the target LU cannot negotiate this parameter. This parameter is not applicable to the SNASVCMG sessions.

- **NO** specifies that the source LU cannot drain its allocation requests. All requests currently awaiting allocation, or subsequently issued, at the source LU are rejected with a return code indicating the session limit is 0.

- **YES** specifies that the source LU can drain its allocation requests. The source LU continues to allocate conversations to the sessions until no requests are awaiting allocation, at which time its draining is ended. All allocation requests issued at the source LU after draining is ended are rejected with a return code indicating the session limit is 0.

For parallel-session connections, the DRAIN_SOURCE and MODE_NAME parameters are interrelated, as follows:

- If MODE_NAME(ALL) and DRAIN_SOURCE(YES) are specified, DRAIN_SOURCE is ignored for those mode names for which the session limit is currently 0.

- If MODE_NAME(ALL) and DRAIN_SOURCE(NO) are specified, DRAIN_SOURCE is accepted for all mode names, regardless of the current session limit.

- If MODE_NAME(ONE(variable)) is specified, the current session limit for that mode name is 0, and DRAIN_SOURCE(YES) is currently in effect, DRAIN_SOURCE(NO) if specified causes the source LU's draining to terminate.

- If MODE_NAME(ONE(variable)) is specified, the current session limit for that mode name is 0, and DRAIN_SOURCE(NO) is currently in effect, DRAIN_SOURCE(NO) must be specified.

DRAIN_TARGET specifies whether the target LU can drain its allocation requests. This parameter is not applicable to the SNASVCMG sessions.

- **NO** specifies that the target LU cannot drain its allocation requests. All requests currently awaiting allocation, or subsequently issued, at the target LU are rejected with a return code indicating the session limit is 0. For parallel-session connections, the target LU cannot negotiate this argument.

- **YES** specifies that the target LU can drain its allocation requests. The target LU continues to allocate conversations to the sessions until no requests are awaiting allocation, at which time its draining is ended. All allocation requests issued at the target LU after draining is ended are rejected with a return code indicating the session limit is 0. For parallel-session connections, the target LU can negotiate this argument to NO, in which case the target LU cannot drain its allocation requests.

For parallel-session connections, the DRAIN_TARGET and MODE_NAME parameters are interrelated, as follows:

- If MODE_NAME(ALL) and DRAIN_TARGET(YES) are specified, DRAIN_TARGET is ignored for those mode names for which the session limit is currently 0.

- If MODE_NAME(ALL) and DRAIN_TARGET(NO) are specified, DRAIN_TARGET is accepted for all mode names, regardless of the current session limit.

- If MODE_NAME(ONE(variable)) is specified, the current session limit for that mode name is 0, and DRAIN_TARGET(YES) is currently in effect, DRAIN_TARGET(NO) if specified causes the target LU's draining to terminate.

- If MODE_NAME(ONE(variable)) is specified, the current session limit for that mode name is 0, and DRAIN_TARGET(NO) is currently in effect, DRAIN_TARGET(YES) if specified can be either accepted by the target LU or negotiated to NO. If accepted, the target LU can drain its remaining allocation requests if draining has not already ended.

FORCE specifies whether the source LU is to force the resetting of its session limit when certain error conditions occur that prevent successful exchange of the CNOS request and reply. This parameter is not applicable to single-session connections or the SNASVCMG sessions.

- **NO** specifies that the session limit is to be reset only upon successful completion of the exchange of the CNOS request and reply.

- **YES** specifies that the session limit is to be reset upon either successful or unsuccessful completion of the exchange of the CNOS request and reply, except for certain error conditions (see the RETURN_CODE parameter). If a forced reset occurs, the source LU's session limit is reset, and RESPONSIBLE(SOURCE) and DRAIN_SOURCE(NO) are assumed (regardless of what the respective parameters specify). The target LU's CNOS parameters may not be changed, depending on the error condition and when it occurred during the CNOS exchange.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the program. The return code indicates the result of verb execution. The FORCE parameter determines which of the following return codes can be returned to the program.

- If FORCE(NO) is specified, one of the following return codes is returned:

  - OK (with one of the following subcodes)
    - AS_SPECIFIED
    - AS_NEGOTIATED
  - ALLOCATION_ERROR
  - COMMAND_RACE_REJECT
  - LU_MODE_SESSION_LIMIT_CLOSED
  - PARAMETER_ERROR (for one of the following reasons)
    - Invalid LU name

    - Invalid mode name
  - RESOURCE_FAILURE_NO_RETRY
  - UNRECOGNIZED_MODE_NAME

- If FORCE(YES) is specified, one of the following return codes is returned:

  - OK (with one of the following subcodes)
    - AS_SPECIFIED
    - AS_NEGOTIATED
    - FORCED
  - COMMAND_RACE_REJECT
  - PARAMETER_ERROR (for one of the following reasons)
    - Invalid LU name
    - Invalid mode name

ABEND Conditions:

Parameter Check

- The program issuing this verb does not have CNOS privilege.
- MODE_NAME(ONE('SNASVCMG')) is specified and one or more (LU,mode) session limits for the target LU are not 0.
- MODE_NAME(ONE(variable)) and DRAIN_SOURCE(YES) are specified, the current (LU,mode) session limit is 0, and DRAIN_SOURCE(NO) is currently in effect.
- RESPONSIBLE(TARGET) is specified and not supported.
- DRAIN_TARGET(NO) is specified and not supported.
- FORCE(YES) is specified and not supported.

Notes:

1. The (LU,mode) session limit for a single-session connection to a target LU is reset only at the source LU; a CNOS request and reply are not exchanged between the two LUs. The source LU deactivates the session, if it is active, in accordance with the DRAIN_SOURCE and DRAIN_TARGET parameters.

2. For parallel-session connections, when a mode name is specified other than the SNA-defined mode name, SNASVCMG, or when ALL mode names are indicated, the responsible LU deactivates the sessions associated with the specified mode name, or all mode names other than SNASVCMG, in accordance with the DRAIN_SOURCE and DRAIN_TARGET parameters. The (LU,mode) session limits and contention-winner polarities for all mode names other than SNASVCMG must be reset before issuing this verb with the SNASVCMG mode name specified.

3. When the SNASVCMG mode name is specified, the (LU,mode) session limit and contention-winner polarities for the SNASVCMG mode name are reset only at the source LU; a CNOS request and reply are not exchanged between the two LUs. The source LU deactivates the sessions associated with the SNASVCMG mode name as soon as all other active sessions between the source LU and target LU are deactivated. If no other sessions between the two LUs are active, the source LU immediately deactivates the sessions associated with the SNASVCMG mode name.

4. This verb can be issued when the (LU,mode) session limit is 0. In particular, this verb may be issued multiple times without issuing an intervening INITIALIZE_SESSION_LIMIT verb. For example, if this verb was first issued specifying DRAIN_TARGET(YES) and subsequently it is decided to disallow further draining by the target LU, this verb can be issued a second time specifying DRAIN_TARGET(NO). When the (LU,mode) session limit is already 0, the RESPONSIBLE parameter is ignored; the LU (SOURCE or TARGET) specified on the first RESET_SESSION_LIMIT remains responsible for deactivating sessions.

PROCESS_SESSION_LIMIT

Processes the session limit, contention-winner polarities, and related CNOS parameters from the source LU and, if necessary, negotiates them to values acceptable to the target LU.

PROCESS_SESSION_LIMIT	Supplied Parameters: RESOURCE (variable)
	Returned Parameters: LU_NAME (variable) MODE_NAME (variable1 variable2) RETURN_CODE (variable)
	;

Supplied Parameters:

RESOURCE specifies the resource ID of the conversation that started this program.

Returned Parameters:

LU_NAME specifies the variable in which is returned the name of the source LU.

MODE_NAME specifies the variables in which are returned an indication of whether one or all mode names are affected, and, if one, the specific mode name.

- **variable1** is the variable in which is returned an indication of whether one or all mode names associated with the source LU are affected.
  - ONE indicates a specific mode name is affected. The mode name is returned in variable2.
  - ALL indicates all mode names are affected. Nothing is placed in variable2.
- **variable2** is the variable in which is returned the specific mode name when only one is affected.

RETURN_CODE specifies the variable in which a return code is returned to the program. The return code indicates the result of verb execution.

- OK (with one of the following subcodes)
  - AS_SPECIFIED
  - AS_NEGOTIATED
- RESOURCE_FAILURE_NO_RETRY

ABEND Conditions:

Parameter Check

The program issuing this verb is not the SNA service transaction program identified as hex 06F1.

Notes:

1. This verb applies only to parallel session connections.
2. This verb is issued by an SNA service transaction program called the "CNOS service transaction program," identified with the name of hex 06F1. The CNOS service transaction program is invoked at the target LU as a result of a CNOS verb being issued at the source LU. The CNOS service transaction program then issues this

verb in order to initiate the target LU's processing of the CNOS request sent by the source LU.

3. The program issues the DISPLAY_MODE verb in order to obtain the new session limit, contention-winner polarities, and related CNOS parameters.

SESSION CONTROL VERBS

This subcategory of control-operator verbs consists of two verbs used for session control, one that activates an LU-LU session and one that deactivates an LU-LU session. The LU executing the verb is designated the source LU and is responsible for the session activation or deactivation. The other LU for the session is the target LU. These verbs are:

ACTIVATE_SESSION
DEACTIVATE_SESSION

Only a transaction program that has session-control privilege may issue these verbs. The program is designated to have session-control privilege when it is defined to the local LU (see the DEFINE_TP verb).

The detailed descriptions of these verbs follows.

ACTIVATE_SESSION

Activates a session with the specified mode name to the target LU. The session is activated as a contention winner for either the source LU or target LU.

ACTIVATE_SESSION	Supplied Parameters: LU_NAME (variable) MODE_NAME (variable) ('SNASVCMG')
	Returned Parameters: RETURN_CODE (variable)
	;

Supplied Parameters:

LU_NAME specifies the name of the target LU to which the session is to be activated. This LU name is any name by which the source LU knows the target LU for the purpose of activating a session. The source LU transforms this locally-known LU name to an LU name used by the network, if the names are different.

MODE_NAME specifies the mode name for the session.

- **variable** contains the mode name.
- **'SNASVCMG'** specifies the SNA-defined mode name, which is used for exchanging the CNOS request and reply when the source LU and target LU are connected by parallel sessions.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the program. The return code indicates the result of verb execution.

- OK (with one of the following subcodes)
  - AS_SPECIFIED
  - AS_NEGOTIATED
- ACTIVATION_FAILURE_NO_RETRY
- ACTIVATION_FAILURE_RETRY
- PARAMETER_ERROR (for one of the following reasons)
  - Invalid LU name.
  - Invalid mode name.
- LU_MODE_SESSION_LIMIT_EXCEEDED

ABEND Conditions:

Parameter Check

- This verb is not supported.
- The program issuing this verb does not have session-control privilege.

Notes:

1. This verb can be used to activate a single session as a contention winner for either the source LU or the target LU. The LU to be the contention winner is established by means of the INITIALIZE_SESSION_LIMIT verb.

2. This verb can be used to activate one or both parallel sessions for the SNASVCMG mode name to a target LU. The source LU is the contention winner for the first session; the target LU is the contention winner for the second session.

3. This verb can be used to activate a parallel session as a contention winner for either the source LU or the target LU. The session is activated as a contention winner for the source LU when the number of currently active contention-winner sessions for the source LU is less than the new (LU,mode) session limit minus the new minimum number of contention-winner sessions for the target LU. Otherwise, the session is activated as a contention winner for the target LU.

DEACTIVATE_SESSION

Deactivates the specified LU-LU session. The type of deactivation can be cleanup or normal.

DEACTIVATE_SESSION	Supplied Parameters: SESSION_ID (variable) [TYPE (CLEANUP) (NORMAL)]
	Returned Parameters: RETURN_CODE (variable)
	;

Supplied Parameters:

SESSION_ID specifies the identifier of the LU-LU session to be deactivated.

TYPE specifies the type of deactivation.

- **CLEANUP** specifies that the session is to be deactivated immediately, regardless of whether a conversation is currently allocated to the session.
- **NORMAL** specifies that the session is to be deactivated normally, after the conversation currently allocated to the session is deallocated. If no conversation is currently allocated to the session, normal deactivation begins immediately.

Returned Parameters:

RETURN_CODE specifies the variable in which a return code is returned to the program. The return code indicates the result of verb execution.

- OK
- PARAMETER_ERROR (for the following reason)
  - The specified session identifier is not assigned to a currently active session.

ABEND Conditions:

Parameter Check

- This verb is not supported.
- The program issuing this verb does not have session-control privilege.

LU DEFINITION VERBS

This subcategory of control-operator verbs consists of the following verbs, which are used to define or modify the local LU's operating parameters, examine the parameters, and delete the parameters. These verbs are:

```
DEFINE_LOCAL_LU
DEFINE_REMOTE_LU
DEFINE_MODE
DEFINE_TP
DISPLAY_LOCAL_LU
DISPLAY_REMOTE_LU
DISPLAY_MODE
DISPLAY_TP
DELETE
```

The execution of these verbs involves only the local LU. They do not cause any information to be sent outside the LU.

Some of the local LU's operating parameters can be added, modified, or deleted only under appropriate conditions. An attempt to alter any of these parameters when conditions are inappropriate is an error, causing the LU to return the PARAMETER_ERROR return code on the verb. The parameters that are restricted in this way and the corresponding errors are identified in the verb descriptions.

The DEFINE verbs may be issued multiple times to initialize or update the local LU's operating parameters. The first time a verb parameter is specified, the LU's corresponding operating parameter is initialized; thereafter, it is changed. The following notes apply to all verb parameters, except where stated otherwise in the individual verb descriptions:

- If the LU's operating parameter is not currently defined and the corresponding verb parameter is specified, the operating parameter is initialized with the supplied value.
- If the LU's operating parameter is not currently defined and the corresponding verb parameter is omitted, the operating parameter remains undefined.
- If the LU's operating parameter is currently defined and the corresponding verb parameter is specified, the operating parameter value is replaced with the supplied value.
- If the LU's operating parameter is currently defined and the corresponding verb parameter is omitted, the operating parameter value remains unchanged.

The DISPLAY verbs return current values of the local LU's operating parameters. When a DISPLAY verb is issued specifying a parameter that is not currently defined at the local LU, a null value is returned.

The DELETE verb deletes the local LU's operating parameters. After a parameter is deleted, it is no longer defined at the local LU.

Only a transaction program that has define privilege may issue the DEFINE verbs and DELETE verb, and only a program that has display privilege may issue the DISPLAY verbs. The program is designated to have define or display privilege when it is defined to the local LU (see the DEFINE_TP verb). The program that initially establishes define privilege for other programs has implicit define privilege.

The detailed descriptions of these verbs follow.

DEFINE_LOCAL_LU

Defines the fully qualified name for the local LU, and initializes or changes parameters that control the operation of the local LU.

```
DEFINE_LOCAL_LU

Supplied Parameters:

FULLY_QUALIFIED_LU_NAME ( variable )

[ LU_SESSION_LIMIT ( NONE )                                    ]
[                  ( VALUE ( variable ) ) ]

[          ( ADD ( USER_ID ( variable )                        ]
[                  PASSWORD ( variable )                       ]
[ SECURITY         PROFILE ( variable ) ) )                    ]
[          ( DELETE ( USER_ID ( variable )                     ]
[                     PROFILE ( variable ) ) )                 ]

[ MAP_NAME ( ADD ( variable ) )                                ]
[          ( DELETE ( variable ) ) ]

Returned Parameters:

RETURN_CODE ( variable )

;
```

Supplied Parameters:

FULLY_QUALIFIED_LU_NAME specifies the fully qualified name of the local LU. If the specified name is not currently defined, this verb defines the name by which the local LU is known throughout the network, replacing the current name if one exists, and initializes the LU-LU session limit. If the specified name is already defined, this verb changes the other parameter values.

LU_SESSION_LIMIT specifies the LU-LU session limit for the total number of sessions for the local LU.

- **NONE** specifies that no limit is to be defined.
- **VALUE** specifies a number representing the LU-LU session limit.

SECURITY specifies to add or delete access security information that the local LU uses for conversation-level security verification of incoming allocation requests on an LU-wide basis. (Contrast this parameter with the SECURITY_REQUIRED and SECURITY_ACCESS parameters of the DEFINE_TP verb.) The local LU updates a conversation-level security verification list from the information supplied on this parameter. The verification list consists of one or more user IDs and corresponding passwords, and zero or more profiles associated with each user ID.

- **ADD** specifies to add access security information to the LU's conversation-level security verification list. A user ID must be specified together with either, or both, a password or profile.
  - **USER_ID** specifies a user ID. If the user ID is not currently defined, a password must also be specified, and the LU adds the user ID, password, and profile (if specified) to its conversation-level security verification list. If the user ID is already defined, the list is updated with the password or profile.
  - **PASSWORD** specifies the password for this user ID. If the user ID is already defined, the password replaces the one currently defined.

  - **PROFILE** specifies a profile for this user ID. If the user ID is already defined, the profile is added to the ones currently defined.

- **DELETE** specifies to delete access security information from the LU's conversation-level security verification list. The user ID may be specified alone or together with a profile.

  - **USER_ID** specifies the user ID. If a profile is not specified, the user ID and its associated password and profiles are deleted. If a profile is also specified, the user ID and its associated password and other profiles remain defined.

  - **PROFILE** specifies the profile to be deleted.

MAP_NAME specifies to add or delete a map name that the local LU is to support for local data mapping. Local transaction programs may specify this map name on the MAP_NAME parameter of the MC_SEND_DATA verb, and remote LUs may send this map name to the local LU over mapped conversations.

- **ADD** specifies the map name to be added.

- **DELETE** specifies the map name to be deleted.

Returned Parameters:

RETURN_CODE returns an indication of the result of verb execution.

- OK
- PARAMETER_ERROR (for one of the following reasons)
  - FULLY_QUALIFIED_LU_NAME specifies a value that is not a type-A symbol string.
  - LU_SESSION_LIMIT(VALUE(variable)) specifies a value that is less than the sum of the (LU,mode) session limits currently in effect.
  - SECURITY(ADD(...)) specifies a user ID, password, or profile that is not a symbol-string type (A, AE, GR, or DB) that the product supports.
  - SECURITY(ADD(...)) specifies only a user ID, or a password or profile but no user ID.
  - SECURITY(DELETE(...)) specifies a user ID or profile that is not currently defined at the local LU.
  - SECURITY(DELETE(...)) specifies only a profile.
  - MAP_NAME(ADD(variable)) specifies a map name that is not a symbol-string type (A, AE, or GR) that the product supports.
  - MAP_NAME(DELETE(variable)) specifies a map name that is not currently defined at the local LU.

ABEND Conditions:

Parameter Check

- This verb is not supported.
- The program issuing this verb does not have define privilege.

Notes:

1. This verb can be used to define the name by which the local LU is known throughout the network. To use it for this purpose, the verb should be issued prior to the local LU's participation in any network activity, such as initializing (LU,mode) session limits.

2. The LU-LU session limit is the maximum number of sessions that the local LU can have active at a time. It represents the upper bound on the sum of the (LU,mode) session limits, and it must be equal to or greater than the sum of the (LU,mode) session limits currently in effect; see the description of the return code, LU_SESSION_LIMIT_EXCEEDED, in "Return Codes" on page 5-51 for more details.

3. If no LU-LU session limit is defined at the local LU, the upper bound, if any, on the sum of the (LU,mode) session limits is

product-determined. For example, the upper bound may be a fixed value or determined by an algorithm.

4. When the first user ID and associated password and profiles are added, the LU's conversation-level security verification list is created. When the last user ID and associated password and profiles are deleted, the conversation-level security verification list itself is deleted.

5. The local LU uses the conversation-level security verification list to verify the access security information on allocation requests it receives. Specifically, when the LU receives an allocation request carrying a user ID and password, it verifies that the user ID and password are present in its conversation-level security verification list. If the allocation request also carries a profile, the LU verifies that the profile is also present in the list. Allocation requests that carry no access security information, or that carry a user ID and an already-verified indication (and may also carry a profile), are not verified against the conversation-level security verification list. However, these requests may be subject to resource-access verification, as determined by the SECURITY_ACCESS parameter on DEFINE_TP.

6. If the conversation-level security verification list does not exist, the local LU will perform no conversation-level security verification. Allocation requests that carry a user ID and password will not be accepted.

7. If no map names are defined at the local LU, it will perform no data mapping.

8. More details concerning the LU's use of these operating parameters are given in SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2.

DEFINE_REMOTE_LU

Initializes or changes parameters that control the operation of the local LU in conjunction with a remote LU.

```
DEFINE_REMOTE_LU

Supplied Parameters:

FULLY_QUALIFIED_LU_NAME ( variable )

[ LOCALLY_KNOWN_LU_NAME ( NONE )                                ]
[                       ( NAME ( variable ) )                   ]

[ UNINTERPRETED_LU_NAME ( NONE )                                ]
[                       ( NAME ( variable ) )                   ]

[ INITIATE_TYPE ( INITIATE_ONLY )                               ]
[               ( INITIATE_OR_QUEUE )                           ]

[ PARALLEL_SESSION_SUPPORT ( YES )                              ]
[                          ( NO )                               ]

[ CNOS_SUPPORT ( YES )                                          ]
[              ( NO )                                           ]

[ LU_LU_PASSWORD ( NONE )                                       ]
[                ( VALUE ( variable ) )                         ]

[                      ( NONE )                                 ]
[ SECURITY_ACCEPTANCE  ( CONVERSATION )                         ]
[                      ( ALREADY_VERIFIED )                     ]

Returned Parameters:

RETURN_CODE ( variable )

;
```

Supplied Parameters:

FULLY_QUALIFIED_LU_NAME specifies the fully qualified name of the remote LU. If the specified name is currently undefined to the local LU, this verb defines the remote LU's fully qualified name and initializes the other parameter values specified on this verb. If the specified name is already defined to the local LU, this verb changes the other parameter values.

LOCALLY_KNOWN_LU_NAME specifies the locally-known name of the remote LU that local transaction programs can specify on the LU_NAME parameter of the MC_ALLOCATE and ALLOCATE verbs.

- **NONE** specifies that no locally-known LU name is to be defined.
- **NAME** specifies the locally-known LU name of the remote LU. This name is not sent outside the local LU.

UNINTERPRETED_LU_NAME specifies the uninterpreted LU name of the remote LU, which the local LU uses on INITIATE and TERMINATE requests it sends to its SSCP.

- **NONE** specifies that no uninterpreted LU name is to be defined.
- **NAME** specifies the uninterpreted LU name of the remote LU.

INITIATE_TYPE specifies the session-initiation type that the local LU is to use on INITIATE requests it sends to its SSCP for initiating sessions with the remote LU.

- **INITIATE_ONLY** specifies that session initiation requests are to indicate "initiate only." The SSCP will not queue the session initiation requests.

- **INITIATE_OR_QUEUE** specifies that session initiation requests are to indicate "initiate or queue." The SSCP may queue session the initiation requests, if necessary, while waiting for the remote LU to become available.

PARALLEL_SESSION_SUPPORT specifies whether the local LU supports parallel sessions with the remote LU. The local LU uses this parameter to determine the indication for parallel session support that it specifies in session activation (BIND) requests and responses.

- **YES** specifies that parallel sessions are supported.

- **NO** specifies that parallel sessions are not supported.

CNOS_SUPPORT specifies whether the local LU supports the exchange of CNOS requests and replies with the remote LU. The local LU uses this parameter to determine the indication for CNOS support that it specifies in session activation (BIND) requests and responses.

- **YES** specifies that CNOS is supported. PARALLEL_SESSION_SUPPORT(YES) must also be specified.

- **NO** specifies that CNOS is not supported. PARALLEL_SESSION_SUPPORT(NO) must also be specified.

LU_LU_PASSWORD specifies the LU-LU password to be used for session-level LU-LU verification during session activation. The LU-LU password must be the same as that defined at the remote LU.

- **NONE** specifies that no LU-LU password is to be defined.

- **NAME** specifies the LU-LU password. It must be a random binary value up to 64 bits (8 bytes) in length. It should be specified in a form that can yield any binary value. For example, it could be specified using the hexadecimal digits 0, 1, 2, ..., E, F to represent each group of 4 bits. After being defined, the LU-LU password is nondisplayable.

SECURITY_ACCEPTANCE specifies the level of access security information that the local LU will accept on allocation requests it receives from the remote LU. Access security information that includes a password is verified against the LU's conversation-level security verification list prior to acceptance, as described for the SECURITY parameter on the DEFINE_LOCAL_LU verb.

- **NONE** specifies that no access security information is to be accepted on allocation requests received from the remote LU.

- **CONVERSATION** specifies that the local LU will accept conversation-level access security information, which must include both a user ID and password, and may also include a profile. The local LU will not accept allocation requests that include the already-verified indication.

- **ALREADY_VERIFIED** specifies that the local LU will accept conversation-level access security information, which may include the already-verified indication in place of a password.

Returned Parameters:

RETURN_CODE returns an indication of the result of verb execution.

- OK
- PARAMETER_ERROR (for one of the following reasons)
  - FULLY_QUALIFIED_LU_NAME specifies a value that is not a type-A symbol string.

- PARALLEL_SESSION_SUPPORT(YES) is specified and the total LU-LU session limit is 1.
- PARALLEL_SESSION_SUPPORT(YES) and CNOS_SUPPORTED(NO) are specified.
- PARALLEL_SESSION_SUPPORT(NO) and CNOS_SUPPORTED(YES) are specified.
- PARALLEL_SESSION_SUPPORT, CNOS_SUPPORT, LU_LU_PASSWORD, or SECURITY_ACCEPTANCE is specified and at least one (LU,mode) session limit for the remote LU is not zero, or the LU-LU session count between the local and remote LUs is not zero.

ABEND Conditions:

Parameter Check

- This verb is not supported.
- The program issuing this verb does not have define privilege.

Notes:

1. This verb can be used to define the fully qualified name of a remote LU. In this case, the verb should be issued prior to the local LU participating in any network activity involving the remote LU.

2. If no locally-known name of the remote LU is defined at the local LU, local transaction programs may specify the remote LU's fully qualified LU name, or its uninterpreted name if one is currently defined at the local LU, on the LU_NAME parameter of the MC_ALLOCATE and ALLOCATE verbs.

3. An uninterpreted name of the remote LU must be defined at both the local LU and its SSCP before the local LU sends INITIATE and TERMINATE requests to its SSCP.

4. If no initiate type is defined at the local LU for the remote LU and the product LU sends INITIATE requests to its SSCP, the type used on the requests is product-determined.

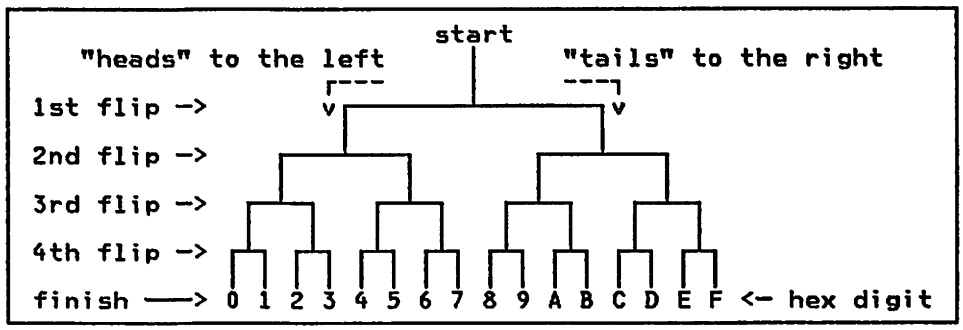
5. Parallel-session support must be defined at the local LU for the remote LU before it activates sessions with the remote LU.

6. CNOS support must be defined at the local LU for the remote LU before it activates sessions with the remote LU.

7. Session-level LU-LU verification is used to verify the identity of each LU to its session partner LU during activation of an LU-LU session. It uses an LU-LU password as the key to the Data Encryption Standard (DES) algorithm, in conjunction with an LU-generated random-data value carried on the session-activation request and response.

   The same LU-LU password specification must be defined at both LUs, either NONE or an LU-LU password. An LU-LU password should be a random binary value. The means for specifying the LU-LU password is product-dependent. The product may provide a utility procedure for generating an LU-LU password, or it may require the user to enter the password manually. In the latter case, the human operator may use the following method to produce a random value:

   - Enter the password value by means of hex digits (the numerals 0 through 9 and the upper-case characters A through F).
   - Enter 16 random hex digits (fewer digits or non-random digits will reduce the effective security for LU-LU verification).
   - For each of the 16 hex digits, flip a coin four times. At each flip of the coin, follow the path illustrated in the following figure. The fourth flip will select the hex digit to be used. Repeat this procedure until all 16 hex digits are obtained. The result is a 64-bit random LU-LU password.



The total number of coin flips is 64. Of course, the equivalent value for the LU-LU password can be obtained in binary notation, where each coin flip, of "heads" or "tails," selects the next binary digit, 0 or 1, respectively.

8. If no LU-LU password is defined at the local LU for the remote LU, no session-level LU-LU verification will take place between the two LUs.

9. Conversation-level access security information is carried on allocation requests in order for the receiving LU to verify the identity of the user ID, and to control access to its resources. The information includes a user ID together with a password or the already-verified indication; the information may also include a profile. Allocation requests that include a password are verified against the LU's conversation-level security verification list; see the SECURITY parameter on the DEFINE_LOCAL_LU verb for more details about conversation-level security verification. The already-verified indication signifies that the identity of the user ID has already been verified.

10. If no conversation-level security is defined at the local LU for the remote LU, the local LU will accept from the remote LU only allocation requests that carry no access security information.

11. More details concerning the LU's use of these operating parameters are given in SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2.

DEFINE_MODE

Initializes or changes parameters that control the operation of the local LU in conjunction with a group of sessions to the specified remote LU, the session group being identified by a mode name.

```
DEFINE_MODE

Supplied Parameters:

FULLY_QUALIFIED_LU_NAME ( variable )

MODE_NAME ( variable )

[ SEND_PACING_WINDOW ( variable ) ]

[ RECEIVE_PACING_WINDOW ( variable ) ]

[ SEND_MAX_RU_SIZE_LOWER_BOUND ( variable ) ]

[ SEND_MAX_RU_SIZE_UPPER_BOUND ( variable ) ]

[ RECEIVE_MAX_RU_SIZE_LOWER_BOUND ( variable ) ]

[ RECEIVE_MAX_RU_SIZE_UPPER_BOUND ( variable ) ]

[ SYNC_LEVEL_SUPPORT ( CONFIRM )
                     ( CONFIRM_SYNCPT ) ]

[                                ( OPERATOR )
  SINGLE_SESSION_REINITIATION    ( PLU )
                                 ( SLU )
                                 ( PLU_OR_SLU ) ]

[ SESSION_LEVEL_CRYPTOGRAPHY ( NO )
                             ( YES ) ]

[ CONWINNER_AUTO_ACTIVATE_LIMIT ( variable ) ]

Returned Parameters:

RETURN_CODE ( variable )

;
```

Supplied Parameters:

FULLY_QUALIFIED_LU_NAME specifies the fully qualified name of the remote LU to which the other parameters of this verb apply.

MODE_NAME specifies the mode name for the group of sessions to which the remaining parameters of this verb apply. If the specified name is currently undefined at the local LU for the remote LU, this verb defines the mode name and initializes the other parameter values specified on this verb. If the specified name is already defined at the local LU for the remote LU, this verb changes the other parameter values.

SEND_PACING_WINDOW specifies the pacing window size to be used on the sessions for normal-flow requests that the local LU sends. The local LU uses this parameter to determine the values for its send window size and the remote LU's receive window size that it specifies in session activation (BIND) requests.

RECEIVE_PACING_WINDOW specifies the pacing window size to be used on the sessions for normal-flow requests that the local LU receives. The local LU uses this parameter to determine the values for its receive window size and the remote LU's send window size that it specifies in session activation (BIND) requests and responses.

SEND_MAX_RU_SIZE_LOWER_BOUND specifies the lower bound for the maximum size of normal-flow requests that the local LU sends on the sessions. This value must be less than or equal to the value specified on SEND_MAX_RU_SIZE_UPPER_BOUND. The local LU uses these lower- and upper-bound values to determine the value for its send maximum RU size that it specifies in session activation (BIND) requests and responses.

SEND_MAX_RU_SIZE_UPPER_BOUND specifies the upper bound for the maximum size of normal-flow requests that the local LU sends.

RECEIVE_MAX_RU_SIZE_LOWER_BOUND specifies the lower bound for the maximum size of normal-flow requests that the local LU receives on the sessions. This value must be less than or equal to the value specified on RECEIVE_MAX_RU_SIZE_UPPER_BOUND. The local LU uses these lower- and upper-bound values to determine the value for its receive maximum RU size that it specifies in session activation (BIND) requests and responses.

RECEIVE_MAX_RU_SIZE_UPPER_BOUND specifies the upper bound for the maximum size of normal-flow requests that the local LU receives.

SYNC_LEVEL_SUPPORT specifies the synchronization levels that the local LU supports for conversations allocated to the sessions. The local LU uses this parameter to determine the indication for the synchronization level that it specifies in session activation (BIND) requests and responses.

- **CONFIRM** specifies that conversations may use a synchronization level of NONE or CONFIRM.
- **CONFIRM_SYNCPT** specifies that conversations may use a synchronization level of NONE, CONFIRM, or SYNCPT.

SINGLE_SESSION_REINITIATION specifies the responsibility for session reinitiation of a single session with the remote LU. The local LU uses this parameter to determine the indication for session reinitiation responsibility that it specifies in session activation (BIND) requests and responses. The remote LU must be defined to not support parallel sessions (see the PARALLEL_SESSION_SUPPORT parameter on the DEFINE_REMOTE_LU verb).

- **OPERATOR** specifies that neither LU will automatically attempt to reinitiate the session. If a reinitiation race occurs, where the operators at both LUs attempt to reinitiate the session at the same time, the reinitiation is successfully completed by the LU with the greater fully qualified LU name (provided no session activation errors are encountered). The comparison of the fully qualified LU names is based on their hexadecimal values.
- **PLU** specifies that the primary LU will automatically attempt to reinitiate the session.
- **SLU** specifies that the secondary LU will automatically attempt to reinitiate the session.
- **PLU_OR_SLU** specifies that either LU may automatically attempt to reinitiate the session. A reinitiation race between the two LUs is resolved in the same way as for operator reinitiation.

SESSION_LEVEL_CRYPTOGRAPHY specifies whether the local LU supports session-level cryptography for the sessions. The local LU uses this parameter to determine the indication for cryptography support that it specifies in session activation (BIND) requests and responses.

- **NONE** specifies that no session-level cryptography is to be used.

- **MANDATORY** specifies that session-level mandatory cryptography is to be used on all FMD requests flowing on the sessions.

CONWINNER_AUTO_ACTIVATE_LIMIT specifies the automatic-activation limit on the number of contention-winner sessions that the local LU can automatically activate when the minimum number of contention-winner sessions for the local LU increases (as a result of CNOS processing). The actual limit on the number of contention-winner sessions automatically activated is the lesser of the value specified on this parameter and the new minimum number of contention-winner sessions for the local LU.

A value of 0 specifies that the local LU is to automatically activate no sessions.

Returned Parameters:

RETURN_CODE returns an indication of the result of verb execution.

- OK
- PARAMETER_ERROR (for one of the following reasons)
  - FULLY_QUALIFIED_LU_NAME does not specify a remote LU name defined at the local LU.
  - FULLY_QUALIFIED_LU_NAME specifies a value that is not a type-A symbol string.
  - MODE_NAME specifies a value that is not a type-A symbol string.
  - SEND_MAX_RU_SIZE_LOWER_BOUND specifies a value exceeding that on SEND_MAX_RU_SIZE_UPPER_BOUND.
  - RECEIVE_MAX_RU_SIZE_LOWER_BOUND specifies a value exceeding that on RECEIVE_MAX_RU_SIZE_UPPER_BOUND.
  - SINGLE_SESSION_REINITIATION is specified for a remote LU that is currently defined as supporting parallel sessions.
  - A parameter other than CONWINNER_AUTO_ACTIVATE_LIMIT is specified and the (LU,mode) session limit and count for the mode name are not zero.

ABEND Conditions:

Parameter Check

- This verb is not supported.
- The program issuing this verb does not have define privilege.

Notes:

1. If the mode name is currently defined, the (LU,mode) session limit and count must be zero when this verb is issued specifying any of the parameters other than CONWINNER_AUTO_ACTIVATE_LIMIT. The auto-activation limit on the number of contention-winner sessions may be initialized or changed at anytime.

2. If no send or receive pacing window size is defined, a product-determined window size is used.

3. If no lower bound is defined for the send or receive maximum size of normal-flow requests, a product-determined lower bound is used.

4. If no upper bound is defined for the send or receive maximum size of normal-flow requests, a product-determined upper bound is used.

5. If no synchronization level support is defined for the mode name, conversations may use NONE or CONFIRM.

6. If no responsibility for session reinitiation of a single session is defined for the mode name, a product-determined responsibility is used.

7. If no session-level cryptography is defined for the mode name, none is used.

8. If no automatic-activation limit for contention-winner sessions is defined for the mode name, the local LU may not automatically activate any sessions, or it may automatically activate sessions up to the minimum number of the LU's contention-winner sessions currently in effect, depending on the product.

9. More details concerning the LU's use of these operating parameters are given in SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2.

DEFINE_TP

Initializes or changes parameters that control the operation of the local LU in conjunction with a transaction program.

```
DEFINE_TP

Supplied Parameters:

TP_NAME ( variable )

[ STATUS ( ENABLED )
         ( TEMP_DISABLED )
         ( PERM_DISABLED ) ]

[ CONVERSATION_TYPE ( MAPPED | BASIC ) ]

[ SYNC_LEVEL ( NONE | CONFIRM | SYNCPT ) ]

[ SECURITY_REQUIRED ( NONE )
                    ( CONVERSATION )
                    ( ACCESS ( PROFILE ) )
                    ( ACCESS ( USER_ID ) )
                    ( ACCESS ( USER_ID_PROFILE ) ) ]

[ SECURITY_ACCESS ( ADD ( USER_ID ( variable )
                          PROFILE ( variable ) ) )
                  ( DELETE ( USER_ID ( variable )
                             PROFILE ( variable ) ) ) ]

[ PIP ( NO )
      ( YES ( variable ) ) ]

[ DATA_MAPPING ( NO )
               ( YES ) ]

[ FMH_DATA ( NO )
           ( YES ) ]

[ PRIVILEGE ( NONE )
            ( CNOS | SESSION_CONTROL | DEFINE | DISPLAY |
              ALLOCATE_SERVICE_TP ) ]

Returned Parameters:

[ RETURN_CODE ( variable ) ]

;
```

Supplied Parameters:

TP_NAME specifies the local transaction program name. If the specified name is not currently defined at the local LU, this verb defines the program name and initializes the other parameter values specified on this verb. If the specified name is already defined to the local LU, this verb changes the other parameter values.

STATUS specifies the status for starting execution of the transaction program when the local LU receives an allocation request naming the program.

- **ENABLED** specifies that the local LU can start the program.

- **TEMP_DISABLED** specifies that the local LU cannot start the program. The local LU rejects the allocation request with an indication that the program is not available but retry is possible.

- **PERM_DISABLED** specifies that the local LU cannot start the program. The local LU rejects the allocation request with an indication that the program is not available and no retry is possible.

CONVERSATION_TYPE specifies the conversation type allowed on allocation requests that start the transaction program.

- **MAPPED** specifies that allocation requests indicating mapped conversation are allowed to start the program.

- **BASIC** specifies that allocation requests indicating basic conversation are allowed to start the program.

One or both of these arguments may be specified.

SYNC_LEVEL specifies the synchronization level allowed on allocation requests that start the transaction program.

- **NONE** specifies that allocation requests indicating a synchronization level of none are allowed to start the program.

- **CONFIRM** specifies that allocation requests indicating a synchronization level of confirm are allowed to start the program.

- **SYNCPT** specifies that allocation requests indicating a synchronization level of sync point are allowed to start the program.

Any combination of these arguments may be specified.

SECURITY_REQUIRED specifies the type of security verification required to be performed on incoming allocation requests that designate the transaction program. (Conversation-level security verification, when required, is performed as specified on the SECURITY parameter of the DEFINE_LOCAL_LU verb.)

- **NONE** specifies that no verification is required. Allocation requests designating the transaction program may omit or include access security information. Conversation-level security verification will be performed on those requests that include a user ID and password, but no resource-access verification is performed.

- **CONVERSATION** specifies that conversation-level security verification is to be performed on requests that carry a user ID and password, but no resource-access verification is performed. Allocation requests designating the transaction program must carry a user ID and either a password or an already-verified indication. (Acceptance of the already-verified indication is determined by the SECURITY_ACCEPTANCE parameter of the DEFINE_REMOTE_LU verb.)

- **ACCESS** specifies that conversation-level security verification is to be performed on requests that carry a user ID and password, and resource-access verification is also to be performed. Allocation requests designating the transaction program must carry a user ID and either a password or an already-verified indication. (Acceptance of the already-verified indication is determined by the SECURITY_ACCEPTANCE parameter of the DEFINE_REMOTE_LU verb.) The local LU performs resource-access verification using a resource-access authorization list associated with the transaction program. The list is created by means of the SECURITY_ACCESS parameter. The type of resource-access verification to be performed is specified as follows:

  - **PROFILE** specifies that the profile carried on the allocation request is to be verified against the resource-access authorization list. The allocation request must carry a profile that matches one in the authorization list. The user ID on the allocation request is ignored for the resource-access verification.

- **USER_ID** specifies that the user ID carried on the allocation request is to be verified against the resource-access authorization list. The allocation request must carry a user ID that matches one in the authorization list. The profile (if present) on the allocation request is ignored for the resource-access verification.

- **USER_ID_PROFILE** specifies that the user ID and profile carried on the allocation request are to be verified against the resource-access authorization list. The allocation request must carry a user ID and profile that match a user ID and associated profile in the authorization list.

SECURITY_ACCESS specifies to add or delete access security information that the local LU uses for resource-access verification. This parameter must be specified when the SECURITY_REQUIRED parameter specifies that resource-access verification is required. The local LU updates a resource-access authorization list, associated with the transaction program, from the information supplied on this parameter. The resource-access authorization list consists of either (1) one or more profiles, or (2) one or more user IDs with zero or more profiles associated with each user ID.

- **ADD** specifies to add access security information to the resource-access authorization list associated with the transaction program. A profile may be specified alone, or a user ID may be specified alone or together with a profile.

  - **USER_ID** specifies a user ID. A user ID must be specified when the SECURITY_REQUIRED parameter specifies resource-access verification that includes verification of user IDs. A profile may also be specified, depending on the SECURITY_REQUIRED parameter. If the user ID is not currently defined for the transaction program, the LU adds it and the profile (if specified) to the resource-access authorization list. If the user ID is already defined, the list is updated with the profile.

  - **PROFILE** specifies a profile to be added to the resource-access authorization list. A profile may be specified only when the SECURITY_REQUIRED parameter specifies resource-access verification that includes verification of profiles. A profile must be specified alone when the resource-access verification includes verification of only profiles. A user ID must also be specified when the resource-access verification includes verification of both user IDs and profiles. If a user ID is also specified, the profile is added to those associated with the user ID.

- **DELETE** specifies to delete access security information from the resource-access authorization list associated with the transaction program. A profile may be specified alone, or a user ID may be specified alone or together with a profile.

  - **USER_ID** specifies a user ID. A user ID must be specified when the SECURITY_REQUIRED parameter specifies resource-access verification that includes verification of user IDs. A profile may also be specified, depending on the SECURITY_REQUIRED parameter. If a user ID is specified alone, the user ID and all of its associated profiles are deleted from the resource-access authorization list. If a profile is also specified, the user ID and profile are deleted when no other profiles are currently defined for the user ID; if other profiles are currently defined for the user ID, the user ID and other profiles remain defined.

  - **PROFILE** specifies a profile to be deleted from the resource-access authorization list. A profile may be specified only when the SECURITY_REQUIRED parameter specifies resource-access verification that includes verification of profiles. A profile must be specified alone when the resource-access verification includes verification of only profiles. A user ID must also be specified when the SECURI-

TY_REQUIRED parameter specifies resource-access verification that includes verification of both user IDs and profiles.

PIP specifies whether PIP data is required on allocation requests that start the transaction program.

- **NO** specifies that no PIP data is required. Only allocation requests carrying no PIP data are allowed to start the program.
- **YES** specifies that PIP data is required. Only allocation requests carrying the number of PIP subfields specified on this parameter are allowed to start the program. The specified number should agree with the number of PIP variables associated with the program. For more information about the association of PIP variables with the program, see "Transaction Program Structure and Execution" in Chapter 3.

DATA_MAPPING specifies whether data mapping support is to be provided to the transaction program. This parameter applies only when CONVERSATION_TYPE(MAPPED) or CONVERSATION_TYPE(MAPPED|BASIC) is also specified.

- **NO** specifies that no data mapping support is to be provided. Map names received on any mapped conversations allocated to the program are rejected.
- **YES** specifies that data mapping support is to be provided.

FMH_DATA specifies whether FMH data support is to be provided to the transaction program. This parameter applies only when CONVERSATION_TYPE(MAPPED) or CONVERSATION_TYPE(MAPPED|BASIC) is also specified.

- **NO** specifies that no FMH data support is to be provided. FMH data received on any mapped conversations allocated to the program is rejected.
- **YES** specifies that FMH data support is to be provided.

PRIVILEGE specifies the category of control operator verbs that the transaction program is allowed to issue. Either NONE or any combination of CNOS, SESSION_CONTROL, DEFINE, DISPLAY, and ALLOCATE_SERVICE_TP may be specified.

- **NONE** specifies that the program is not allowed to issue verbs that require a privilege to do so.
- **CNOS** specifies that the program is allowed to issue the CNOS verbs.
- **SESSION_CONTROL** specifies that the program is allowed to issue the ACTIVATE_SESSION and DEACTIVATE_SESSION verbs.
- **DEFINE** specifies that the program is allowed to issue the DEFINE verbs and the DELETE verbs.
- **DISPLAY** specifies that the program is allowed to issue the DISPLAY verbs.
- **ALLOCATE_SERVICE_TP** specifies that the program is allowed to issue the ALLOCATE verb with its TPN parameter specifying an SNA service transaction program.

Returned Parameters:

RETURN_CODE returns an indication of the result of verb execution.

- OK
- PARAMETER_ERROR (for one of the following reasons)
  - CONVERSATION_TYPE(BASIC) and either DATA_MAPPING(YES) or FMH_DATA(YES) are specified.
  - SECURITY_ACCESS is specified, SECURITY_REQUIRED is omitted, and no type of security verification is currently defined.

- SECURITY_ACCESS is specified and the type of security verification specified on SECURITY_REQUIRED or currently defined does not include resource-access verification.
- SECURITY_ACCESS specifies a user ID or profile and the type of security verification specified on SECURITY_REQUIRED or currently defined does not include the respective verification of user IDs or profiles.
- SECURITY_ACCESS specifies only a profile and the type of security verification specified on SECURITY_REQUIRED or currently defined includes verification of user IDs.
- SECURITY_ACCESS(ADD(...)) specifies a user ID or profile that is not a symbol-string type (A, AE, GR, or DB) that the product supports.
- SECURITY_ACCESS(DELETE(...)) specifies a user ID or profile that is not currently defined at the local LU.

ABEND Conditions:

Parameter Check

- This verb is not supported.
- The program issuing this verb does not have define privilege.

Notes:

1. The values specified on the parameters of this verb take effect at the next invocation of the transaction program.

2. If the status for starting execution of the transaction program is not defined, the program's status is ENABLED.

3. If the conversation type allowed on allocation requests that start the transaction program is not defined, a mapped or basic conversation is allowed.

4. If the synchronization level allowed on allocation requests that start the transaction program is not defined, a synchronization level of NONE or CONFIRM is allowed.

5. Resource-access verification is used to verify the access security information on incoming allocation requests for the authority to access the transaction program named on the requests and the local resources that the program allocates. The local LU maintains a resource-access authorization list for this purpose. The authorization list is created and updated from information supplied on the SECURITY_ACCESS parameter. The list may consist of profiles alone, or it may consist of user IDs alone or with associated profiles, as determined by the SECURITY_REQUIRED parameter.

6. There is a resource-access authorization list for each transaction program for which resource-access verification is defined. When the first user ID and associated password and profiles are added, or the first profile is added, a resource-access authorization list is created for the program. When the last user ID and associated password and profiles are deleted, or the last profile is deleted, the resource-access authorization list itself is deleted.

7. If resource-access verification is to be performed and it includes verification of user IDs, the authorization list must contain one or more user IDs when the verification of allocation requests takes place. Similarly, if the verification includes profiles, the list must contain one or more profiles.

8. If the type of security verification required on incoming allocation requests is currently defined and a different type is specified, the new type replaces the current type. If resource-access verification is currently defined as being required and a different type of resource-access verification is specified, the resource-access authorization list is deleted and a new list is created.

9. If SECURITY_ACCESS is specified without SECURITY_REQUIRED, the type of security verification currently defined applies to the use of the SECURITY_ACCESS parameter.

10. If no type of security verification is currently defined, none is required to start the program. However, if the allocation request carries access security information, the local LU performs conversation-level security verification.

11. If no PIP subfield number is defined at the local LU for the transaction program, only allocation requests carrying no PIP data are allowed to start the program.

12. If no data mapping support is defined at the local LU for the transaction program, none is provided.

13. If no FMH data support is defined at the local LU for the transaction program, none is provided.

14. If no privilege is defined at the local LU for the transaction program, the program may issue only conversation verbs.

15. More details concerning the LU's use of these operating parameters are given in SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2.

DISPLAY_LOCAL_LU

Returns current values of parameters that control the operation of the local LU.

DISPLAY_LOCAL_LU	Supplied Parameters: FULLY_QUALIFIED_LU_NAME (variable)
	Returned Parameters: RETURN_CODE (variable) [LU_SESSION_LIMIT (variable)] [LU_SESSION_COUNT (variable)] [SECURITY (variable)] [MAP_NAMES (variable)] [REMOTE_LU_NAMES (variable)] [TP_NAMES (variable)]
	;

Supplied Parameters:

FULLY_QUALIFIED_LU_NAME specifies the fully qualified name of the local LU.

Returned Parameters:

RETURN_CODE returns an indication of the result of verb execution.

- OK
- PARAMETER_ERROR (for the following reason)
  - FULLY_QUALIFIED_LU_NAME does not specify a local LU name currently defined at the local LU.

LU_SESSION_LIMIT returns the LU-LU session limit currently defined at the local LU.

LU_SESSION_COUNT returns the LU-LU session count, which is the total number of active sessions for the local LU.

SECURITY returns the conversation-level security verification list currently defined at the local LU.

MAP_NAMES returns a list of the local map names currently defined at the local LU.

REMOTE_LU_NAMES returns a list of the remote LU names currently defined at the local LU.

TP_NAMES returns a list of the local transaction program names currently defined at the local LU.

ABEND Conditions:

Parameter Check

- This verb is not supported.
- The program issuing this verb does not have display privilege.

Notes:

1. This verb can be used to obtain operating parameter values that are established by the DEFINE_LOCAL_LU, DEFINE_REMOTE_LU, and DEFINE_TP verbs, as well as the current LU-LU session count.

2. More details concerning the LU's use of these operating parameters are given in SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2.

DISPLAY_REMOTE_LU

Returns current values of parameters that control the operation of the local LU in conjunction with a remote LU.

DISPLAY_REMOTE_LU	Supplied Parameters: FULLY_QUALIFIED_LU_NAME (variable)
	Returned Parameters: RETURN_CODE (variable) [LOCALLY_KNOWN_LU_NAME (variable)] [UNINTERPRETED_LU_NAME (variable)] [INITIATE_TYPE (variable)] [PARALLEL_SESSION_SUPPORT (variable)] [CNOS_SUPPORT (variable)] [SECURITY_ACCEPTANCE_LOCAL_LU (variable)] [SECURITY_ACCEPTANCE_REMOTE_LU (variable)] [MODE_NAMES (variable)]
	;

Supplied Parameters:

FULLY_QUALIFIED_LU_NAME specifies the fully qualified name of the remote LU.

Returned Parameters:

RETURN_CODE returns an indication of the result of verb execution.

- OK
- PARAMETER_ERROR (for the following reason)
  - FULLY_QUALIFIED_LU_NAME does not specify a remote LU name currently defined at the local LU.

LOCALLY_KNOWN_LU_NAME returns the locally-known name of the remote LU, currently defined at the local LU.

UNINTERPRETED_LU_NAME returns the uninterpreted name of the remote LU, currently defined at the local LU.

INITIATE_TYPE returns an indication of the session-initiation type for the remote LU, currently defined at the local LU.

PARALLEL_SESSION_SUPPORT returns an indication of the parallel session support for sessions with the remote LU. If one or more sessions are active between the local and remote LUs, this parameter returns an indication of the actual parallel session support; otherwise, it returns an indication of the support currently defined at the local LU.

CNOS_SUPPORT returns an indication of the CNOS support for sessions with the remote LU. If one or more sessions are active between the

local and remote LUs, this parameter returns an indication of the actual CNOS support; otherwise, it returns an indication of the support currently defined at the local LU.

SECURITY_ACCEPTANCE_LOCAL_LU returns an indication of the level of access security information that the local LU will accept on allocation requests it receives from the remote LU, currently defined at the local LU.

SECURITY_ACCEPTANCE_REMOTE_LU returns an indication of the level of access security information that the remote LU will accept on allocation requests it receives from the local LU, when one or more sessions are active between the local and remote LUs. The value returned is what is currently defined at the remote LU, which is conveyed to the local LU during session activation.

MODE_NAMES returns a list of the mode names currently defined at the local LU for sessions with the remote LU.

ABEND Conditions:

Parameter Check

- This verb is not supported.
- The program issuing this verb does not have display privilege.

Notes:

1. This verb is used to obtain operating parameter values that are defined by the DEFINE_REMOTE_LU and DEFINE_MODE verbs.

2. More details concerning the LU's use of these operating parameters are given in SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2.

DISPLAY_MODE

Returns current values of parameters that control the operation of the local LU in conjunction with a group of sessions to a remote LU, the session group being identified by a mode name.

DISPLAY_MODE	Supplied Parameters: FULLY_QUALIFIED_LU_NAME (variable) MODE_NAME (variable)
	Returned Parameters: RETURN_CODE (variable) [SEND_PACING_WINDOW (variable)] [RECEIVE_PACING_WINDOW (variable)] [SEND_MAX_RU_SIZE_LOWER_BOUND (variable)] [SEND_MAX_RU_SIZE_UPPER_BOUND (variable)] [RECEIVE_MAX_RU_SIZE_LOWER_BOUND (variable)] [RECEIVE_MAX_RU_SIZE_UPPER_BOUND (variable)] [SYNC_LEVEL_SUPPORT (variable)] [SINGLE_SESSION_REINITIATION (variable)] [SESSION_LEVEL_CRYPTOGRAPHY (variable)] [CONWINNER_AUTO_ACTIVATE_LIMIT (variable)] [LU_MODE_SESSION_LIMIT (variable)] [MIN_CONWINNERS (variable)] [MIN_CONLOSERS (variable)] [TERMINATION_COUNT (variable)] [DRAIN_LOCAL_LU (variable)] [DRAIN_REMOTE_LU (variable)] [LU_MODE_SESSION_COUNT (variable)] [CONWINNERS_SESSION_COUNT (variable)]

	(continued from preceding page) [CONLOSERS_SESSION_COUNT (variable)] [SESSION_IDS (variable)]
	;

Supplied Parameters:

FULLY_QUALIFIED_LU_NAME specifies the fully qualified name of the remote LU.

MODE_NAME specifies the mode name.

Returned Parameters:

RETURN_CODE returns an indication of the result of verb execution.

- OK
- PARAMETER_ERROR (for one of the following reasons)
  - FULLY_QUALIFIED_LU_NAME does not specify a remote LU name currently defined at the local LU.
  - MODE_NAME does not specify a mode name currently defined at the local LU.

SEND_PACING_WINDOW returns the local LU's send pacing window size for the sessions, currently defined at the local LU.

RECEIVE_PACING_WINDOW returns the local LU's receive pacing window size for the sessions, currently defined at the local LU.

SEND_MAX_RU_SIZE_LOWER_BOUND returns the lower bound for the maximum size of normal-flow requests that the local LU sends on the sessions, currently defined at the local LU.

SEND_MAX_RU_SIZE_UPPER_BOUND returns the upper bound for the maximum size of normal-flow requests that the local LU sends on the sessions, currently defined at the local LU.

RECEIVE_MAX_RU_SIZE_LOWER_BOUND returns the lower bound for the maximum size of normal-flow requests that the local LU receives on the sessions, currently defined at the local LU.

RECEIVE_MAX_RU_SIZE_UPPER_BOUND returns the upper bound for the maximum size of normal-flow requests that the local LU receives on the sessions, currently defined at the local LU.

SYNC_LEVEL returns an indication of the synchronization levels that are supported for conversations allocated to the sessions. If one or more sessions within the mode name group are active between the local and remote LUs, this parameter returns an indication of the actual synchronization level support; otherwise, it returns an indication of the support currently defined at the local LU.

SINGLE_SESSION_REINITIATION returns an indication of the session reinitiation responsibility for a single session with the remote LU. If a session is active between the local and remote LUs, this parameter returns an indication of the actual session reinitiation responsibility; otherwise, it returns an indication of the responsibility currently defined at the local LU.

SESSION_LEVEL_CRYPTOGRAPHY returns an indication of the session-level cryptography support for the sessions. If one or more sessions within the mode name group are active between the local and remote LUs, this parameter returns an indication of the actual session-level cryptography support; otherwise, it returns an indication of the support currently defined at the local LU.

CONWINNER_AUTO_ACTIVATE_LIMIT returns the local LU's automatic-activation limit on the number of contention winner sessions, currently defined at the local LU.

LU_MODE_SESSION_LIMIT returns the current (LU,mode) session limit.

MIN_CONWINNERS returns the current minimum number of sessions for which the local LU is designated to be the contention winner.

MIN_CONLOSERS returns the current minimum number of sessions for which the remote LU is designated to be the contention winner, making the local LU the contention loser.

TERMINATION_COUNT returns the termination count, which is the number of sessions for which that the local LU is responsible to deactivate as a result of CNOS processing.

DRAIN_LOCAL_LU returns an indication of whether the local LU is allowed to drain its allocation requests as a result of CNOS processing that resets the (LU,mode) session limit.

DRAIN_REMOTE_LU returns an indication of whether the remote LU is allowed to drain its allocation requests as a result of CNOS processing that resets the (LU,mode) session limit.

LU_MODE_SESSION_COUNT returns the current (LU,mode) session count.

CONWINNERS_SESSION_COUNT returns the number of active sessions for which the local LU is the contention winner.

CONLOSERS_SESSION_COUNT returns the number of active sessions for which the local LU is the contention loser.

SESSION_IDS returns a list of the session identifiers assigned to the active sessions.

ABEND Conditions:

Parameter Check

- This verb is not supported.
- The program issuing this verb does not have display privilege.

Notes:

1. This verb can be used to obtain operating parameter values that are established by the DEFINE_MODE verb and the CNOS verbs.

2. More details concerning the LU's use of these operating parameters are given in SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2.

DISPLAY_TP

Returns current values of parameters that control the operation of the local LU in conjunction with a transaction program.

DISPLAY_TP	**Supplied Parameters:** TP_NAME (variable)
	Returned Parameters: [RETURN_CODE (variable)] [STATUS (variable)] [CONVERSATION_TYPE (variable)] [SYNC_LEVEL (variable)] [SECURITY_REQUIRED (variable)] [SECURITY_ACCESS (variable)] [PIP (variable)] [DATA_MAPPING (variable)] [FMH_DATA (variable)] [PRIVILEGE (variable)]
	;

Supplied Parameters:

TP_NAME specifies the local transaction program name.

Returned Parameters:

RETURN_CODE returns an indication of the result of verb execution.

- OK
- PARAMETER_ERROR (for the following reason)
  - TP_NAME does not specify a transaction program name that is currently defined at the local LU.

STATUS returns an indication of the status for starting execution of the transaction program, as currently defined at the local LU.

CONVERSATION_TYPE returns an indication of the conversation type required on allocation requests that start the transaction program, as currently defined at the local LU.

SYNC_LEVEL returns an indication of the synchronization level required on allocation requests that start the transaction program, as currently defined at the local LU.

SECURITY_REQUIRED returns an indication of the type of security verification that is required to be performed on incoming allocation requests designating the transaction program.

SECURITY_ACCESS returns the resource-access authorization list currently defined for the transaction program at the local LU.

PIP returns the number of PIP subfields required on allocation requests that start the transaction program, as currently defined at the local LU.

DATA_MAPPING returns an indication of whether data mapping support is provided to the transaction program, as currently defined at the local LU.

FMH_DATA returns an indication of whether FMH data support is provided to the transaction program, as currently defined at the local LU.

PRIVILEGE returns an indication of the class of privileged verbs that the transaction program is allowed to issue, as currently defined at the local LU.

ABEND Conditions:

Parameter Check

- This verb is not supported.
- The program issuing this verb does not have display privilege.

Notes:

1. This verb can be used to obtain operating parameter values that are established by the DEFINE_TP verb.

2. More details concerning the LU's use of these operating parameters are given in SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2.

DELETE

Deletes parameter values, established by means of the DEFINE verbs, that control the operation of the local LU. The execution of this verb involves only the local LU; it does not cause any information to be sent outside the LU.

DELETE	**Supplied Parameters:** [LOCAL_LU_NAME (variable)] [REMOTE_LU_NAME (variable)] [MODE_NAME (variable)] [TP_NAME (variable)]
	Returned Parameters: RETURN_CODE (variable)
	;

Supplied Parameters:

LOCAL_LU_NAME specifies the fully qualified name of the local LU. This parameter must be specified alone. Specifying this parameter deletes the local LU name and all parameter values associated with the local LU; that is, it deletes all parameter values that have been defined by means of the DEFINE_LOCAL_LU, DEFINE_REMOTE_LU, DEFINE_MODE, and DEFINE_TP verbs.

REMOTE_LU_NAME specifies the fully qualified name of the remote LU. This parameter may be specified together with the MODE_NAME parameter. Specifying this parameter without the MODE_NAME parameter deletes the remote LU name and all parameter values associated with the remote LU; that is, it deletes all parameter values that have been defined by means of the DEFINE_REMOTE_LU and DEFINE_MODE verbs. Specifying this parameter together with the MODE_NAME parameter deletes parameter values associated with the mode name, but the remote LU name and all parameter values not associated with the mode name remain unchanged.

MODE_NAME specifies the mode name. This parameter must be specified together with the REMOTE_LU_NAME parameter. Specifying this parameter deletes all parameter values associated with the mode name for the remote LU; that is, it deletes the mode name and all parameter values that have been defined by means of the DEFINE_MODE_NAME verb.

TP_NAME specifies the local transaction program name. Specifying this parameter deletes all parameter values associated with the transaction program; that is, it deletes the program name and all parameter values that have been defined by means of the DEFINE_TP verb.

Returned Parameters:

RETURN_CODE returns an indication of the result of verb execution.

- OK
- PARAMETER_ERROR (for one of the following reasons)
  - LOCAL_LU_NAME specifies a local LU name not currently defined at the local LU.
  - LOCAL_LU_NAME is not specified alone.
  - REMOTE_LU_NAME specifies a remote LU name not currently defined at the local LU.

- MODE_NAME specifies a mode name not currently defined at the local LU.
- MODE_NAME is specified without REMOTE_LU_NAME.
- TP_NAME specifies a local transaction program name not currently defined at the local LU.

ABEND Conditions:

Parameter Check

- This verb is not supported.
- The program issuing this verb does not have define privilege.

Notes:

1. Deleting parameter values makes those values undefined to the local LU.

2. When deleting a local LU name and all its associated parameter values, verb should be issued only when the local LU is not participating in any network activity.

3. When deleting a remote LU name and all its associated parameter values, verb should be issued only when the local LU is not participating in any network activity involving the remote LU.

4. When deleting a mode name and all its associated parameter values, verb should be issued only when the local LU is not participating in any network activity involving the remote LU and mode name.

5. When deleting a transaction program name and all its associated parameter values, verb should be issued only when the transaction program is not in use.

6. More details concerning the LU's use of the LU-LU session limit are given in SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2.

RETURN CODES

Some verbs have a parameter called RETURN_CODE used to pass a return code back to the transaction program at the completion of the LU's execution of a verb. The return code indicates the result of processing the verb on which it is returned. Only one code is returned at a time. Other verb-specific information may be passed back in verb-unique parameters. See each specific verb for a description of any verb-unique parameters.

The return codes are described below. Each description includes the meaning of the return code and the origin of the condition indicated by the return code.

ACTIVATION_FAILURE_NO_RETRY indicates the ACTIVATE_SESSION verb failed to activate the session because of a condition that is not temporary. For example, the session cannot be activated because the (LU,mode) session limit for the specified target LU and mode name is currently 0 at the target LU—this applies to single sessions and to sessions for the SNASVCMG mode name; or because of a system definition error or a session-activation protocol error. The control operator should not retry the transaction until the condition is corrected.

ACTIVATION_FAILURE_RETRY indicates the ACTIVATE_SESSION verb failed to activate the session because of a temporary condition. For example, the session cannot be activated because of a temporary lack of resources at the source LU or target LU. The control operator may retry the session activation later.

ALLOCATION_ERROR indicates the CNOS verb did not execute successfully because the allocation of the control operator conversation with the target LU cannot be completed. The ALLOCATION_ERROR indication together with one of the following subcodes form the complete return code that is returned to the transaction program; the subcode identifies the specific error. The source and target LUs' CNOS parameters are not changed.

- **ALLOCATION_FAILURE_NO_RETRY** indicates the control operator conversation cannot be allocated because of a condition that is not temporary. For example, the session to be used for the control operator conversation cannot be activated because the (LU,mode) session limit for the specified target LU and SNASVCMG mode name is currently 0 at either the source LU or target LU; or because of a system definition error or a session-activation protocol error; or because a session protocol error caused the session to be deactivated before the conversation could be allocated. The control operator should not retry the transaction until the condition is corrected.

- **ALLOCATION_FAILURE_RETRY** indicates the control operator conversation cannot be allocated because of a temporary condition. For example, the session to be used for the control operator conversation cannot be activated because of a temporary lack of resources at the source LU or target LU; or the session was deactivated because of session outage before the conversation could be allocated. The condition is temporary, and the control operator can retry the transaction later.

- **TRANS_PGM_NOT_AVAIL_RETRY** indicates the target LU is currently unable to start the transaction program identified as hex 06F1, which is the SNA service transaction program for the control operator. For example, there may be a temporary lack of resources the target LU needs to start the transaction program. The condition is temporary, and the control operator can retry the transaction later.

COMMAND_RACE_REJECT indicates the CNOS verb did not execute successfully because the source LU or target LU is currently processing another CNOS transaction for the same mode name. The other CNOS transaction is processed to completion. The source and target LUs' CNOS parameters are not changed by the unsuccessful CNOS verb.

LU_MODE_SESSION_LIMIT_CLOSED indicates the CNOS verb did not execute successfully because the target LU currently will not allow the (LU,mode) session limit for the specified mode name to be raised above 0. The (LU,mode) session limit remains at 0. This condition is not necessarily permanent; the control operator may retry the CNOS transaction later.

LU_MODE_SESSION_LIMIT_EXCEEDED indicates the ACTIVATE_SESSION verb could not activate the session with the specified mode name to the target LU, for one of the following reasons:

1. For a single session connection to the target LU, either the (LU,mode) session limit is currently 0, or an LU-LU session is already active (with the specified or a different mode name).

2. For a parallel session connection to the target LU, the number of currently active sessions with the specified mode name equals the (LU,mode) session limit.

LU_MODE_SESSION_LIMIT_NOT_ZERO indicates the program attempted to initialize an (LU,mode) session limit that is already initialized, that is, the session limit is already greater than 0. The source and target LUs' CNOS parameters are not changed.

LU_MODE_SESSION_LIMIT_ZERO indicates the program attempted to change an (LU,mode) session limit that has not been initialized, that is, the session limit is 0. The source and target LUs' CNOS parameters are not changed.

LU_SESSION_LIMIT_EXCEEDED indicates the CNOS verb did not execute successfully because the new (LU,mode) session limit would cause the sum of the (LU,mode) session limits to exceed the total LU-LU session limit for the source LU (see the DEFINE_LOCAL_LU verb). The sum of the (LU,mode) session limits is calculated as follows:

1. A single session connection to a target LU is counted as 1 if at least one of the (LU,mode) session limits for that target LU is 1, including the specified session limit. Otherwise, it is counted as 0.

2. A parallel session connection to a target LU is counted as the sum of all (LU,mode) session limits for the target LU, including the specified session limit.

OK indicates the verb executed successfully. The following subcodes augment this return code and indicate whether the parameter values were processed as specified or as negotiated by the target LU:

- **AS_SPECIFIED** indicates the two LUs executed the verb as specified, without negotiation.

- **AS_NEGOTIATED** indicates the two LUs executed the verb as negotiated by the target LU. One or more parameter values have been negotiated. The transaction program can obtain the negotiated parameter values by issuing the DISPLAY_MODE verb. The verb descriptions define which parameter values can be negotiated.

- **FORCED** indicates the source LU forced the resetting of its (LU,mode) session limit as a result of an error condition that prevented successful completion of the exchange of the CNOS request and reply. The target LU's CNOS parameters may not be changed, depending on the error condition and when it occurred during the CNOS exchange.

PARAMETER_ERROR indicates the verb did not execute successfully because it specifies a parameter that contains an invalid argument. The source of the argument may be outside the transaction program definition, such as a control-operator supplied LU name or mode name. When this return code is returned on a CNOS verb, the source and target LUs' CNOS parameters are not changed. When it is returned on a session activation or deactivation verb, the LU-LU session is not activated or deactivated, respectively.

When it is returned on a define, display, or delete verb, the LU operating parameters are not altered or returned.

REQUEST_EXCEEDS_MAX_ALLOWED indicates the CNOS verb did not execute successfully because it specifies an (LU,mode) session limit that exceeds the source LU's maximum (LU,mode) session limit defined for the target LU and mode name (see the DEFINE_MODE verb). The source and target LUs' CNOS parameters are not changed.

RESOURCE_FAILURE_NO_RETRY indicates the CNOS verb did not execute successfully because of a failure that caused the control operator conversation to be prematurely deallocated. For example, the session being used for the control operator conversation was deactivated because of a session protocol error, or because of session outage from which the control operator component of the LU could not recover; or the conversation was deallocated because of a protocol error between the control operator components of the LUs. The condition is not temporary, and the control operator should not retry the transaction until the condition is corrected. The CNOS parameters remain unchanged at the source LU, or both the source and target LUs, depending on when the failure occurred.

UNRECOGNIZED_MODE_NAME indicates the CNOS verb did not execute successfully because the target LU does not recognize the specified mode name. The source and target LUs' CNOS parameters are not changed.

Figure 5-1 on page 5-54 shows the correlation of the return codes to the verbs on which they can be returned. The "X" in the figure means the return code can be returned on the corresponding verb. A verb without any "X"s under it means no return codes are defined for the verb. The individual verb descriptions list the applicable return codes. However, the subcodes of ALLOCATION_ERROR are not explicitly listed, as any of them can be returned as part of the ALLOCATION_ERROR return code.

Verbs	Return Codes													
	ACTIVATION FAILURE NO RETRY	ACTIVATION FAILURE RETRY	ALLOCATION ERROR	COMMAND RACE REJECT	LU MODE SESSION LIMIT CLOSED	LU MODE SESSION LIMIT EXCEEDED	LU MODE SESSION LIMIT NOT ZERO	LU MODE SESSION LIMIT ZERO	LU SESSION LIMIT EXCEEDED	OK	PARAMETER ERROR	REQUEST EXCEEDS MAX ALLOWED	RESOURCE FAILURE NO RETRY	UNRECOGNIZED MODE NAME
CHANGE_SESSION_LIMIT			X	X				X	X	X	X	X	X	
INITIALIZE_SESSION_LIMIT			X	X	X		X		X	X	X	X	X	X
PROCESS_SESSION_LIMIT										X			X	
RESET_SESSION_LIMIT			X	X	X					X	X		X	X
ACTIVATE_SESSION	X	X				X				X	X			
DEACTIVATE_SESSION										X	X			
DEFINE_LOCAL_LU										X	X			
DEFINE_REMOTE_LU										X	X			
DEFINE_MODE										X	X			
DEFINE_TP										X	X			
DISPLAY_LOCAL_LU										X	X			
DISPLAY_REMOTE_LU										X	X			
DISPLAY_MODE										X	X			
DISPLAY_TP										X	X			
DELETE										X	X			

Figure 5-1. Correlation of Return Codes to Verbs

APPENDIX A. BASE AND OPTION SETS FOR PRODUCT SUPPORT

The LU 6.2 functions available to a transaction program are described in this book by means of verbs and their supplied and returned parameters. The returned parameters include the return codes of the RETURN_CODE parameter, defined for most verbs, and the what-received indications of the WHAT_RECEIVED parameter, defined for the MC_RECEIVE_AND_WAIT, MC_RECEIVE_IMMEDIATE, RECEIVE_AND_WAIT, and RECEIVE_IMMEDIATE verbs.

An LU 6.2 product may provide support for all of the verbs, parameters, return codes, and what-received indications, or a permitted subset of them. The permitted subsetting for LU 6.2 products is defined by means of a base set and a number of option sets (see "Product-Support Subsetting" in Chapter 3 for a discussion of base and option sets). The option sets defined for the verbs, parameters, return codes, and what-received indications are:[1]

1. **Conversations between programs located at the same LU:** This option set allows a local program to allocate a conversation to a remote program located at the same LU as the local program.

2. **Delayed allocation of a session:** This option set allows a program to delay allocation of a session until the LU must flush its send buffer.

3. **Immediate allocation of a session:** This option set allows a program to allocate a contention-winner session only if one is immediately available; otherwise, the allocation is unsuccessful.

4. **Sync point services:** This option set allows a program to request sync point processing of all protected resources throughout the scope of the transaction. This option set includes the SYNCPT and BACKOUT verbs.

5. **Session-level LU-LU verification:** This option set allows a program or operator to designate the LU-LU passwords, associated with remote LUs, that the local LU uses to verify the identity of a remote LU at session activation time.

6. **User ID verification:** This option set allows a program or operator to designate the user IDs and associated passwords that the local LU uses to verify the identity of a user ID carried on allocation requests it receives, and to designate the remote LUs that are permitted to send to the local LU allocation requests carrying a user ID and either a password or an already-verified indication. This option set also allows the program allocating a conversation to specify that the allocation request carry the user ID received on the request that started the program, together with an already-verified indication. Option set 5 is a prerequisite.

7. **Program supplied user ID and password:** This option set allows the program allocating a conversation to supply the user ID and password to be sent on the allocation request. Option set 5 is a prerequisite.

8. **User ID authorization:** This option set allows a program or operator to designate the user IDs that are authorized access to specific resources of the LU, such as transaction programs. Option set 6 is a prerequisite.

9. **Profile verification and authorization:** This option set allows a program or operator to designate the profiles that the local LU uses to verify a profile carried on allocation requests it

---

[1] The numbers associated with these option sets are used only for descriptive purposes; they have no architectural significance, and may change from one edition of this book to the next.

receives, and to designate the profiles that are authorized access to specific resources of the LU, such as transaction programs. Option set 6 is a prerequisite.

10. **Profile passthrough:** This option set allows the program allocating a conversation to specify that the allocation request carry the profile received on the request that started the program. Option set 6 is a prerequisite.

11. **Program supplied profile:** This option set allows the program allocating a conversation to supply the profile to be sent on the allocation request. Option set 7 is a prerequisite.

12. **PIP data:** This option set allows the program allocating a conversation to provide the remote program with initialization parameters.

13. **Logging of data in a system log:** This option set allows a program to record error information in the system's error log.

14. **Flush the LU's send buffer:** This option set allows a program to explicitly cause the LU to flush its send buffer.

15. **LUW identifier:** This option set allows an LU implementation to use the LUW identifier for accounting purposes.

16. **Prepare to receive:** This option set allows a program to change the conversation from send state to receive state and at the same time flush the LU's send buffer, request confirmation, or request sync point.

17. **Long locks:** This option set allows a program to perform the prepare-to-receive function and request confirmation, and resume processing when information, such as data or conversation status, is received from the remote program following an affirmative reply. Option set 16 is a prerequisite.

18. **Post on receipt with wait:** This option set allows a program to request posting of multiple conversations and then to wait (suspend its processing) until information is available on any one of the conversations. Option set 16 is a prerequisite.

19. **Post on receipt with test for posting:** This option set allows a program to request posting of a conversation and then to test the conversation to determine whether information is available. Option set 16 is a prerequisite.

20. **Receive immediate:** This option set allows a program to receive whatever information is available on a conversation without having to request posting of the conversation. Option set 16 is a prerequisite.

21. **Test for request-to-send received:** This option set allows a program to test whether a request-to-send notification has been received on a conversation, for example following sync point processing.

22. **Data mapping:** This option set allows a program to request mapping of the data by the local and remote LUs.

23. **FMH data:** This option set allows programs to send and receive data records containing FM header data. The FM header data has meaning only to the application programs.

24. **Get attributes:** This option set allows a program to obtain attributes of a mapped conversation.

25. **Get conversation type:** This option set allows a program that supports both the basic conversation and mapped conversation protocol boundaries to determine which category of verbs it should use in conjunction with a resource ID.

26. **Mapped Conversation LU Services Component:** This option set allows implementation of a mapped conversation LU services component program, which processes mapped conversation verbs.

27. **CHANGE_SESSION_LIMIT verb:** This option set allows a program or operator at the source LU to request a change in the (LU,mode) session limit from one nonzero value to another, or a change in the minimum number of contention-winner sessions for the source LU or target LU.

28. **MIN_CONWINNERS_TARGET parameter:** This option set allows a program or operator at the source LU to request a nonzero value for the target LU's minimum number of contention-winner sessions. Option set 27 is a prerequisite for this parameter on CHANGE_SESSION_LIMIT.

29. **RESPONSIBLE(TARGET) parameter:** This option set allows a program or operator at the source LU to request that the target LU be responsible for session deactivations when the verb requires a decrease in the number of active sessions. Option set 27 is a prerequisite for this parameter on CHANGE_SESSION_LIMIT.

30. **DRAIN_TARGET(NO) parameter:** This option set allows a program or operator at the source LU to prevent the target LU from draining its allocation requests as a result of resetting the (LU,mode) session limit to 0.

31. **FORCE parameter:** This option set allows a program or operator to specify that the (LU,mode) session limit is to be reset to 0 even if the CNOS exchange between the source LU and target LU is unsuccessful.

32. **ACTIVATE_SESSION verb:** This option set allows a program or operator to activate LU-LU sessions.

33. **DEACTIVATE_SESSION verb:** This option set allows a program or operator to deactivate LU-LU sessions.

34. **LU-parameter verbs:** This option set allows a program or operator to specify the operating parameters of its LU. Within this option set, the individual operating parameters that a product supports and makes accessible to the program or operator are product-dependent.

35. **LU-LU session limit:** This option set allows a program or operator to specify the LU-LU session limit.

36. **Locally-known LU names:** This option set allows a program or operator to specify the locally-known names of remote LUs.

37. **Uninterpreted LU names:** This option set allows a program or operator to specify the uninterpreted names of remote LUs.

38. **Single-session reinitiation:** This option set allows a program or operator to specify the responsibility for reinitiation of single sessions to remote LUs.

39. **Maximum RU size bounds:** This option set allows a program or operator to specify the lower and upper bounds for the maximum RU sizes on sessions within an (LU,mode) group.

40. **Session-level mandatory cryptography:** This option set allows a program or operator to specify that session-level mandatory cryptography is to be used on sessions within an (LU,mode) group.

41. **Contention winner automatic activation limit:** This option set allows a program or operator to specify the limit for automatically activating contention-winner sessions within an (LU,mode) group.

The following figures identify local and remote support of the base set and option sets for the verbs, parameters, return codes, and what-received indications. Local support is defined for all of these. Remote support is defined only for the verbs and parameters, as it

does not apply to the return codes and what-received indications. Two hyphens (--) are shown for remote support of a verb or parameter that does not invoke remote processing.

The verbs, parameters, return codes, and what-received indications belonging to the base set are identified by "B" in the local-support or remote-support column. Those belonging to an option set are identified by the number of that option set.

For some of the verbs, parameters, return codes, and what-received indications, more than one option set is identified. An identification of the form "a or b" means the verb, parameter, return code, or what-received indication is supported when either option set "a" or "b" is supported. An identification of the form "a and b" means the verb, parameter, return code, or what-received indication is supported when both option sets "a" and "b" are supported.

Notes pertaining to the base and optional support of the verbs, parameters, return codes, and what-received indications are listed following the figures, beginning on page A-20. The verbs, parameters, return codes, and what-received indications to which the notes apply include a note reference, shown as "[n]," in the local- or remote-support column. The notes explain certain implementation details, which are product dependent.

Note: As shown in the figures for the conversation verbs and parameters, most of the option sets are optional only for local support; remote support for the verbs and parameters of these option sets is either part of the base set (indicated with "B") or is not applicable (indicated with "--"). The local program may use these conversation verbs and parameters whenever its product supports them. Use of the remaining conversation verbs and parameters—those for which remote support of an option set is shown—depends on the remote support that the remote product provides. In particular, the local program may use the verbs and parameters of the following option sets whenever its product supports them and the remote product provides the support indicated in the remote-support column:

4. Sync point services
6. End-user verification
7. Program supplied user ID and password
10. Profile passthrough
11. Program supplied profile
12. PIP data
22. Data mapping
23. FMH data

SUPPORT FOR MAPPED CONVERSATION VERBS AND PARAMETERS

Verb and Parameter	Local Support	Remote Support
MC_ALLOCATE	B	B [1]
LU_NAME(OWN)	1	--
LU_NAME(OTHER(variable))	B	B
MODE_NAME	B	B
TPN	B	B
RETURN_CONTROL(WHEN_SESSION_ALLOCATED)	B [2]	--
RETURN_CONTROL(DELAYED_ALLOCATION_PERMITTED)	2	--
RETURN_CONTROL(IMMEDIATE)	3	--
SYNC_LEVEL(NONE)	B	B
SYNC_LEVEL(CONFIRM)	B	B
SYNC_LEVEL(SYNCPT)	4	4
SECURITY(NONE)	B	B
SECURITY(SAME)	6 or 10 [3]	6, 8, 9 or 10
SECURITY(PGM(USER_ID(variable)	7	6 or 8
PASSWORD(variable)	7	6
PROFILE(variable)))	11	9 or 10
PIP(NO)	B	B
PIP(YES(variable))	12 [4]	12 [4]
RESOURCE	B	--
RETURN_CODE	B	--
MC_CONFIRM	B	B
RESOURCE	B	--
RETURN_CODE	B	--
REQUEST_TO_SEND_RECEIVED	B	--
MC_CONFIRMED	B	B
RESOURCE	B	--
MC_DEALLOCATE	B	B
RESOURCE	B	--
TYPE(SYNC_LEVEL)	B	B
TYPE(FLUSH)	B	B
TYPE(CONFIRM)	B	B
TYPE(ABEND)	B	B
TYPE(LOCAL)	B	--
RETURN_CODE	B	--
MC_FLUSH	14	B
RESOURCE	14	--

Figure A-1. Support for Mapped Conversation Verbs and Parameters (Part 1 of 3)

Verb and Parameter	Local Support	Remote Support
MC_GET_ATTRIBUTES	4, 6, 9, 15, or 24	--
RESOURCE	4, 6, 9, 15, or 24	--
OWN_FULLY_QUALIFIED_LU_NAME	24	--
PARTNER_LU_NAME	24	--
PARTNER_FULLY_QUALIFIED_LU_NAME	24	--
MODE_NAME	24	--
SYNC_LEVEL	4 or 24	--
SECURITY_USER_ID	6	--
SECURITY_PROFILE	9	--
LUW_IDENTIFIER	4 or 15	--
CONVERSATION_CORRELATOR	4	--
MC_POST_ON_RECEIPT	18 or 19 [5]	--
RESOURCE	18 or 19	--
LENGTH	18 or 19 [10]	--
MC_PREPARE_TO_RECEIVE	16	B
RESOURCE	16	--
TYPE(SYNC_LEVEL)	16	B
TYPE(FLUSH)	16	B
TYPE(CONFIRM)	16	B
LOCKS(SHORT)	16	B
LOCKS(LONG)	17	B
RETURN_CODE	16	--
MC_RECEIVE_AND_WAIT	B [6]	--
RESOURCE	B	--
LENGTH	B [10]	--
RETURN_CODE	B	--
REQUEST_TO_SEND_RECEIVED	B [9]	--
DATA	B	--
WHAT_RECEIVED	B [7]	--
MAP_NAME	22	--
MC_RECEIVE_IMMEDIATE	20	--
RESOURCE	20	--
LENGTH	20 [10]	--
RETURN_CODE	20	--
REQUEST_TO_SEND_RECEIVED	20 [9]	--
DATA	20	--
WHAT_RECEIVED	20 [7]	--
MAP_NAME	20 and 22	--
MC_REQUEST_TO_SEND	B [8]	B
RESOURCE	B	--

Figure A-2. Support for Mapped Conversation Verbs and Parameters (Part 2 of 3)

Verb and Parameter	Local Support	Remote Support
MC_SEND_DATA	B	B
RESOURCE	B	--
DATA	B	B
LENGTH	B [10]	B
MAP_NAME(NO)	B [11]	B
MAP_NAME(YES(variable))	22	22
FMH_DATA(NO)	B	B
FMH_DATA(YES)	23 [12]	23 [12]
RETURN_CODE	B	--
REQUEST_TO_SEND_RECEIVED	B	--
MC_SEND_ERROR	B	B
RESOURCE	B	--
RETURN_CODE	B	--
REQUEST_TO_SEND_RECEIVED	B	--
MC_TEST	19 or 21	--
RESOURCE	19 or 21	--
TEST(POSTED)	19	--
TEST(REQUEST_TO_SEND_RECEIVED)	21	--
RETURN_CODE	19 or 21	--

Figure A-3. Support for Mapped Conversation Verbs and Parameters (Part 3 of 3)

SUPPORT FOR TYPE-INDEPENDENT CONVERSATION VERBS AND PARAMETERS

Verb and Parameter	Local Support	Remote Support
BACKOUT	4	4
GET_TYPE	25	--
RESOURCE	25	--
TYPE	25	--
SYNCPT	4	4
RETURN_CODE	4	--
REQUEST_TO_SEND_RECEIVED	4	--
WAIT	18	--
RESOURCE_LIST	18	--
RETURN_CODE	18	--
RESOURCE_POSTED	18	--

Figure A-4. Support for Type-Independent Conversation Verbs and Parameters

SUPPORT FOR BASIC CONVERSATION VERBS AND PARAMETERS

Verb and Parameter	Local Support	Remote Support
ALLOCATE	B	B [1]
LU_NAME(OWN)	1	--
LU_NAME(OTHER(variable))	B	B
MODE_NAME	B	B
TPN	B	B
TYPE(BASIC_CONVERSATION)	B	B
TYPE(MAPPED_CONVERSATION)	26	B
RETURN_CONTROL(WHEN_SESSION_ALLOCATED)	B [2]	--
RETURN_CONTROL(DELAYED_ALLOCATION_PERMITTED)	2	--
RETURN_CONTROL(IMMEDIATE)	3	--
SYNC_LEVEL(NONE)	B	B
SYNC_LEVEL(CONFIRM)	B	B
SYNC_LEVEL(SYNCPT)	4	4
SECURITY(NONE)	B	B
SECURITY(SAME)	6 or 10 [3]	6, 8, 9 or 10
SECURITY(PGM(USER_ID(variable)	7	6 or 8
PASSWORD(variable)	7	6
PROFILE(variable)))	11	9 or 10
PIP(NO)	B	B
PIP(YES(variable))	12 [4]	12 [4]
RESOURCE	B	--
RETURN_CODE	B	--
CONFIRM	B	B
RESOURCE	B	--
RETURN_CODE	B	--
REQUEST_TO_SEND_RECEIVED	B	--
CONFIRMED	B	B
RESOURCE	B	--
DEALLOCATE	B	B
RESOURCE	B	--
TYPE(SYNC_LEVEL)	B	B
TYPE(FLUSH)	B	B
TYPE(CONFIRM)	B	B
TYPE(ABEND_PROG)	B	B
TYPE(ABEND_SVC)	26 [14]	B
TYPE(ABEND_TIMER)	26 [14]	B
TYPE(LOCAL)	B	--
LOG_DATA(NO)	B	B
LOG_DATA(YES(variable))	13	B [15]
RETURN_CODE	B	--
FLUSH	14	B
RESOURCE	14	--

Figure A-5. Support for Basic Conversation Verbs and Parameters (Part 1 of 3)

Verb and Parameter	Local Support	Remote Support
GET_ATTRIBUTES	B	--
RESOURCE	B	--
OWN_FULLY_QUALIFIED_LU_NAME	B	--
PARTNER_LU_NAME	B	--
PARTNER_FULLY_QUALIFIED_LU_NAME	B	--
MODE_NAME	B	--
SYNC_LEVEL	B	--
SECURITY_USER_ID	6	--
SECURITY_PROFILE	9	--
LUW_IDENTIFIER	4 or 15	--
CONVERSATION_CORRELATOR	4	--
POST_ON_RECEIPT	18 or 19 [5]	--
RESOURCE	18 or 19	--
FILL(LL)	18 or 19 [16]	--
FILL(BUFFER)	18 or 19 [16]	--
LENGTH	18 or 19	--
PREPARE_TO_RECEIVE	16	B
RESOURCE	16	--
TYPE(SYNC_LEVEL)	16	B
TYPE(FLUSH)	16	B
TYPE(CONFIRM)	16	B
LOCKS(SHORT)	16	B
LOCKS(LONG)	17	B
RETURN_CODE	16	--
RECEIVE_AND_WAIT	B [6]	--
RESOURCE	B	--
FILL(LL)	B [16]	--
FILL(BUFFER)	B [16]	--
LENGTH	B	--
RETURN_CODE	B	--
REQUEST_TO_SEND_RECEIVED	B [9]	--
DATA	B	--
WHAT_RECEIVED	B [7]	--
RECEIVE_IMMEDIATE	20	--
RESOURCE	20	--
FILL(LL)	20 [16]	--
FILL(BUFFER)	20 [16]	--
LENGTH	20	--
RETURN_CODE	20	--
REQUEST_TO_SEND_RECEIVED	20 [9]	--
DATA	20	--
WHAT_RECEIVED	20 [7]	--
REQUEST_TO_SEND	B [8]	B
RESOURCE	B	--

Figure A-6. Support for Basic Conversation Verbs and Parameters (Part 2 of 3)

Verb and Parameter	Local Support	Remote Support
SEND_DATA	B	B
RESOURCE	B	--
DATA	B	B
LENGTH	B	B
RETURN_CODE	B	--
REQUEST_TO_SEND_RECEIVED	B	--
SEND_ERROR	B	B
RESOURCE	B	--
TYPE(PROG)	B	B
TYPE(SVC)	26 [14]	B
LOG_DATA(NO)	B	B
LOG_DATA(YES(variable))	13	B [15]
RETURN_CODE	B	--
REQUEST_TO_SEND_RECEIVED	B	--
TEST	19 or 21	--
RESOURCE	19 or 21	--
TEST(POSTED)	19	--
TEST(REQUEST_TO_SEND_RECEIVED)	21	--
RETURN_CODE	19 or 21	--

Figure A-7. Support for Basic Conversation Verbs and Parameters (Part 3 of 3)

SUPPORT FOR CONVERSATION RETURN CODES AND WHAT-RECEIVED INDICATIONS

Return Code	Local Support
ALLOCATION_ERROR	B
ALLOCATION_FAILURE_NO_RETRY	B
ALLOCATION_FAILURE_RETRY	B
CONVERSATION_TYPE_MISMATCH	B
PIP_NOT_ALLOWED	12
PIP_NOT_SPECIFIED_CORRECTLY	B
SECURITY_NOT_VALID	6, 7, 10 or 11
SYNC_LEVEL_NOT_SUPPORTED_BY_PGM	B
SYNC_LEVEL_NOT_SUPPORTED_BY_LU	4
TPN_NOT_RECOGNIZED	B
TRANS_PGM_NOT_AVAIL_NO_RETRY	B
TRANS_PGM_NOT_AVAIL_RETRY	B
BACKED_OUT	4
DEALLOCATE_ABEND_PROG	B
DEALLOCATE_ABEND_SVC	B
DEALLOCATE_ABEND_TIMER	B
DEALLOCATE_NORMAL	B
FMH_DATA_NOT_SUPPORTED	23
HEURISTIC_MIXED	4
MAP_EXECUTION_FAILURE	22
MAP_NOT_FOUND	22
MAPPING_NOT_SUPPORTED	22
OK	B
DATA	18 or 19
NOT_DATA	18 or 19
PARAMETER_ERROR	B
POSTING_NOT_ACTIVE	18 or 19
PROG_ERROR_NO_TRUNC	B
PROG_ERROR_PURGING	B
RESOURCE_FAILURE_NO_RETRY	B
RESOURCE_FAILURE_RETRY	B
SVC_ERROR_NO_TRUNC	B
SVC_ERROR_PURGING	B
SVC_ERROR_TRUNC	B
UNSUCCESSFUL	3, 19, 20 or 21

Figure A-8. Support for Conversation Return Codes

What-Received Indication	Local Support
CONFIRM	B
CONFIRM_DEALLOCATE	B
CONFIRM_SEND	B
DATA	B
DATA_COMPLETE	B
DATA_INCOMPLETE	B
DATA_TRUNCATED	B [13]
FMH_DATA_COMPLETE	23
FMH_DATA_INCOMPLETE	23
FMH_DATA_TRUNCATED	23 [13]
LL_TRUNCATED	B [17]
SEND	B
TAKE_SYNCPT	4
TAKE_SYNCPT_DEALLOCATE	4
TAKE_SYNCPT_SEND	4

Figure A-9. Support for Conversation What-Received Indications

SUPPORT FOR CONTROL-OPERATOR VERBS AND PARAMETERS FOR CNOS

Verb and Parameter	Local Support	Remote Support
CHANGE_SESSION_LIMIT	27	B
LU_NAME	27	B
MODE_NAME	27	B
LU_MODE_SESSION_LIMIT	27	B
MIN_CONWINNERS_SOURCE	27	B
MIN_CONWINNERS_TARGET	28	B
RESPONSIBLE(SOURCE)	27	B
RESPONSIBLE(TARGET)	29	B [19]
RETURN_CODE	27	--
INITIALIZE_SESSION_LIMIT	B	B
LU_NAME	B	B
MODE_NAME	B	B
LU_MODE_SESSION_LIMIT	B	B
MIN_CONWINNERS_SOURCE	B	B
MIN_CONWINNERS_TARGET	28	B
RETURN_CODE	B	--
PROCESS_SESSION_LIMIT	B	B
LU_NAME	B	--
MODE_NAME	B	--
RETURN_CODE	B	--
RESET_SESSION_LIMIT	B	B
LU_NAME	B	B
MODE_NAME(ALL)	B	B
MODE_NAME(ONE(variable))	B	B
RESPONSIBLE(SOURCE)	B	B
RESPONSIBLE(TARGET)	29	B [19]
DRAIN_SOURCE(NO)	B [18]	B
DRAIN_SOURCE(YES)	B [18]	B
DRAIN_TARGET(NO)	30	B
DRAIN_TARGET(YES)	B	B [20]
FORCE(NO)	B	B
FORCE(YES)	31	B
RETURN_CODE	B	--

Figure A-10. Support for Control Operator Verbs and Parameters for CNOS

SUPPORT FOR CONTROL-OPERATOR VERBS AND PARAMETERS FOR SESSION CONTROL

Verb and Parameter	Local Support	Remote Support
ACTIVATE_SESSION	32	B
LU_NAME	32	B
MODE_NAME	32	B
RETURN_CODE	32	--
DEACTIVATE_SESSION	33	B
SESSION_ID	33	--
TYPE(CLEANUP)	33	B
TYPE(NORMAL)	33	B
RETURN_CODE	33	--

Figure A-11. Support for Control Operator Verbs and Parameters for Session Control

SUPPORT FOR CONTROL-OPERATOR VERBS AND PARAMETERS FOR LU DEFINITION

Verb and Parameter	Local Support	Remote Support
DEFINE_LOCAL_LU	34	--
FULLY_QUALIFIED_LU_NAME	34	--
LU_SESSION_LIMIT(NONE)	34	--
LU_SESSION_LIMIT(VALUE(variable))	34 and 35	--
SECURITY(ADD(USER_ID(variable)	34 and 6	--
PASSWORD(variable)	34 and 6	--
PROFILE(variable)))	34 and 9	--
SECURITY(DELETE(USER_ID(variable)	34 and 6	--
PROFILE(variable)))	34 and 9	--
MAP_NAME(ADD(variable))	34 and 22	--
MAP_NAME(DELETE(variable))	34 and 22	--
RETURN_CODE	34	--
DEFINE_REMOTE_LU	34	--
FULLY_QUALIFIED_LU_NAME	34	--
LOCALLY_KNOWN_LU_NAME(NONE)	34	--
LOCALLY_KNOWN_LU_NAME(NAME(variable))	34 and 36	--
UNINTERPRETED_LU_NAME(NONE)	34	--
UNINTERPRETED_LU_NAME(NAME(variable))	34 and 37	--
INITIATE_TYPE(INITIATE_ONLY)	34	--
INITIATE_TYPE(INITIATE_OR_QUEUE)	34	--
PARALLEL_SESSION_SUPPORT(YES)	34	--
PARALLEL_SESSION_SUPPORT(NO)	34	--
CNOS_SUPPORT(YES)	34	--
CNOS_SUPPORT(NO)	34	--
LU_LU_PASSWORD(NONE)	34	--
LU_LU_PASSWORD(VALUE(variable))	34 and 5	--
SECURITY_ACCEPTANCE(NONE)	34	--
SECURITY_ACCEPTANCE(CONVERSATION)	34 and 6	--
SECURITY_ACCEPTANCE(ALREADY_VERIFIED)	34 and 6	--
RETURN_CODE	34	--
DEFINE_MODE	34 [21]	--
FULLY_QUALIFIED_LU_NAME	34	--
MODE_NAME	34	--
SEND_PACING_WINDOW	34	--
RECEIVE_PACING_WINDOW	34	--
SEND_MAX_RU_SIZE_LOWER_BOUND(256)	34	--
SEND_MAX_RU_SIZE_LOWER_BOUND(¬256)	34 and 39	--
SEND_MAX_RU_SIZE_UPPER_BOUND(256)	34	--
SEND_MAX_RU_SIZE_UPPER_BOUND(¬256)	34 and 39	--
RECEIVE_MAX_RU_SIZE_LOWER_BOUND(256)	34	--
RECEIVE_MAX_RU_SIZE_LOWER_BOUND(¬256)	34 and 39	--
RECEIVE_MAX_RU_SIZE_UPPER_BOUND(256)	34	--
RECEIVE_MAX_RU_SIZE_UPPER_BOUND(¬256)	34 and 39	--
SYNC_LEVEL_SUPPORT(CONFIRM)	34	--
SYNC_LEVEL_SUPPORT(CONFIRM_SYNCPT)	34 and 4	--
SINGLE_SESSION_REINITIATION(OPERATOR)	34	--
SINGLE_SESSION_REINITIATION(PLU)	34 and 38	--
SINGLE_SESSION_REINITIATION(SLU)	34 and 38	--
SINGLE_SESSION_REINITIATION(PLU_OR_SLU)	34 and 38	--
SESSION_LEVEL_CRYPTOGRAPHY(NO)	34	--
SESSION_LEVEL_CRYPTOGRAPHY(YES)	34 and 40	--
CONWINNER_AUTO_ACTIVATE_LIMIT	34 and 41	--
RETURN_CODE	34	--

Figure A-12. Support for Control Operator Verbs and Parameters for LU Definition (Part 1 of 3)

Verb and Parameter	Local Support	Remote Support
DEFINE_TP	34	--
TP_NAME	34	--
STATUS(ENABLED)	34	--
STATUS(TEMP_DISABLED)	34	--
STATUS(PERM_DISABLED)	34	--
CONVERSATION_TYPE(MAPPED)	34	--
CONVERSATION_TYPE(BASIC)	34	--
SYNC_LEVEL(NONE)	34	--
SYNC_LEVEL(CONFIRM)	34	--
SYNC_LEVEL(SYNCPT)	34 and 4	--
SECURITY_REQUIRED(NONE)	34	--
SECURITY_REQUIRED(CONVERSATION)	34 and 6	--
SECURITY_REQUIRED(ACCESS(PROFILE))	34 and 9	--
SECURITY_REQUIRED(ACCESS(USER_ID))	34 and 8	--
SECURITY_REQUIRED(ACCESS(USER_ID_PROFILE))	34, 8 and 9	--
SECURITY_ACCESS(ADD(USER_ID(variable)	34 and 8	--
PROFILE(variable)))	34 and 9	--
SECURITY_ACCESS(DELETE(USER_ID(variable)	34 and 8	--
PROFILE(variable)))	34 and 9	--
PIP(NO)	34	--
PIP(YES(variable))	34 and 12	--
DATA_MAPPING(NO)	34	--
DATA_MAPPING(YES)	34 and 22	--
FMH_DATA(NO)	34	--
FMH_DATA(YES)	34 and 23	--
PRIVILEGE(NONE)	34	--
PRIVILEGE(CNOS)	34	--
PRIVILEGE(SESSION_CONTROL)	34, and 32 or 33	--
PRIVILEGE(DEFINE)	34	--
PRIVILEGE(DISPLAY)	34	--
PRIVILEGE(ALLOCATE_SERVICE_TP)	34	--
RETURN_CODE	34	--
DISPLAY_LOCAL_LU	34	--
FULLY_QUALIFIED_LU_NAME	34	--
RETURN_CODE	34	--
LU_SESSION_LIMIT	34	--
LU_SESSION_COUNT	34	--
SECURITY	34 and 6	--
MAP_NAMES	34	--
REMOTE_LU_NAMES	34	--
TP_NAMES	34	--
DISPLAY_REMOTE_LU	34	--
FULLY_QUALIFIED_LU_NAME	34	--
RETURN_CODE	34	--
LOCALLY_KNOWN_LU_NAME	34 and 36	--
UNINTERPRETED_LU_NAME	34 and 37	--
INITIATE_TYPE	34	--
PARALLEL_SESSION_SUPPORT	34	--
CNOS_SUPPORT	34	--
SECURITY_ACCEPTANCE_LOCAL_LU	34 and 6	--
SECURITY_ACCEPTANCE_REMOTE_LU	34 and 6	--
MODE_NAMES	34	--

Figure A-13. Support for Control Operator Verbs and Parameters for LU Definition (Part 2 of 3)

Verb and Parameter	Local Support	Remote Support
DISPLAY_MODE	34	--
FULLY_QUALIFIED_LU_NAME	34	--
MODE_NAME	34	--
RETURN_CODE	34	--
SEND_PACING_WINDOW	34	--
RECEIVE_PACING_WINDOW	34	--
SEND_MAX_RU_SIZE_LOWER_BOUND	34 and 39	--
SEND_MAX_RU_SIZE_UPPER_BOUND	34 and 39	--
RECEIVE_MAX_RU_SIZE_LOWER_BOUND	34 and 39	--
RECEIVE_MAX_RU_SIZE_UPPER_BOUND	34 and 39	--
SYNC_LEVEL_SUPPORT	34	--
SINGLE_SESSION_REINITIATION	34 and 38	--
SESSION_LEVEL_CRYPTOGRAPHY	34 and 40	--
CONWINNER_AUTO_ACTIVATE_LIMIT	34 and 41	--
LU_MODE_SESSION_LIMIT	34	--
MIN_CONWINNERS	34	--
MIN_CONLOSERS	34	--
TERMINATION_COUNT	34	--
DRAIN_LOCAL_LU	34	--
DRAIN_REMOTE_LU	34	--
LU_MODE_SESSION_COUNT	34	--
CONWINNERS_SESSION_COUNT	34	--
CONLOSERS_SESSION_COUNT	34	--
SESSION_IDS	34	--
DISPLAY_TP	34	--
TP_NAME	34	--
RETURN_CODE	34	--
STATUS	34	--
CONVERSATION_TYPE	34	--
SYNC_LEVEL	34	--
SECURITY_REQUIRED	34 and 6	--
SECURITY_ACCESS	34, and 8 or 9	--
PIP	34 and 12	--
DATA_MAPPING	34 and 22	--
FMH_DATA	34 and 23	--
PRIVILEGE	34	--
DELETE	34	--
LOCAL_LU_NAME	34	--
REMOTE_LU_NAME	34	--
MODE_NAME	34	--
TP_NAME	34	--
RETURN_CODE	34	--

Figure A-14. Support for Control Operator Verbs and Parameters for LU Definition (Part 3 of 3)

SUPPORT FOR CONTROL-OPERATOR RETURN CODES

Return Code	Local Support
ACTIVATION_FAILURE_NO_RETRY	32
ACTIVATION_FAILURE_RETRY	32
ALLOCATION_ERROR ALLOCATION_FAILURE_NO_RETRY ALLOCATION_FAILURE_RETRY TRANS_PGM_NOT_AVAIL_RETRY	B B B B
COMMAND_RACE_REJECT	B
LU_MODE_SESSION_LIMIT_CLOSED	B
LU_MODE_SESSION_LIMIT_EXCEEDED	32
LU_MODE_SESSION_LIMIT_NOT_ZERO	B
LU_MODE_SESSION_LIMIT_ZERO	27
LU_SESSION_LIMIT_EXCEEDED	B
OK AS_SPECIFIED AS_NEGOTIATED FORCED	B B B 31
PARAMETER_ERROR	B
REQUEST_EXCEEDS_MAX_ALLOWED	B
RESOURCE_FAILURE_NO_RETRY	B
UNRECOGNIZED_MODE_NAME	B

Figure A-15. Support for Control Operator Return Codes

NOTES ON IMPLEMENTATION DETAILS

The following notes pertain to the base and optional support shown in the preceding figures. These notes describe certain implementation details, which are product dependent.

Notes that Apply to Conversation Verbs:

1. The MC_ALLOCATE and ALLOCATE verbs send an allocation request to the remote LU. The remote LU starts the transaction program named in the allocation request. As this function is described in this book, the remote LU starts a new execution instance of the named transaction program. All products support this capability. A product may, in addition, allow an already-executing instance of the named transaction program to receive an allocation request by means of a product-dependent verb. This product-dependent capability provides no means in the LU for correlating the new conversation to a previous one.

2. The RETURN_CONTROL parameter on the MC_ALLOCATE and ALLOCATE verbs specifies when local processing of the verb is to be completed, in terms of the allocation of a session for the conversation. A product may provide for the specification of additional conditions for allocating sessions, such as a variation of the argument WHEN_SESSION_ALLOCATED that omits contention-loser sessions from the selection process.

3. Products that provide local support for option set 6 (conversation-level security verification) but not option set 7 (program supplied user ID and password) may choose to not make the SECURITY parameter on the MC_ALLOCATE and ALLOCATE verbs explicitly available to their transaction programs. Such products implicitly support both SECURITY(SAME) and SECURITY(NONE) in that they downgrade SECURITY(SAME) to SECURITY(NONE) when the remote LU does not accept conversation-level security, or the already-verified indication, from the local LU.

4. Support of the PIP parameter on the MC_ALLOCATE and ALLOCATE verbs is optional and local support is independent of remote support. A product may provide either local or remote support, or both.

5. The MC_POST_ON_RECEIPT and POST_ON_RECEIPT verbs, as described in this book, may be issued for a given conversation any number of times before posting is reset or cancelled. (See the notes under the descriptions of the verbs for more details.) However, a product may, instead, permit the verbs to be issued only once for a given conversation and disallow subsequent use of the verbs on that conversation until posting is reset or cancelled.

6. The MC_RECEIVE_AND_WAIT and RECEIVE_AND_WAIT verbs are used to wait until the requested amount of data or other information is available for the program to receive, receive the data or other information, and then resume program execution. Rather than resuming program execution as soon as the requested amount of data is received, the product may defer resuming program execution until it receives something other than data, such as a SEND, CONFIRM, TAKE_SYNCPT, or DEALLOCATE indication.

7. The WHAT_RECEIVED parameter on the MC_RECEIVE_AND_WAIT, MC_RECEIVE_IMMEDIATE, RECEIVE_AND_WAIT, and RECEIVE_IMMEDIATE verbs returns to the transaction program only one indication at a time. This serialization of returning what-received indications to the program results in discrete states of the conversation for each returned indication. (See the verb descriptions in this book for the state changes that can occur.) A product may, instead, return more than one indication at a time. For example, the product may return indications for both DATA and SEND at the completion of the verbs. In this case, the state changes that occur at the completion of the verbs may differ from that described in this book. Refer to the product's publication for further details.

8. The MC_REQUEST_TO_SEND and REQUEST_TO_SEND verbs, as described in this book, may be issued only when the conversation is in receive,

confirm, or sync point state. Issuing the verbs for a conversation in any other state is described as a state-check ABEND condition. As an alternative to the ABEND condition, a product may also permit its transaction programs to issue the verbs for conversations in send or defer state.

9. The REQUEST_TO_SEND_RECEIVED parameter on the MC_RECEIVE_AND_WAIT, MC_RECEIVE_IMMEDIATE, RECEIVE_AND_WAIT, and RECEIVE_IMMEDIATE verbs is used to receive a request-to-send notification when the conversation is in receive state. (See the notes under the descriptions of these verbs for details of when this can occur.) However, a product may defer passing the request-to-send notification to the program until the program issues an MC_SEND_DATA or SEND_DATA verb, or issues an MC_SEND_ERROR or SEND_ERROR verb when the conversation is in send state.

   Also, a product that defers passing the request-to-send notification to the program may discard the notification, and not pass it to the program, if the program issues an MC_PREPARE_TO_RECEIVE or PREPARE_TO_RECEIVE verb, issues an MC_RECEIVE_AND_WAIT or RECEIVE_AND_WAIT verb when the conversation is in send state, or receives a PROG_ERROR_PURGING or SVC_ERROR_PURGING return code when the conversation is in send state.

Notes that Apply Only to Mapped Conversation Verbs:

10. A base local and remote support is defined for the data record length specified by the LENGTH parameter on the MC_POST_ON_RECEIPT, MC_RECEIVE_AND_WAIT, MC_RECEIVE_IMMEDIATE, and MC_SEND_DATA verbs. The base support for the data record length is 2048. All transaction programs are allowed to send and receive data records up to 2048 bytes in length. Local and remote support for data records greater than 2048 bytes in length is optional, and the maximum length is product-dependent.

11. The MAP_NAME parameter on the MC_SEND_DATA verb is used to specify data mapping. MAP_NAME(NO) yields a null value for the map name, which suppresses data mapping, whereas MAP_NAME(YES(variable)) specifies a non-null map name, which invokes data mapping. Products that support option set 22 (data mapping) provide local and remote support for both MAP_NAME(NO) and MAP_NAME(YES(variable)). However, a product may provide local support only for MAP_NAME(YES(variable)) on MC_SEND_DATAs issued on conversations that use data mapping.

12. The FMH_DATA(YES) parameter on the MC_SEND_DATA verb specifies that the data record contains FM header data. Transaction programs written for a product that implements LU 6.1 make use of the specification of FM header data. A product that implements LU 6.2 may provide local and remote support for this parameter, either because it processes LU 6.1 programs or because it processes LU 6.2 programs that connect to LU 6.1 programs.

13. The what-received indications, DATA_TRUNCATED and FMH_DATA_TRUNCATED, inform the program that it received only part of the data record and the LU has discarded the remaining part. A product may, instead, retain the remaining data and indicate DATA_INCOMPLETE or FMH_DATA_INCOMPLETE. Alternatively, the product may support both capabilities and allow the program to select whether the LU is to discard or retain remaining data.

Notes that Apply Only to Basic Conversation Verbs:

14. The TYPE(ABEND_SVC) and TYPE(ABEND_TIMER) parameters on the DEALLOCATE verb and the TYPE(SVC) parameter on the SEND_ERROR verb are used to indicate errors that the mapped conversation LU services component detects. A product that does not support option set 26 (mapped conversation LU services component) may, nevertheless, support these parameters at its basic conversation protocol boundary.

15. The LOG_DATA parameter on the DEALLOCATE and SEND_ERROR verbs is used to record product-unique error information in the system

error logs of the local and remote LUs. The capability to receive the log data is part of the base remote support. However, a product that does not provide local support for option set 13 (logging of data in a system log) may discard the received log data rather than process it.

16. The FILL parameter on the POST_ON_RECEIPT, RECEIVE_AND_WAIT, and RECEIVE_IMMEDIATE verbs has two arguments: LL and BUFFER. A product may support only one of the arguments, or it may support both of them.

    The arguments are described in this book as being independent of each other; that is, the specification of either one does not depend on the past use of the parameter, and has no bearing on its subsequent use. A product supporting both arguments may treat them as described, or it may treat them in a dependent manner, allowing the program to specify only one or the other for certain sequences of the verbs and indicating an error if this restriction is violated.

17. The what-received indication, LL_TRUNCATED, informs the program that the LU received only the first byte of the LL field of a logical record, because it was truncated. The truncated LL field is discarded by the receiving LU rather than being passed to the receiving program. A product may, instead, pass the truncated LL field to the program and indicate DATA_INCOMPLETE rather than LL_TRUNCATED.

Notes that Apply to Control-Operator Verbs:

18. A product may provide local (source LU) support for only one of the arguments, NO or YES, of the DRAIN_SOURCE parameter on the RESET_SESSION_LIMIT verb, or it may support both arguments. However, all products provide remote (target LU) support for both arguments.

19. Remote support for the RESPONSIBLE(TARGET) parameter of the CHANGE_SESSION_LIMIT and RESET_SESSION_LIMIT verbs is part of the base set of functions. However, a product may provide remote support for this parameter by always negotiating the TARGET argument to SOURCE during its remote processing, as a target LU, of CHANGE_SESSION_LIMIT and RESET_SESSION_LIMIT.

20. Remote support for the DRAIN_TARGET(YES) parameter of the the RESET_SESSION_LIMIT verb is part of the base set of functions. However, a product may provide remote support for this parameter by always negotiating the YES argument to NO during its remote processing, as a target LU, of RESET_SESSION_LIMIT.

21. A product may allow the control operator or transaction program to specify certain parameters of the DEFINE_MODE verb by means of the DEFINE_LOCAL_LU or DEFINE_REMOTE_LU verb, instead. These parameters are:

    SYNC_LEVEL_SUPPORT
    SINGLE_SESSION_REINITIATION
    SESSION_LEVEL_CRYPTOGRAPHY

    When these parameters are specified by means of DEFINE_LOCAL_LU, the local LU's corresponding operating parameters are constant across all mode names for all remote LUs. Similarly, when these parameters are specified by means of DEFINE_REMOTE_LU, the local LU's corresponding operating parameters are constant across all mode names for a given remote LU, but they may differ for each remote LU.

APPENDIX B. EXAMPLES USING BASIC CONVERSATION VERBS

This appendix contains examples of the use of some of the basic conversation verbs. Each example shows two transaction programs, TP(a) and TP(b), connected by a conversation. The letters a and b represent each program's name.

Each example concentrates on the use of one, two or three verbs, in conjunction with several other verbs, and how the verbs issued by one program relate to the verbs issued by the other program. When a verb causes the LU to send information to the other program, the resulting flow is shown as an arrow (——>). Some verbs cause the LU to suspend the program's processing until the LU completes execution of the verb; a vertical line (|) under the verb indicates the suspension of program processing.

Some parameters are shown with the verbs. The parameters shown are those that are significant to the example. Supplied parameters are shown as "parameter-name(supplied-value)," and returned parameters are shown as "parameter-name=returned-value." Parameters not significant to the example are not shown.

On the page facing the example are notes that explain what the example is illustrating. The notes are numbered. The part of the example to which the note applies is keyed with the same number, shown within braces. For instance, the part of the example in Figure B-1 on page B-2 that is keyed by "{1}" is explained by note 1 on the facing page.

The examples contain a few comments, which are shown within brackets. For example, the comment, "[TP(a) running]," in Figure B-1 on page B-2 means the program is already processing at the point the example begins.


```
[TP(a) running]

ALLOCATE  {1}
  TPN('b')
  SYNC_LEVEL(NONE)  {2}
  RETURN_CODE=OK  {3}

SEND_DATA  {4}
  RETURN_CODE=OK

DEALLOCATE  {5}
  TYPE(SYNC_LEVEL)
  RETURN_CODE=OK  {6}          ------------------------> [start TP(b)]  {8}

[end conversation]  {7}                                  RECEIVE_AND_WAIT  {9}
                                                           RETURN_CODE=OK
                                                           WHAT_RECEIVED=DATA_COMPLETE

                                                         RECEIVE_AND_WAIT  {10}
                                                           RETURN_CODE=DEALLOCATE_NORMAL

                                                         DEALLOCATE  {11}
                                                           TYPE(LOCAL)
                                                           RETURN_CODE=OK

                                                         [end conversation]  {12}
```

Figure B-1. ALLOCATE, SEND_DATA, DEALLOCATE -- SYNC_LEVEL(NONE)

Notes for Figure B-1:

1. TP(a) issues ALLOCATE to request a conversation with partner program 'b', designated by the TPN parameter.

2. The SYNC_LEVEL(NONE) parameter specifies a synchronization level of NONE, which means no confirmation or sync point processing.

3. The LU places the allocation request in its send buffer and returns control to TP(a) with the conversation in send state. Nothing is sent.

4. TP(a) issues SEND_DATA, causing the LU to place the data (a logical record) in its buffer behind the allocation request. Still, nothing is sent, because the LU does not yet have a sufficient amount of information for transmission (in this example).

5. TP(a) issues DEALLOCATE with TYPE(SYNC_LEVEL), which implies no synchronization and causes the LU to send the contents of its buffer together with a DEALLOCATE_NORMAL indication.

6. Because the synchronization level is NONE, DEALLOCATE with TYPE(SYNC_LEVEL) completes immediately and successfully. Contrast this with Figure B-2 on page B-4.

7. The conversation is deallocated from the session at the completion of DEALLOCATE.

8. TP(b) is started, with the conversation in receive state, when its LU receives the allocation request.

9. TP(b) issues RECEIVE_AND_WAIT and receives the complete logical record.

10. TP(b) issues another RECEIVE_AND_WAIT and receives the DEALLOCATE_NORMAL indication.

11. TP(b) issues DEALLOCATE with TYPE(LOCAL), causing the LU to discard its control information for the conversation.

12. The conversation ends for TP(b).

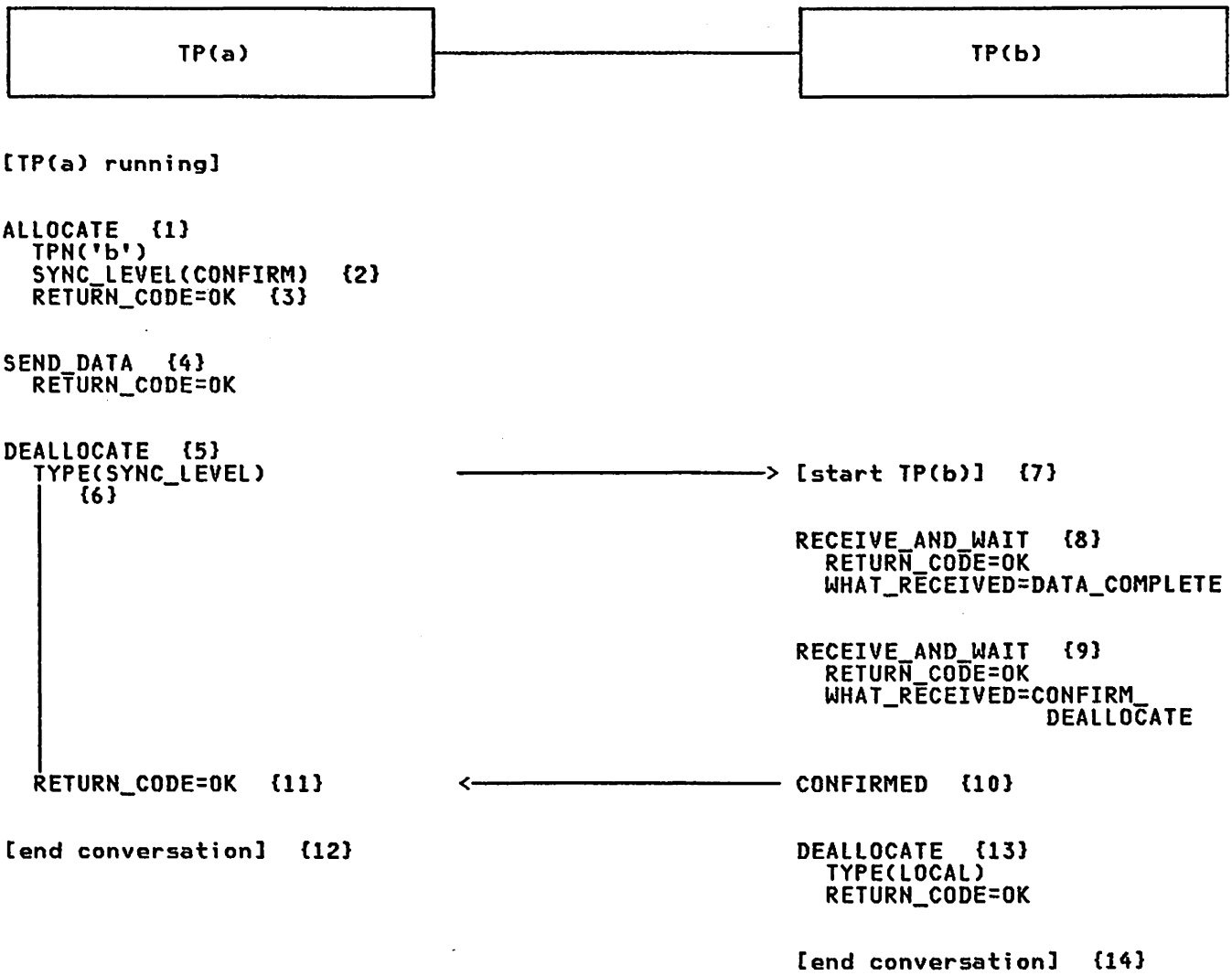


Figure B-2. ALLOCATE, SEND_DATA, DEALLOCATE -- SYNC_LEVEL(CONFIRM)

Notes for Figure B-2:

1. TP(a) issues ALLOCATE to request a conversation with partner program 'b'.

2. The SYNC_LEVEL(CONFIRM) parameter specifies a synchronization level of CONFIRM, which means confirmation processing is permitted.

3. The LU places the allocation request in its send buffer and returns control to TP(a) with the conversation in send state. Nothing is sent.

4. TP(a) issues SEND_DATA, causing the LU to place the data (a logical record) in its buffer behind the allocation request. Still, nothing is sent.

5. TP(a) issues DEALLOCATE with TYPE(SYNC_LEVEL), which implies confirmation processing and causes the LU to send the contents of its buffer together with a CONFIRM_DEALLOCATE request.

6. Because the synchronization level is CONFIRM, DEALLOCATE with TYPE(SYNC_LEVEL) causes the LU to suspend TP(a)'s processing until it receives a response, affirmative or negative.

7. TP(b) is started, with the conversation in receive state, when its LU receives the allocation request.

8. TP(b) issues RECEIVE_AND_WAIT and receives the complete logical record.

9. TP(b) issues another RECEIVE_AND_WAIT and receives the CONFIRM_DEALLOCATE request.

10. TP(b) responds affirmatively by issuing CONFIRMED, causing its LU to send a positive response. If TP(b) responded negatively by issuing SEND_ERROR (not shown), the conversation would remain allocated.

11. The LU returns control to TP(a), indicating an affirmative response and successful completion of the DEALLOCATE.

12. The conversation is deallocated from the session at the completion of DEALLOCATE.

13. TP(b) issues DEALLOCATE with TYPE(LOCAL), causing the LU to discard its control information for the conversation.

14. The conversation ends for TP(b).

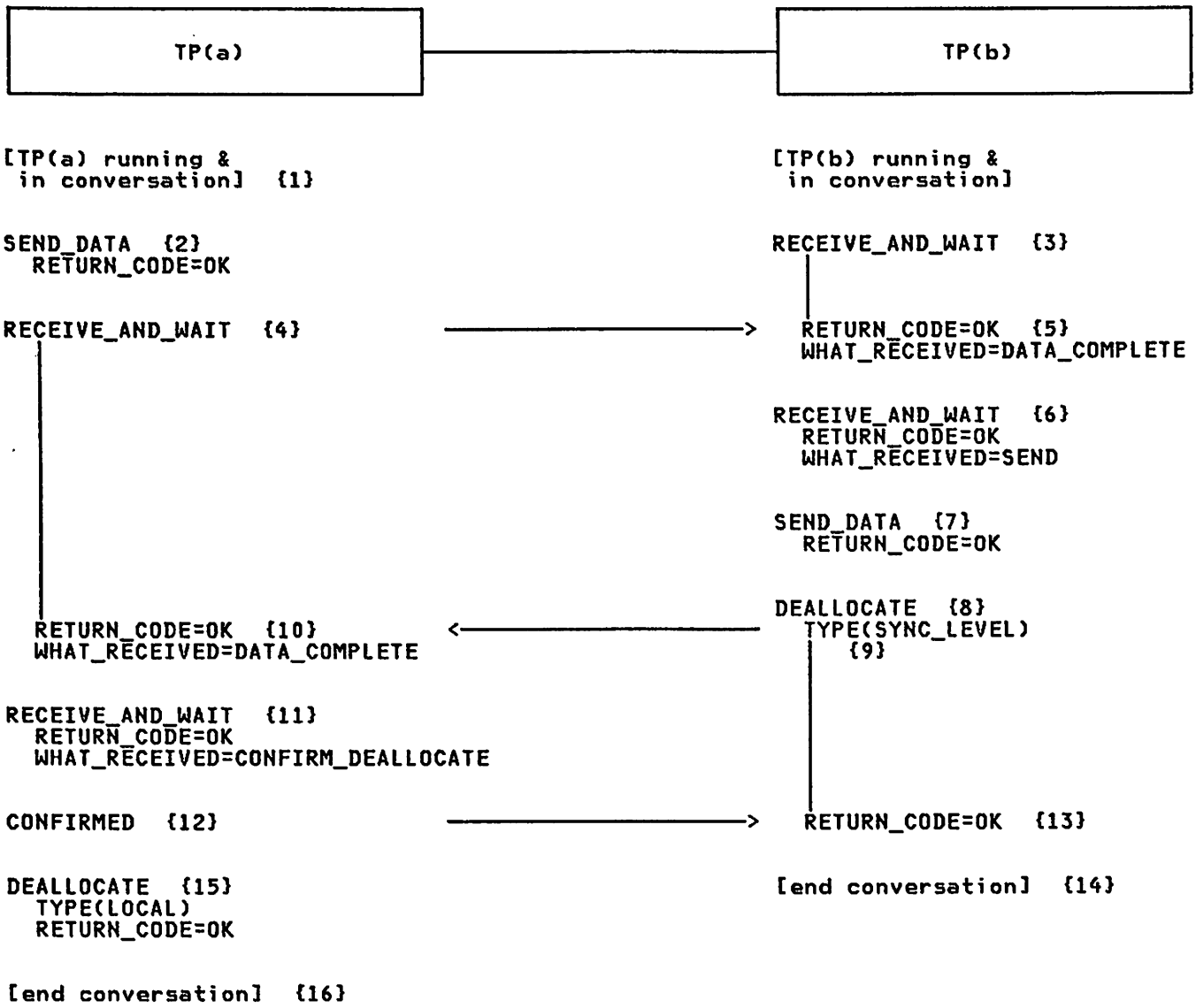


Figure B-3. RECEIVE_AND_WAIT, DEALLOCATE -- SYNC_LEVEL(CONFIRM)

Notes for Figure B-3:

1. Assume TP(a) has already allocated the conversation with SYNC_LEVEL(CONFIRM), and the conversation is now in send state for TP(a) and receive state for TP(b).

2. TP(a) issues SEND_DATA, causing the LU to place the data (a logical record) in its buffer. Nothing is sent.

3. TP(b) issues RECEIVE_AND_WAIT, causing the LU to suspend TP(b)'s processing until it receives information.

4. TP(a) issues RECEIVE_AND_WAIT, causing the LU to send the contents of its buffer together with the SEND indication. The LU suspends TP(a)'s processing until it receives information.

5. The LU returns control to TP(b), indicating that the program has received the complete logical record.

6. TP(b) issues another RECEIVE_AND_WAIT and receives the SEND indication.

7. TP(b) issues SEND_DATA, causing the LU to place the data (a logical record) in its buffer. Nothing is sent.

8. TP(b) issues DEALLOCATE with TYPE(SYNC_LEVEL), which implies confirmation processing and causes the LU to send the contents of its buffer together with a CONFIRM_DEALLOCATE request.

9. Because the synchronization level is CONFIRM, DEALLOCATE with TYPE(SYNC_LEVEL) causes the LU to suspend TP(b)'s processing until it receives a response, affirmative or negative.

10. The LU returns control to TP(a), indicating that the program has received the complete logical record.

11. TP(a) issues another RECEIVE_AND_WAIT and receives the CONFIRM_DEALLOCATE request.

12. TP(a) responds affirmatively by issuing CONFIRMED, causing its LU to send a positive response.

13. The LU returns control to TP(b), indicating an affirmative response and successful completion of the DEALLOCATE.

14. The conversation is deallocated from the session at the completion of DEALLOCATE.

15. TP(a) issues DEALLOCATE with TYPE(LOCAL), causing the LU to discard its control information for the conversation.

16. The conversation ends for TP(a).

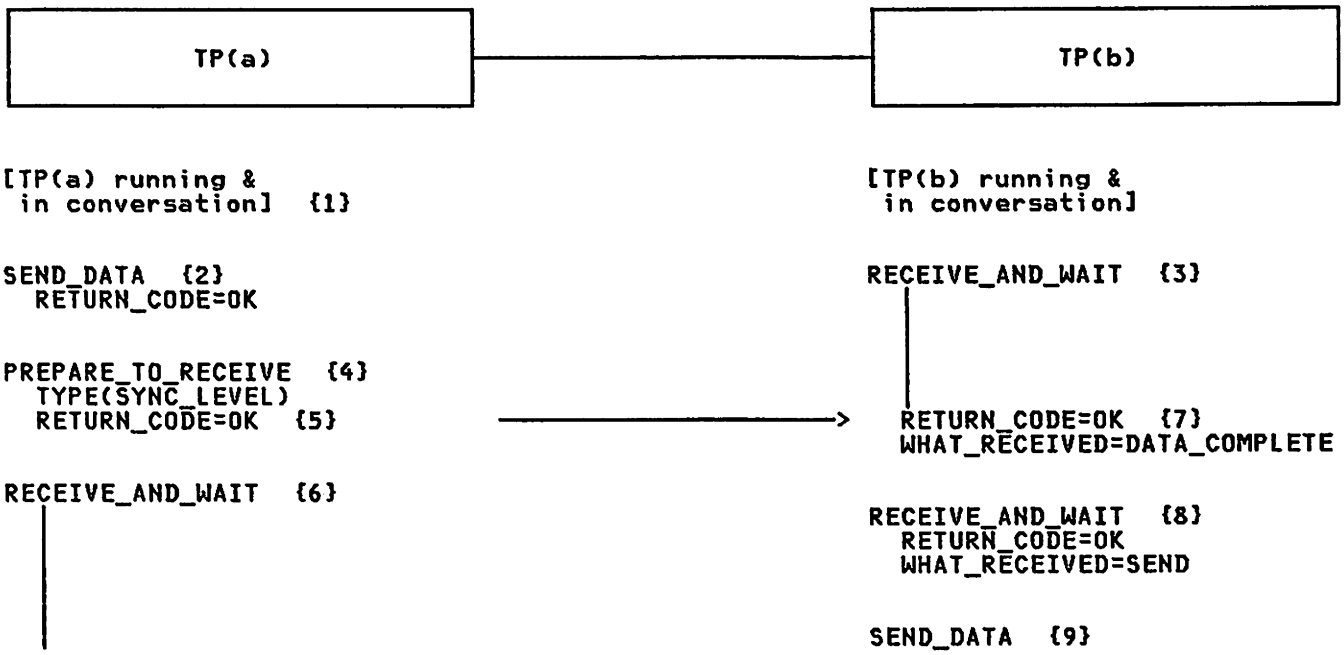


Figure B-4. PREPARE_TO_RECEIVE -- SYNC_LEVEL(NONE)

Notes for Figure B-4:

1. Assume TP(a) has already allocated the conversation with SYNC_LEVEL(NONE), and the conversation is now in send state for TP(a) and receive state for TP(b).

2. TP(a) issues SEND_DATA, causing the LU to place the data (a logical record) in its buffer. Nothing is sent.

3. TP(b) issues RECEIVE_AND_WAIT, causing the LU to suspend TP(b)'s processing until it receives information.

4. TP(a) issues PREPARE_TO_RECEIVE with TYPE(SYNC_LEVEL), which implies no synchronization and causes the LU to send the contents of its buffer together with the SEND indication.

5. Because the synchronization level is NONE, PREPARE_TO_RECEIVE with TYPE(SYNC_LEVEL) completes immediately and successfully. Contrast this with Figure B-5 on page B-10.

6. The conversation for TP(a) is now in receive state, so TP(a) issues a RECEIVE_AND_WAIT.

7. The LU returns control to TP(b), indicating that the program has received the complete logical record.

8. TP(b) issues another RECEIVE_AND_WAIT and receives the SEND indication.

9. The conversation for TP(b) is now in send state, so TP(b) issues a SEND_DATA.

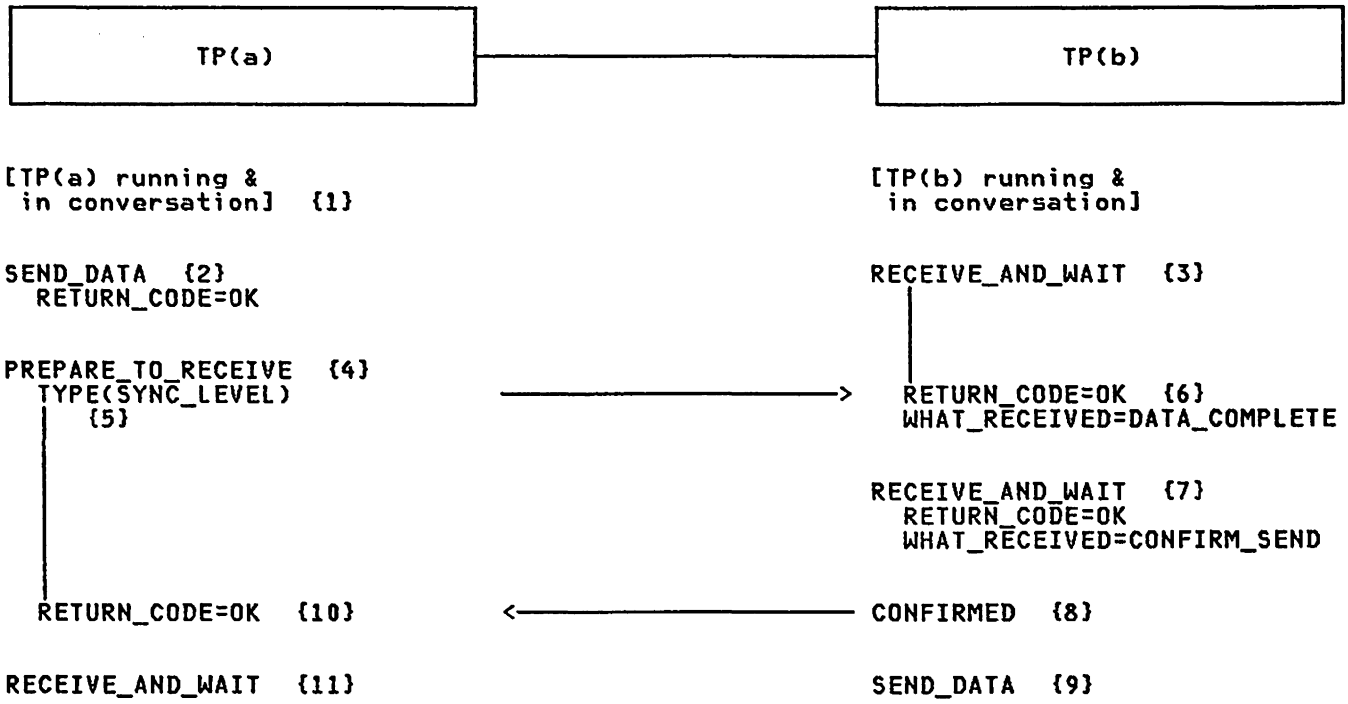


Figure B-5. PREPARE_TO_RECEIVE -- SYNC_LEVEL(CONFIRM)

Notes for Figure B-5:

1. Assume TP(a) has already allocated the conversation with SYNC_LEVEL(CONFIRM), and the conversation is now in send state for TP(a) and receive state for TP(b).

2. TP(a) issues SEND_DATA, causing the LU to place the data (a logical record) in its buffer. Nothing is sent.

3. TP(b) issues RECEIVE_AND_WAIT, causing the LU to suspend TP(b)'s processing until it receives information.

4. TP(a) issues PREPARE_TO_RECEIVE with TYPE(SYNC_LEVEL), which implies confirmation processing and causes the LU to send the contents of its buffer together with the CONFIRM_SEND request.

5. Because the synchronization level is CONFIRM, PREPARE_TO_RECEIVE with TYPE(SYNC_LEVEL) causes the LU to suspend TP(a)'s processing until it receives a response, affirmative or negative.

6. The LU returns control to TP(b), indicating that the program has received the complete logical record.

7. TP(b) issues another RECEIVE_AND_WAIT and receives the CONFIRM_SEND request.

8. TP(b) responds affirmatively by issuing CONFIRMED, causing its LU to send a positive response.

9. The conversation for TP(b) is now in send state, so TP(b) issues a SEND_DATA.

10. The LU returns control to TP(a), indicating an affirmative response and successful completion of the PREPARE_TO_RECEIVE.

11. The conversation for TP(a) is now in receive state, so TP(a) issues a RECEIVE_AND_WAIT.

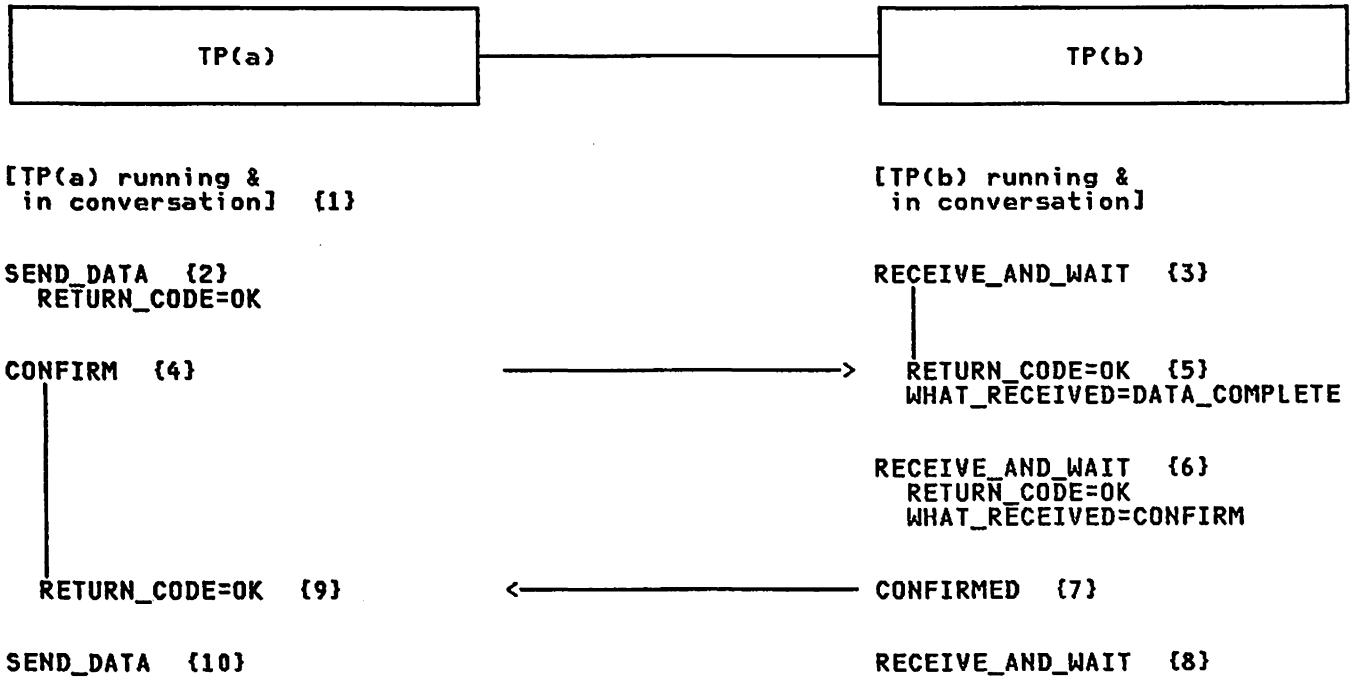


Figure B-6. CONFIRM

Notes for Figure B-6:

1. Assume TP(a) has already allocated the conversation with SYNC_LEVEL(CONFIRM), and the conversation is now in send state for TP(a) and receive state for TP(b).

2. TP(a) issues SEND_DATA, causing the LU to place the data (a logical record) in its buffer. Nothing is sent.

3. TP(b) issues RECEIVE_AND_WAIT, causing the LU to suspend TP(b)'s processing until it receives information.

4. TP(a) issues CONFIRM in order to synchronize the processing of the two programs. The CONFIRM verb causes the LU to send the contents of its buffer together with the CONFIRM request. The LU suspends TP(a)'s processing until it receives a response, affirmative or negative.

5. The LU returns control to TP(b), indicating that the program has received the complete logical record.

6. TP(b) issues another RECEIVE_AND_WAIT and receives the CONFIRM request.

7. TP(b) responds affirmatively by issuing CONFIRMED, causing its LU to send a positive response.

8. The conversation for TP(b) is still in receive state, so TP(b) issues another RECEIVE_AND_WAIT.

9. The LU returns control to TP(a), indicating an affirmative response and successful completion of the CONFIRM.

10. The conversation for TP(a) is still in send state, so TP(a) issues another SEND_DATA.

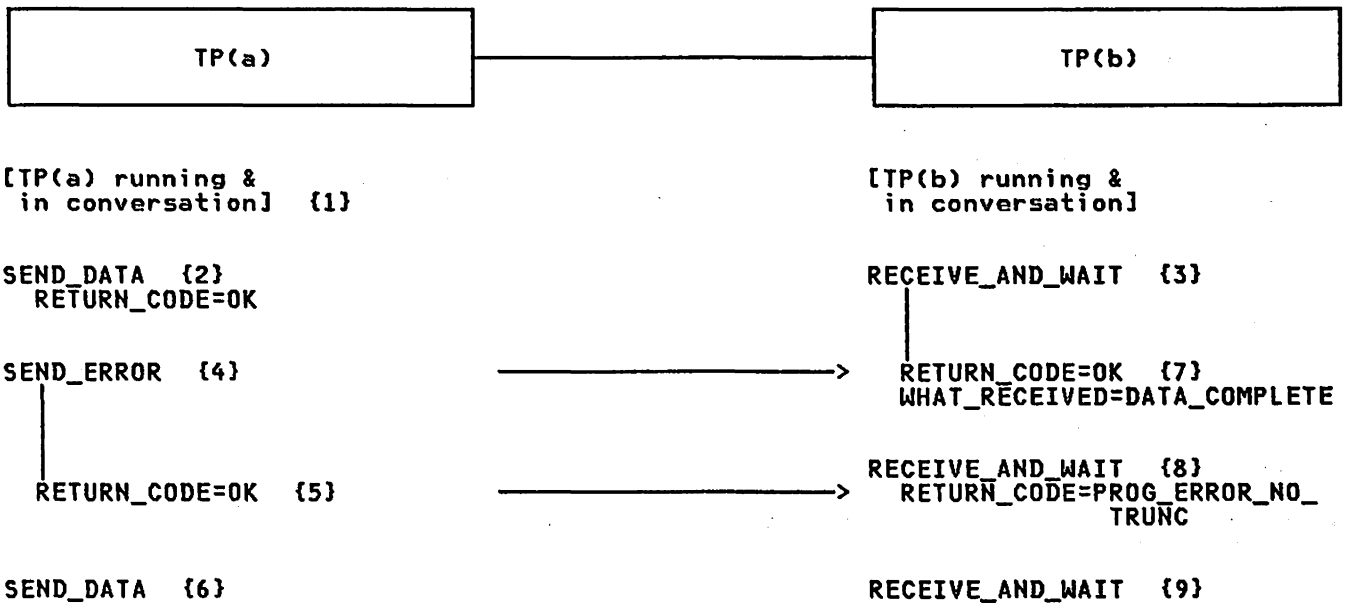


Figure B-7. SEND_ERROR in Send State

Notes for Figure B-7:

1. Assume TP(a) has already allocated the conversation and it is now send state for TP(a) and receive state for TP(b).

2. TP(a) issues SEND_DATA, causing the LU to place the data (a logical record) in its buffer. Nothing is sent.

3. TP(b) issues RECEIVE_AND_WAIT, causing the LU to suspend TP(b)'s processing until it receives information.

4. TP(a) issues SEND_ERROR in order to notify the partner program of an error. The SEND_ERROR verb causes the LU to send the contents of its buffer.

5. After sending the contents of its buffer, the LU sends the error notification and returns control to the program.

6. The conversation for TP(a) is still in send state, so TP(a) issues another SEND_DATA, possibly containing additional error-recovery information.

7. The LU returns control to TP(b), indicating that the program has received the complete logical record.

8. TP(b) issues another RECEIVE_AND_WAIT and receives the error notification, PROG_ERROR_NO_TRUNC, meaning TP(a) issued a SEND_ERROR that did not truncate the logical record it sent.

9. The conversation for TP(b) is still in receive state, so TP(b) issues another RECEIVE_AND_WAIT.

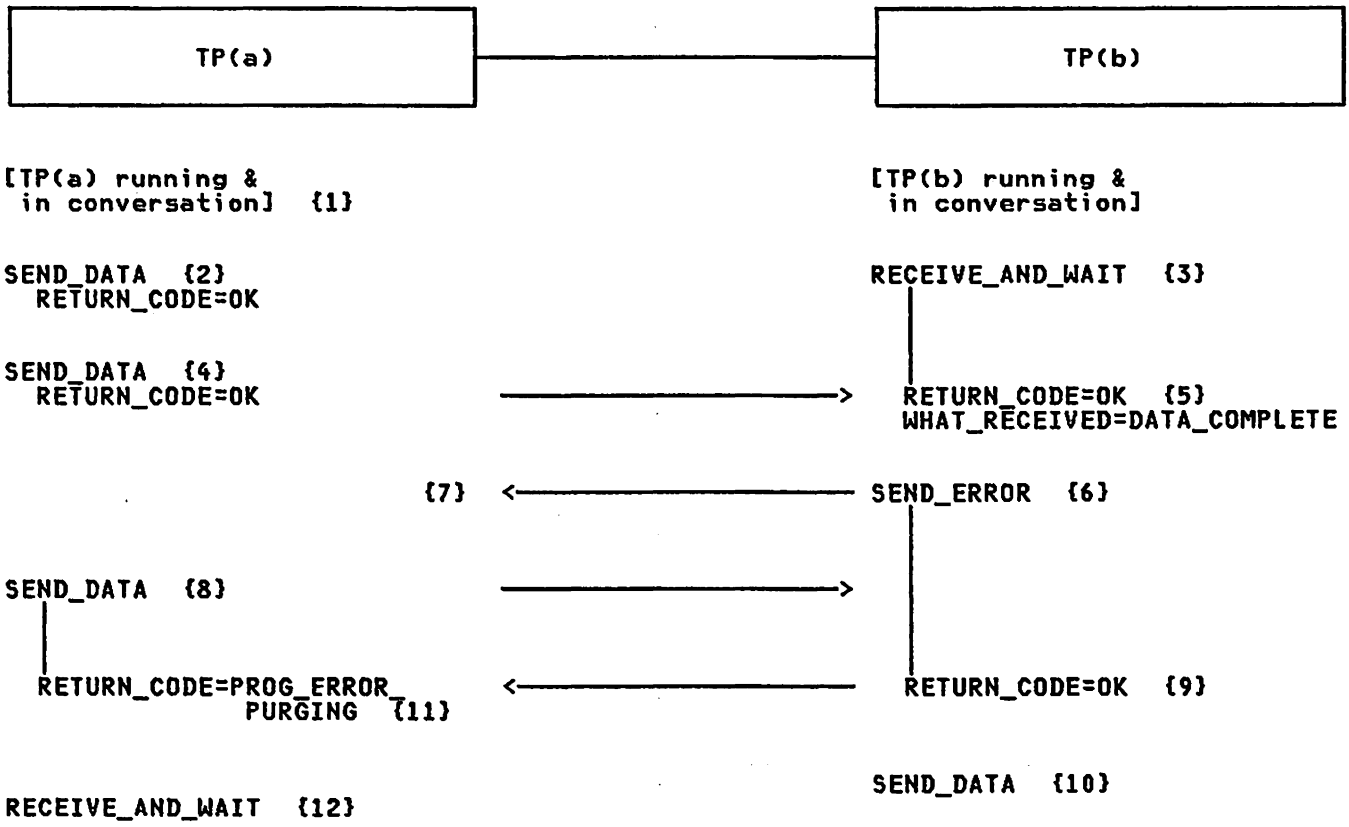


Figure B-8. SEND_ERROR in Receive State

Notes for Figure B-8:

1. Assume TP(a) has already allocated the conversation and it is now send state for TP(a) and receive state for TP(b).

2. TP(a) issues SEND_DATA, causing the LU to place the data (a logical record) in its buffer. Nothing is sent.

3. TP(b) issues RECEIVE_AND_WAIT, causing the LU to suspend TP(b)'s processing until it receives information.

4. TP(a) issues another SEND_DATA, causing the LU to place the data (another logical record) in its buffer. The LU now has more than enough data for transmission, so it sends some of the contents of its buffer, and retains the remainder for later transmission.

5. The LU returns control to TP(b), indicating that the program has received a complete logical record.

6. TP(b) issues SEND_ERROR in order to notify the partner program of an error. The SEND_ERROR verb causes the LU to purge information it has received and not yet passed to TP(b), and to send a negative response. The LU then suspends TP(b)'s processing awaiting the receipt of SEND control.

7. The LU for TP(a) receives the negative response, causing it to purge the remaining contents of its buffer.

8. TP(a) issues SEND_DATA. The SEND_DATA is unsuccessful -- the LU does not place the data in its buffer. The SEND_DATA causes the LU to send the SEND control without data. The LU then suspends TP(a)'s processing awaiting the receipt of the error notification.

9. The LU for TP(b) receives the SEND control, sends the error notification, and returns control to TP(b).

10. The conversation for TP(b) is now in send state, so TP(b) issues a SEND_DATA, possibly containing additional error-recovery information.

11. The LU returns control to TP(a), indicating that it has received the error notification, PROG_ERROR_PURGING.

12. The conversation for TP(a) is now in receive state, so TP(a) issues a RECEIVE_AND_WAIT.

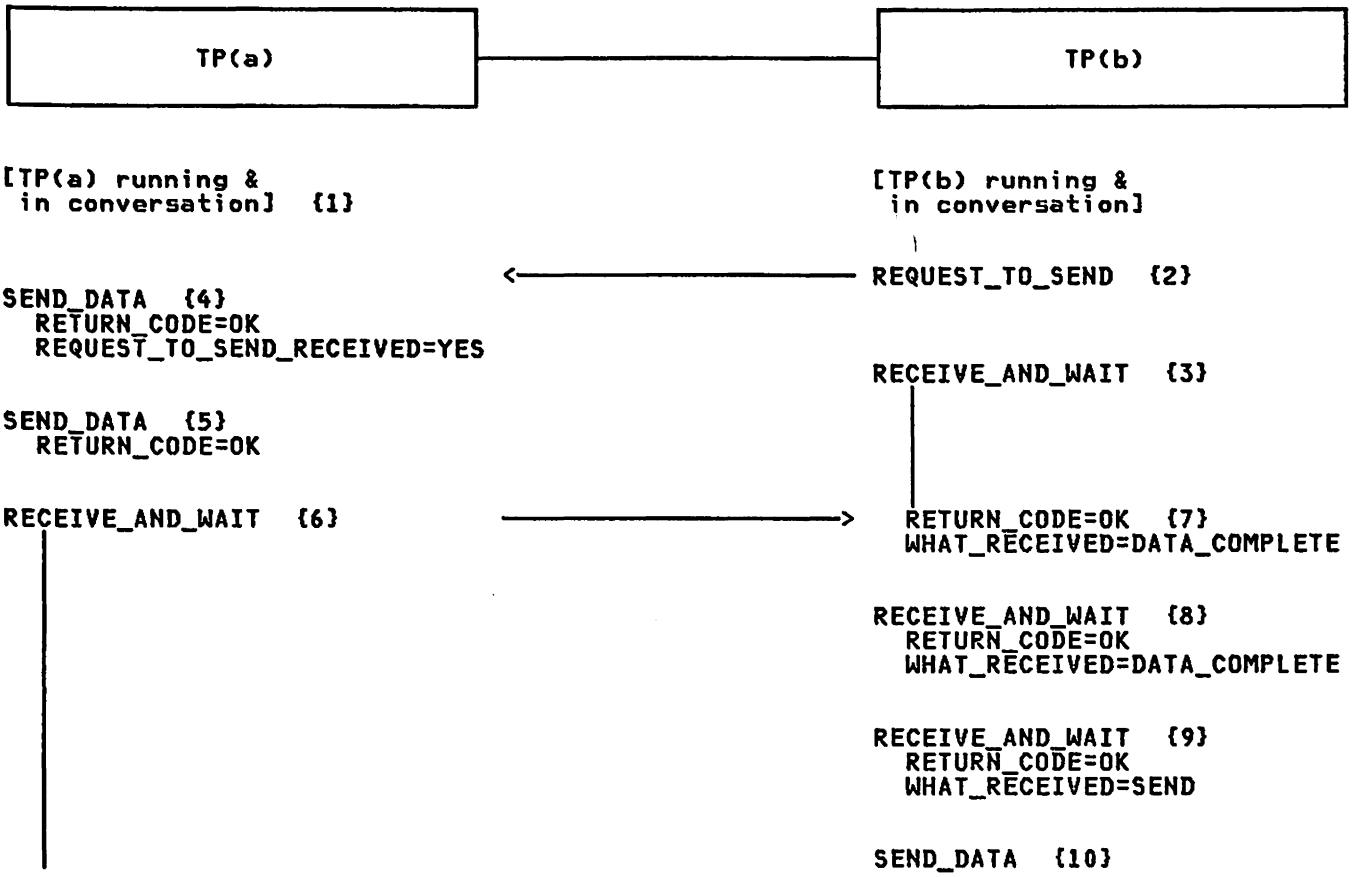


Figure B-9. REQUEST_TO_SEND

Notes for Figure B-9:

1. Assume TP(a) has already allocated the conversation and it is now send state for TP(a) and receive state for TP(b).

2. TP(b) issues REQUEST_TO_SEND and the LU sends the REQUEST_TO_SEND indication. The conversation for TP(b) remains in receive state because REQUEST_TO_SEND does not force a turnaround of SEND control. Contrast this with SEND_ERROR in Figure B-8 on page B-16.

3. TP(b) issues RECEIVE_AND_WAIT, causing the LU to suspend TP(b)'s processing until it receives information.

4. TP(a) issues SEND_DATA, causing the LU to place the data (a logical record) in its buffer, and indicate that it has received a REQUEST_TO_SEND indication. Nothing is sent.

5. TP(a) issues another SEND_DATA, causing the LU to place the data (another logical record) in its buffer. The LU still does not have enough data for transmission, so nothing is sent.

6. TP(a) issues RECEIVE_AND_WAIT, causing the LU to send the contents of its buffer together with the SEND indication. The LU suspends TP(a)'s processing until it receives information.

7. The LU returns control to TP(b), indicating that the program has received a complete logical record.

8. TP(b) issues another RECEIVE_AND_WAIT and receives another complete logical record.

9. TP(b) issues another RECEIVE_AND_WAIT and receives the SEND indication.

10. The conversation for TP(b) is now in send state, so TP(b) issues a SEND_DATA.

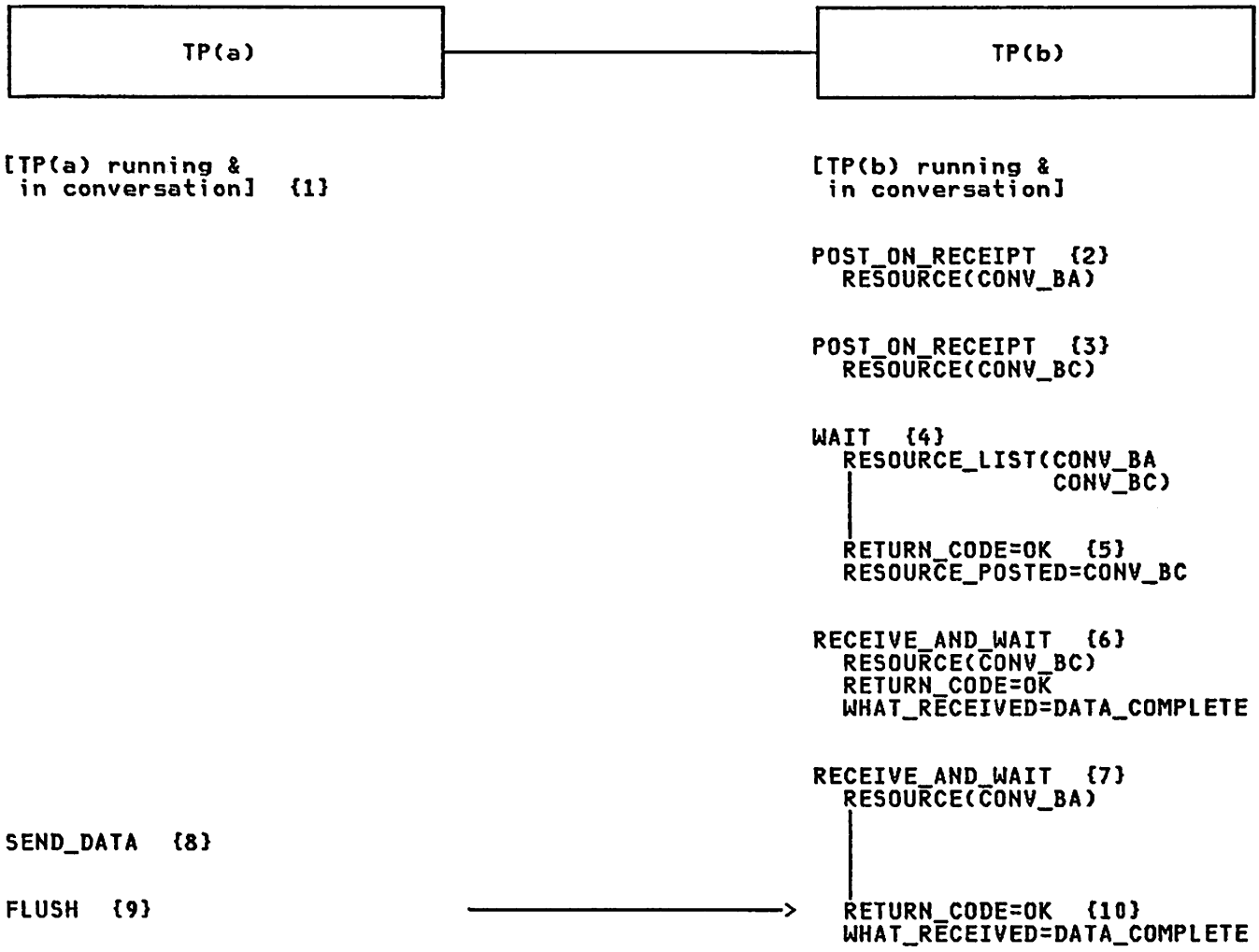


Figure B-10. POST_ON_RECEIPT, WAIT

Notes for Figure B-10:

1. Assume TP(a) has already allocated the conversation and it is now in send state for TP(a) and receive state for TP(b).

2. TP(b) issues POST_ON_RECEIPT for the conversation with TP(a).

3. TP(b) issues another POST_ON_RECEIPT for the conversation with TP(c), not shown.

4. TP(b) then issues WAIT with a resource list specifying both conversations. In this way, the program can receive the information on the conversation on which it arrives first. The WAIT causes the LU to suspend TP(b)'s processing until it receives information on either conversation.

5. Information arrives on the conversation with TP(c). The LU resets the posting on that conversation and returns control to TP(b).

6. TP(b) issues RECEIVE_AND_WAIT for the conversation with TP(c) and receives a complete logical record.

7. TP(b) then issues RECEIVE_AND_WAIT for the conversation with TP(a), causing the LU to suspend TP(b)'s processing until it receives information on that conversation.

8. TP(a) issues SEND_DATA, causing the LU to place the data (a logical record) in its buffer. Nothing is sent.

9. TP(a) issues FLUSH, causing the LU to send the contents of its buffer.

10. The LU returns control to TP(b), indicating that the program has received a complete logical record.

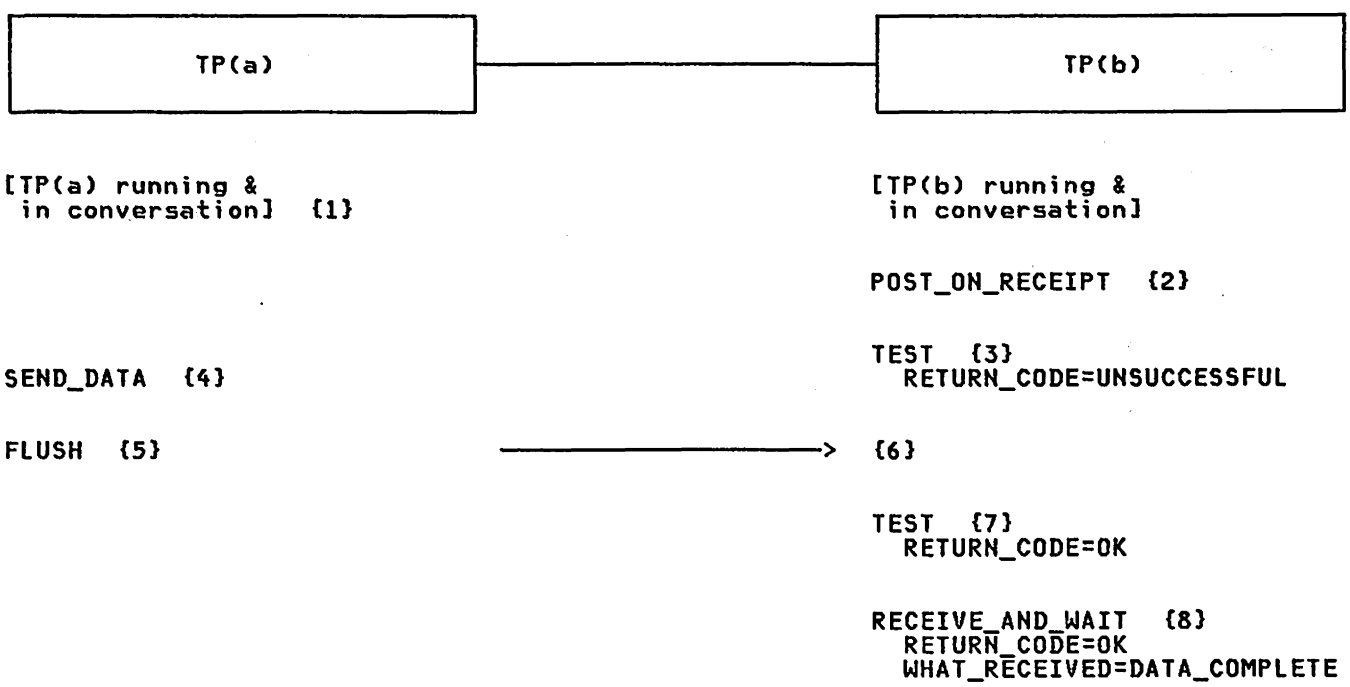


Figure B-11. POST_ON_RECEIPT, TEST

Notes for Figure B-11:

1. Assume TP(a) has already allocated the conversation and it is now in send state for TP(a) and receive state for TP(b).
2. TP(b) issues POST_ON_RECEIPT for the conversation with TP(a).
3. TP(b) then issues TEST, which returns with RETURN_CODE=UNSUCCESSFUL indicating information is not available. Posting remains active and TP(b) continues processing.
4. TP(a) issues SEND_DATA, causing the LU to place the data (a logical record) in its buffer. Nothing is sent.
5. TP(a) issues FLUSH, causing the LU to send the contents of its buffer.
6. Data arrives on the conversation, causing the LU to post the event.
7. Some time later, TP(b) issues TEST again. This time it returns with RETURN_CODE=OK, and posting is reset.
8. TP(b) issues RECEIVE_AND_WAIT for the conversation and receives a complete logical record.

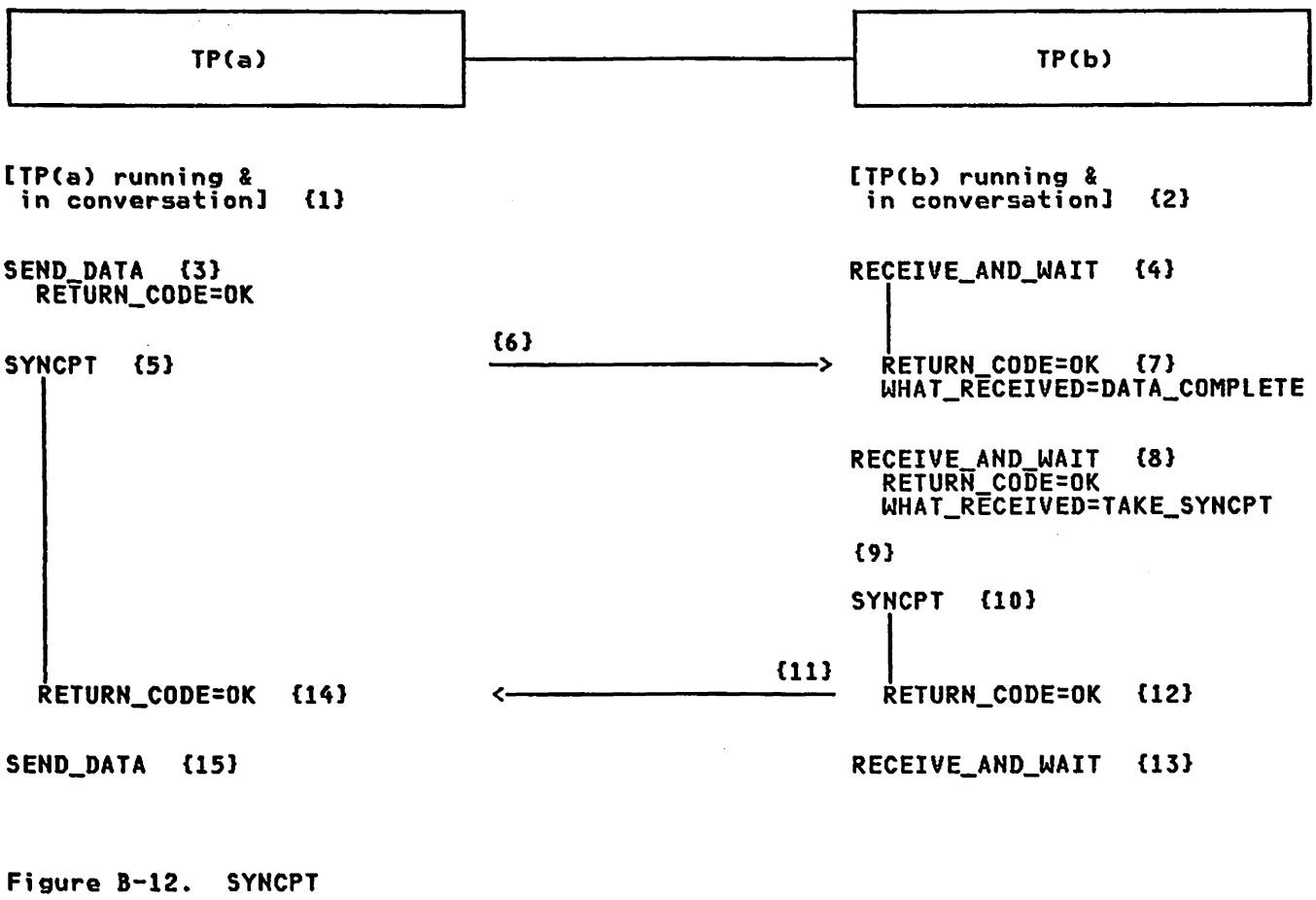


Figure B-12. SYNCPT

Notes for Figure B-12:

1. Assume TP(a) has already allocated the conversation with SYNC_LEVEL(SYNCPT), and the conversation is now in send state for TP(a) and receive state for TP(b).

2. Assume TP(b) has also allocated one or more conversations with SYNC_LEVEL(SYNCPT) to other programs.

3. TP(a) issues SEND_DATA, causing the LU to place the data (a logical record) in its buffer. Nothing is sent.

4. TP(b) issues RECEIVE_AND_WAIT, causing the LU to suspend TP(b)'s processing until it receives information.

5. TP(a) issues SYNCPT in order to advance all protected resources throughout the distributed logical unit of work to the next synchronization point. The LU suspends TP(a)'s processing until the sync point processing is complete. As part of the sync point processing, the LUs send and receive commands on the conversations; the commands are referred to in this example as a sync point request and reply. These commands are not apparent to the programs.

6. The SYNCPT verb causes the LU to send the contents of its buffer together with the initial sync point request.

7. The LU for TP(b) receives the data and sync point request. The LU returns control to TP(b), indicating that the program has received a complete logical record.

8. TP(b) issues another RECEIVE_AND_WAIT and receives the TAKE_SYNCPT request, which is what the LU indicates to the program as a result of receiving the sync point request.

9. TP(b) finishes processing of protected local resources, if necessary, and ensures all other protected conversations are in send state.

10. TP(b) issues SYNCPT, causing the LU to send the contents of its buffers (one for each conversation) together with a sync point request on all other protected conversations. The LU suspends TP(b)'s processing until a sync point reply is received on all these conversations.

11. After receiving sync point replies on all of the other protected conversations, the LU for TP(b) sends a sync point reply on the conversation on which it received the initial sync point request.

12. The LU returns control to TP(b) indicating successful completion of the SYNCPT for all protected resources allocated to TP(b) and all "down stream" TPs, that is, to all TPs other than TP(a).

13. The conversation for TP(b) is still in receive state, so TP(b) issues another RECEIVE_AND_WAIT.

14. The LU for TP(a) receives the final sync point reply and returns control to TP(a) indicating successful completion of the SYNCPT for all protected resources throughout the distributed logical unit of work.

15. The conversation for TP(a) is still in send state, so TP(a) issues another SEND_DATA.

Note: More sync point commands may actually be exchanged between the participating LUs than the flows in this example indicate. See SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2 for details.

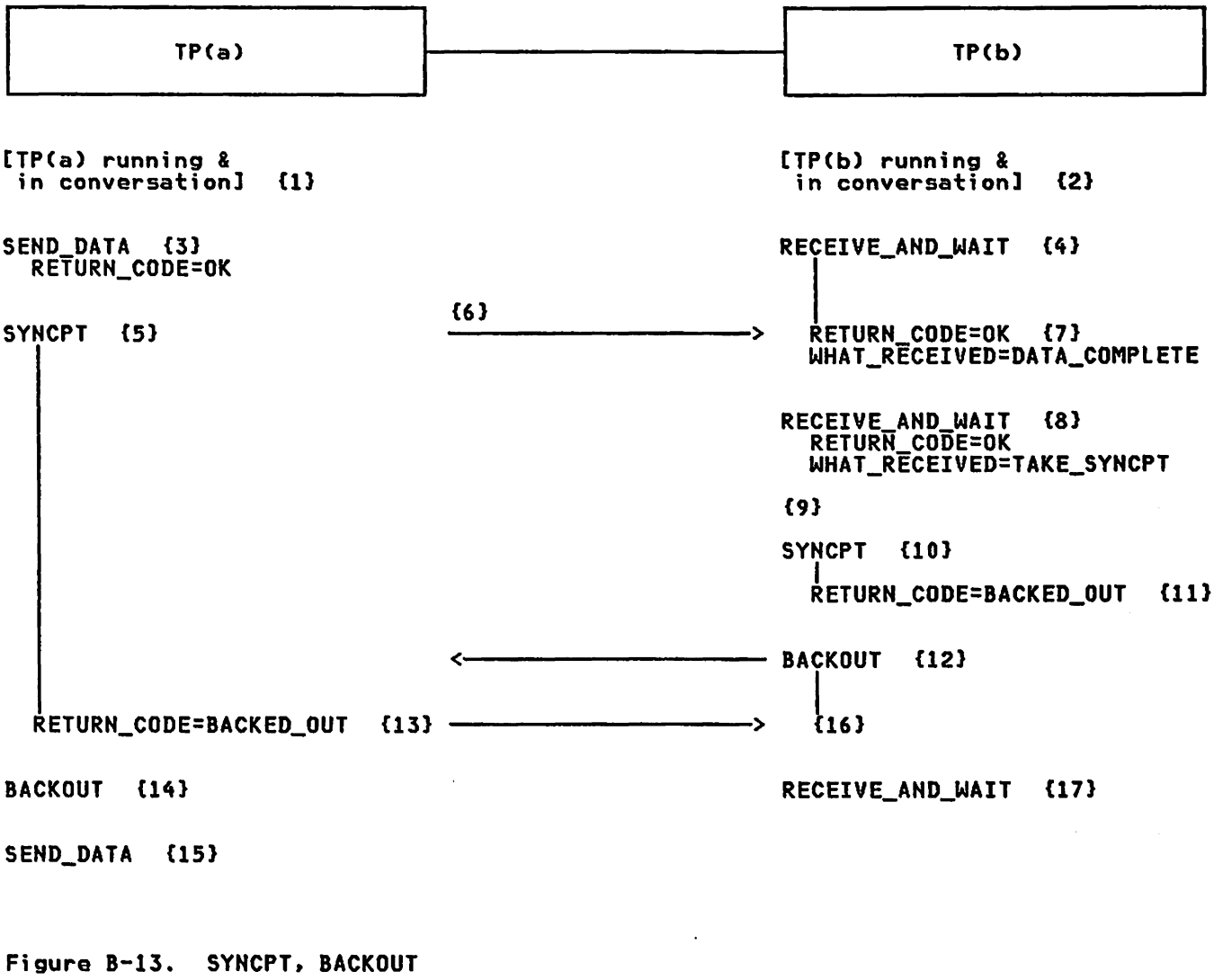


Figure B-13. SYNCPT, BACKOUT

Notes for Figure B-13:

1. Assume TP(a) has already allocated the conversation with SYNC_LEVEL(SYNCPT), and the conversation is now in send state for TP(a) and receive state for TP(b).

2. Assume TP(b) has also allocated one or more conversations with SYNC_LEVEL(SYNCPT) to other programs.

3. TP(a) issues SEND_DATA, causing the LU to place the data (a logical record) in its buffer. Nothing is sent.

4. TP(b) issues RECEIVE_AND_WAIT, causing the LU to suspend TP(b)'s processing until it receives information.

5. TP(a) issues SYNCPT in order to advance all protected resources throughout the distributed logical unit of work to the next synchronization point. The LU suspends TP(a)'s processing until the sync point processing is complete. As part of the sync point processing, the LUs send and receive commands on the conversations; the commands are referred to in this example as a sync point request and reply. These commands are not apparent to the programs.

6. The SYNCPT verb causes the LU to send the contents of its buffer together with the initial sync point request.

7. The LU for TP(b) receives the data and sync point request. The LU returns control to TP(b), indicating that the program has received a complete logical record.

8. TP(b) issues another RECEIVE_AND_WAIT and receives the TAKE_SYNCPT request, which is what the LU indicates to the program as a result of receiving the sync point request.

9. TP(b) finishes processing of protected local resources, if necessary, and ensures all subordinate conversations are in send state.

10. TP(b) issues SYNCPT, causing the LU to send the contents of its buffers (one for each conversation) together with a sync point request on all other protected conversations. The LU suspends TP(b)'s processing until the sync point processing on all these conversations is complete.

11. Instead of receiving sync point replies on all of the other protected conversations, the LU for TP(b) receives at least one BACKED_OUT indication. The LU returns control to TP(b) with the BACKED_OUT indication.

12. TP(b) issues BACKOUT, causing the LU to restore all protected local resources to the last synchronization point, and send BACKED_OUT indications on all protected conversations except the one(s) on which it received the preceding BACKED_OUT indication. BACKOUT is accomplished throughout the distributed logical unit of work in the same manner as for TP(b). That is, the LU for each program receives a BACKED_OUT indication, returns control to its program with the BACKED_OUT indication, and then when the program issues BACKOUT it restores all protected local resources to the last synchronization point and sends BACKED_OUT indications on all remaining protected conversations, if any.

13. The LU for TP(a) receives the BACKED_OUT indication, sends back a positive response, and returns control to TP(a) with the BACKED_OUT indication.

14. TP(a) issues BACKOUT, causing the LU to restore all protected local resources to the last synchronization point.

15. The conversation for TP(a) is restored to send state—the state at the completion of the last synchronization point—so TP(a) issues another SEND_DATA, possibly containing error-recovery information.

16. The LU for TP(b) receives the positive response from the LU for TP(a) and all other LUs to which it sent the BACKED_OUT indication, and returns control to TP(b).

17. The conversation for TP(b) is restored to receive state—the state at the completion of the last synchronization point—so TP(b) issues another RECEIVE_AND_WAIT.

This page intentionally left blank

APPENDIX C. SYMBOL STRING CONVENTIONS

This manual refers to the following symbol strings:

Network ID
LU Name
Fully-Qualified LU Name
Mode Name
Transaction Program Name
SECURITY Subfields
PIP Subfields
Map Name

This appendix defines the type and length of these symbol strings. The meanings of these symbol strings are defined in the chapters describing the individual verbs that refer to these symbol strings. The type and length of each symbol string is defined in terms of the send and receive support of all LU 6.2 products that implement the symbol string.

SYMBOL STRING TYPE

The symbol-string type identifies the set of characters from which the symbol string can be composed, and therefore the characters a transaction program can use to specify the symbol string. The following symbol-string types are defined:

- Type A (Assembler oriented): a character string consisting of one or more EBCDIC uppercase letters A through Z; numerics 0 through 9; and special characters $, #, and @; the first character of which is an uppercase letter or a special character.
- Type AE (A extended): a character string consisting of one or more EBCDIC lowercase letters a through z; uppercase letters A through Z; numerics 0 through 9; special characters $, #, @; and the period (.); with no restriction on the first character.
- Type GR (EBCDIC graphics): a character string consisting of one or more EBCDIC characters in the range hex 41 through hex FE with no restriction on the first character.
- Type DB (double byte): a byte string consisting of an even number of four or more bytes beginning with a byte of hex 0E, followed by bytes in the range hex 41 through hex FE, and ending with a byte of hex 0F.
- Type G (general): a byte string consisting of one or more bytes in the range hex 00 through hex FF, with no restriction on the first byte.

The set of type-A and type-AE characters, and the hex codes for these characters, are shown in Figure C-1 on page C-2.

Hex Code	Graphic	Description	Set		Hex Code	Graphic	Description	Set	
			A	AE				A	AE
4B	.	Period		X	C4	D	D, Capital	X	X
5B	$	Dollar Sign	X	X	C5	E	E, Capital	X	X
7B	#	Number Sign	X	X	C6	F	F, Capital	X	X
7C	@	At Sign	X	X	C7	G	G, Capital	X	X
81	a	a, Small		X	C8	H	H, Capital	X	X
82	b	b, Small		X	C9	I	I, Capital	X	X
83	c	c, Small		X	D1	J	J, Capital	X	X
84	d	d, Small		X	D2	K	K, Capital	X	X
85	e	e, Small		X	D3	L	L, Capital	X	X
86	f	f, Small		X	D4	M	M, Capital	X	X
87	g	g, Small		X	D5	N	N, Capital	X	X
88	h	h, Small		X	D6	O	O, Capital	X	X
89	i	i, Small		X	D7	P	P, Capital	X	X
91	j	j, Small		X	D8	Q	Q, Capital	X	X
92	k	k, Small		X	D9	R	R, Capital	X	X
93	l	l, Small		X	E2	S	S, Capital	X	X
94	m	m, Small		X	E3	T	T, Capital	X	X
95	n	n, Small		X	E4	U	U, Capital	X	X
96	o	o, Small		X	E5	V	V, Capital	X	X
97	p	p, Small		X	E6	W	W, Capital	X	X
98	q	q, Small		X	E7	X	X, Capital	X	X
99	r	r, Small		X	E8	Y	Y, Capital	X	X
A2	s	s, Small		X	E9	Z	Z, Capital	X	X
A3	t	t, Small		X	F0	0	Zero	X	X
A4	u	u, Small		X	F1	1	One	X	X
A5	v	v, Small		X	F2	2	Two	X	X
A6	w	w, Small		X	F3	3	Three	X	X
A7	x	x, Small		X	F4	4	Four	X	X
A8	y	y, Small		X	F5	5	Five	X	X
A9	z	z, Small		X	F6	6	Six	X	X
C1	A	A, Capital	X	X	F7	7	Seven	X	X
C2	B	B, Capital	X	X	F8	8	Eight	X	X
C3	C	C, Capital	X	X	F9	9	Nine	X	X

Figure C-1. Character Sets A and AE

Figure C-2 on page C-3 defines the product send support and receive support for each symbol string in terms of the symbol-string types. Depending on the symbol string, product send support or receive support is indicated either by a single type or multiple types. Where multiple types are indicated, the type selected is product-defined and send support may differ from receive support.

Symbol String	Type	
	Send Support	Receive Support
Network ID	A	A
LU Name [1]	–	–
Fully-Qualified LU Name [2]	A.A	A.A
Mode Name	A	A
Transaction Program Name [3]	AE, GR, or DB	A, AE, GR, or DB
LU-LU Password [4]	–	–
SECURITY Subfields	AE, GR, or DB	A, AE, GR, or DB
PIP Subfields	G	G
Map Name	A, AE, or GR	A, AE, or GR

Notes:

1. The LU name is a locally-known name; it is the name by which one LU knows another LU. A transaction program specifies this LU name in conjunction with the mode name when it allocates a session for a conversation. This LU name is not sent outside the LU. The symbol-string type is G.

2. The fully-qualified LU name consists of two symbol strings of type A concatenated by a period (".."). The lefthand symbol string represents the network ID; the righthand symbol string represents the network LU name. The period is not part of the network ID or the network LU name.

3. The first character of an SNA service transaction program name is a character ranging in value from hex 00 through hex 0D and hex 10 through hex 3F (excluding hex 0E and hex 0F). More details about SNA service transaction program names and a list of SNA service transaction programs is given in "Appendix D. List of SNA Service Transaction Programs".

4. The LU-LU password is a locally-specified value and is not sent outside the LU. The symbol-string type is G.

Figure C-2. Symbol-String Types

SYMBOL STRING LENGTH

The symbol-string length represents the number of characters a symbol string can contain. Three symbol-string lengths are defined:

- Minimum specification length: the minimum number of characters that a transaction program is allowed to use to specify the symbol string. For some symbol strings, the minimum specification length is zero. Zero-length strings are valid symbol strings and are subject to the same usage conditions as valid non-zero length strings.[1]

- Maximum send support: the maximum number of characters that all products can send for the symbol string.

- Maximum receive support: the maximum number of characters that all products can receive for the symbol string.

The maximum send or receive support for a symbol string's length is defined either by a single value or within a range of values, depending on the symbol string.

The single value is the maximum number of characters in a symbol string that all products can send or receive.

The range of values represents a lower and upper bound of the maximum number of characters in a symbol string that a product can send or receive. The specific maximum number of characters a product can send or receive for each of these symbol strings is product-defined within the range.

Figure C-3 on page C-5 defines the product maximum send and receive support for each symbol string in terms of the symbol-string lengths. Where support is defined to be within a range of values, the range is given as "lower-value<->upper-value," which identifies the lower and upper bounds of the range.

Note: The variable to which a type-A, type-AE, type-GR, or type-DB symbol string is assigned may be longer than the symbol string; in this case, the symbol string is left-justified within the variable and the variable is filled out to the right with space (hex 40) characters. Space characters, if present, are not part of the symbol string. If the symbol string is formed from the concatenation of two or more individual symbol strings, such as the fully-qualified LU name, the concatenated symbol string as a whole is left-justified within the variable and the variable is filled out to the right with space characters. Space characters, if present, are not part of the concatenated symbol string.

---

[1] A valid symbol string is one that meets the requirements of the symbol-string type defined for that symbol string.

Symbol String	Length		
	Minimum Specification	Maximum Send Support	Maximum Receive Support
Network ID	0	8	8
LU Name [1]	0	–	–
Fully-Qualified LU Name	1	17	17
Mode Name	0	8	8
Transaction Program Name	1	8<–>64	8<–>64
LU-LU Password [2]	1	–	–
SECURITY Subfields	0	8<–>10	0<–>10
PIP Subfields [3]	0	64<–>* [4]	64<–>* [5]
Map Name	1	8<–>64	7<–>64

Notes:

1. The LU name is a locally-known name; it is the name by which one LU knows another LU. A transaction program specifies this LU name in conjunction with the mode name when it allocates a session for a conversation. This LU name is not sent outside the LU. The maximum specification length of the LU name is product-defined.

2. The LU-LU password is a locally-specified value and is not sent outside the LU. It is 64 bits (8 bytes) in length. At least 8 bits (1 byte) must be specified. If less than 64 bits are specified, the local LU fills the password out to the right with binary 0's.

3. Product support of PIP subfields is optional, and send support is independent of receive support. The maximum number of PIP subfields a product can send or receive is product-defined; it can be any number greater than or equal to 7.

4. The maximum send support for PIP subfields is product-defined; it can be any length greater than or equal to 64.

5. The maximum receive support for PIP subfields is product-defined; it can be any length greater than or equal to 64.

Figure C-3. Symbol-String Lengths

This page intentionally left blank

APPENDIX D. LIST OF SNA SERVICE TRANSACTION PROGRAMS

This appendix lists the classes of SNA service transaction programs. The SNA service transaction programs are categorized according to functional classes. A class is identified by the first one or two characters of the name. All SNA service transaction programs belonging to a given class have the same class identifier.

The SNA service transaction program classes are:

Class	Identifier (in hex)
Scheduler	02
Queue	03
DL/1	05
Change Number of Sessions	06F1
Resynchronization	06F2
Distributed Data Management	07F0
Document Interchange Architecture	20F0
SNA Distribution Services	21F0
Product Oriented	30F0

SNA SERVICE TRANSACTION PROGRAM NAMES

SNA service transaction programs are distinguished by their names, in particular the first (leftmost) character. The name of an SNA service transaction program can be one to four characters in length; typically, however, they are four characters in length. The first character of the name can range in value from hex 00 through hex 0D and hex 10 through hex 3F (excluding hex 0E and hex 0F). The remaining characters of the name are type-A, without any restriction on the first type-A character. By contrast, names of programs other than SNA service transaction programs are type-A, type-AE, type-GR, or type-DB symbol strings.

Using the first character of the name, a product may restrict the right of access of its programs to SNA service transaction programs. For example, a product may allow only its "privileged" programs to allocate conversations to SNA service transaction programs, and permit its "nonprivileged" programs to allocate conversations only to transaction programs other than SNA service transaction programs.

Listed below are the individual SNA service transaction programs, grouped by class.

SCHEDULER

Name (in hex)	Description
02	LU 6.1 scheduler transaction program.

See the supporting IBM product's publications for more details.

QUEUE

Name (in hex)	Description
03	LU 6.1 queue transaction program.

See the supporting IBM product's publications for more details.

DL/1

Name (in hex)	Description
05	LU 6.1 DL/1 transaction program.

See the supporting IBM product's publications for more details.

CHANGE NUMBER OF SESSIONS

Name (in hex)	Description
06F1	CNOS service transaction program

See SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2 for more details.

RESYNCHRONIZATION

Name (in hex)	Description
06F2	Sync point resynchronization transaction program.

See SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2 and the supporting IBM product's publications for more details.

DISTRIBUTED DATA MANAGEMENT

Name (in hex)	Description
07F0F0F1	Distributed data management synchronous conversation transaction program.

See the supporting IBM product's publications for more details.

DOCUMENT INTERCHANGE ARCHITECTURE

Name (in hex)	Description
20F0F0F0	DIA process transaction program.
20F0F0F1	DIA file server transaction program.

See Document Interchange Architecture: Technical Reference for more details.

SNA DISTRIBUTION SERVICES

Name (in hex)	Description
21F0F0F1	SNADS DS_SEND transaction program.
21F0F0F2	SNADS DS_RECEIVE transaction program.
21F0F0F3	SNADS DS_ROUTER_DIRECTOR transaction program.
21F0F0F6	SNADS general server transaction program.

See SNA Format and Protocol Reference Manual: Distribution Services for more details.

PRODUCT ORIENTED

The following SNA service transaction programs are provided by specific products.

Name (in hex)	Description
30F0F0F0	Printer CPDS transaction program for IBM 3820.
30F0F0F1	Printer level 2 transaction program for IBM 3820.

30F0F0F3	Object distribution transaction program for IBM System/38.
30F0F0F4	NETDATA server transaction program for IBM System/38.
30F0F0F5	IBM 5250 device passthrough transaction program for IBM System/36 and IBM System/38.
30F0F0F6	Virtual disk transaction program for IBM System/36 and IBM System/38.
30F0F0F7	Virtual printer transaction program for IBM System/36 and IBM System/38.

See the IBM product's publications for more details.

This page intentionally left blank

APPENDIX E. CONVERSATION STATE MATRICES

This appendix shows the conversation state transitions that can occur when a program issues a basic or type-independent conversation verb. A state transition can occur as a result of a verb the local program issues, a verb the remote program issued, or a network error; the latter two are indicated when a return code of other than OK is returned to the local program.

The conversation state transitions are represented by means of a matrix; see Figure E-1, Figure E-1, and Figure E-3. The columns of the matrix show the individual states, and the rows show the individual verbs. A verb is shown more than once when a parameter of the verb or the return code determines the state transitions that can occur.

Following the state-transition matrix is a matrix showing the state-check ABEND conditions that can occur; see Figure E-4. A state check occurs when the program attempts to issue a verb for a conversation that is in a state in which the verb is not allowed.

The conversation states are:

Reset — the state in which the program can allocate the conversation.

Send — the state in which the program can send data, request confirmation, or request sync point.

Defer Receive and Defer Deallocate — the states in which the program can request sync point or confirmation, or simply flush the LU's send buffer, when the synchronization level is SYNCPT.

Receive — the state in which the program can receive information from the remote program.

Confirm, Confirm Send, and Confirm Deallocate — the states in which the program can reply to a confirmation request.

Sync-point, Sync-point Send, and Sync-point Deallocate — the states in which the program can respond to a sync point request.

Deallocate — the state in which the program can deallocate the conversation locally.

Abbreviations are used for the parameters, return codes, and what-received indications. The abbreviations and symbols used in the state-transition matrix are defined at the bottom of Figure E-3. Abbreviations and symbols used in the state-check matrix are defined at the bottom of Figure E-4.

Verb	Conversation States					
	Reset	Send	Defer Receive	Defer Deallo-cate	Receive	Confirm
	1	2	3	4	5	6
Initiating Conversation	5	/	/	/	/	/
ALLOCATE[ok]	2	/	/	/	/	/
ALLOCATE[ae]	12	/	/	/	/	/
ALLOCATE[pe]	-	/	/	/	/	/
ALLOCATE[un]	-	/	/	/	/	/
BACKOUT	/	*	*	*	*	*
CONFIRM[ok]	/	-	5	1	/	/
CONFIRM[ae]	/	12	12	12	/	/
CONFIRM[bo]	/	*	*	*	/	/
CONFIRM[da]	/	12	12	12	/	/
CONFIRM[ep]	/	5	5	5	/	/
CONFIRM[rf]	/	12	12	12	/	/
CONFIRMED	/	/	/	/	/	5
DEALLOCATE(F)[ok]	/	1	/	/	/	/
DEALLOCATE(C)[ok]	/	1	/	/	/	/
DEALLOCATE(S)[ok]	/	4	/	/	/	/
DEALLOCATE(A)[ok]	/	1	1	1	1	1
DEALLOCATE(L)[ok]	/	/	/	/	/	/
DEALLOCATE(C)[ae]	/	12	/	/	/	/
DEALLOCATE(C)[da]	/	12	/	/	/	/
DEALLOCATE(C)[ep]	/	5	/	/	/	/
DEALLOCATE(C)[rf]	/	12	/	/	/	/
FLUSH	/	-	5	1	/	/
GET_ATTRIBUTES	/	-	-	-	-	-
GET_TYPE	/	-	-	-	-	-
POST_ON_RECEIPT	/	/	/	/	-	/
PREPARE_TO_RECEIVE(F)[ok]	/	5	/	/	/	/
PREPARE_TO_RECEIVE(C)[ok]	/	5	/	/	/	/
PREPARE_TO_RECEIVE(S)[ok]	/	3	/	/	/	/
PREPARE_TO_RECEIVE(C)[ae]	/	12	/	/	/	/
PREPARE_TO_RECEIVE(C)[da]	/	12	/	/	/	/
PREPARE_TO_RECEIVE(C)[ep]	/	5	/	/	/	/
PREPARE_TO_RECEIVE(C)[rf]	/	12	/	/	/	/
RECEIVE_AND_WAIT[ok]{da}	/	5	/	/	-	/
RECEIVE_AND_WAIT[ok]{se}	/	-	/	/	2	/
RECEIVE_AND_WAIT[ok]{co}	/	6	/	/	6	/
RECEIVE_AND_WAIT[ok]{cs}	/	7	/	/	7	/
RECEIVE_AND_WAIT[ok]{cd}	/	8	/	/	8	/
RECEIVE_AND_WAIT[ok]{sy}	/	9	/	/	9	/
RECEIVE_AND_WAIT[ok]{ss}	/	10	/	/	10	/
RECEIVE_AND_WAIT[ok]{sd}	/	11	/	/	11	/
RECEIVE_AND_WAIT[ae]	/	12	/	/	12	/
RECEIVE_AND_WAIT[bo]	/	*	/	/	*	/
RECEIVE_AND_WAIT[da]	/	12	/	/	12	/
RECEIVE_AND_WAIT[dn]	/	12	/	/	12	/
RECEIVE_AND_WAIT[en]	/	5	/	/	-	/
RECEIVE_AND_WAIT[ep]	/	5	/	/	-	/
RECEIVE_AND_WAIT[et]	/	/	/	/	-	/
RECEIVE_AND_WAIT[rf]	/	12	/	/	12	/

Figure E-1. Conversation State Transition Matrix (Part 1 of 3)

Conversation States (continued)						Verb
Confirm Send 7	Confirm Deallo-cate 8	Sync-point 9	Sync-point Send 10	Sync-point Deallo-cate 11	Deallo-cate 12	
/ / / / /	/ / / / /	/ / / / /	/ / / / /	/ / / / /	/ / / / /	Initiating Conversation ALLOCATE[ok] ALLOCATE[ae] ALLOCATE[pe] ALLOCATE[un]
* / / / /	* / / / /	* / / / /	* / / / /	* / / / /	/ / / / /	BACKOUT CONFIRM[ok] CONFIRM[ae] CONFIRM[bo] CONFIRM[da]
/ / 2 / /	/ / 12 / /	/ / / / /	/ / / / /	/ / / / /	/ / / / /	CONFIRM[ep] CONFIRM[rf] CONFIRMED DEALLOCATE(F)[ok] DEALLOCATE(C)[ok]
/ 1 / / /	/ 1 / / /	/ 1 / / /	/ 1 / / /	/ 1 / / /	/ / 1 / /	DEALLOCATE(S)[ok] DEALLOCATE(A)[ok] DEALLOCATE(L)[ok] DEALLOCATE(C)[ae] DEALLOCATE(C)[da]
/ / / - -	/ / / - -	/ / / - -	/ / / - -	/ / / - -	/ / / - -	DEALLOCATE(C)[ep] DEALLOCATE(C)[rf] FLUSH GET_ATTRIBUTES GET_TYPE
/ / / / /	/ / / / /	/ / / / /	/ / / / /	/ / / / /	/ / / / /	POST_ON_RECEIPT PREPARE_TO_RECEIVE(F)[ok] PREPARE_TO_RECEIVE(C)[ok] PREPARE_TO_RECEIVE(S)[ok] PREPARE_TO_RECEIVE(C)[ae]
/ / / / /	/ / / / /	/ / / / /	/ / / / /	/ / / / /	/ / / / /	PREPARE_TO_RECEIVE(C)[da] PREPARE_TO_RECEIVE(C)[ep] PREPARE_TO_RECEIVE(C)[rf] RECEIVE_AND_WAIT[ok]{da} RECEIVE_AND_WAIT[ok]{se}
/ / / / /	/ / / / /	/ / / / /	/ / / / /	/ / / / /	/ / / / /	RECEIVE_AND_WAIT[ok]{co} RECEIVE_AND_WAIT[ok]{cs} RECEIVE_AND_WAIT[ok]{cd} RECEIVE_AND_WAIT[ok]{sy} RECEIVE_AND_WAIT[ok]{ss}
/ / / / /	/ / / / /	/ / / / /	/ / / / /	/ / / / /	/ / / / /	RECEIVE_AND_WAIT[ok]{sd} RECEIVE_AND_WAIT[ae] RECEIVE_AND_WAIT[bo] RECEIVE_AND_WAIT[da] RECEIVE_AND_WAIT[dn]
/ / / /	/ / / /	/ / / /	/ / / /	/ / / /	/ / / /	RECEIVE_AND_WAIT[en] RECEIVE_AND_WAIT[ep] RECEIVE_AND_WAIT[et] RECEIVE_AND_WAIT[rf]

Verb	Conversation States					
	Reset	Send	Defer Receive	Defer Deallo-cate	Receive	Confirm
	1	2	3	4	5	6
RECEIVE_IMMEDIATE[ok]{da}	/	/	/	/	-	/
RECEIVE_IMMEDIATE[ok]{se}	/	/	/	/	2	/
RECEIVE_IMMEDIATE[ok]{co}	/	/	/	/	6	/
RECEIVE_IMMEDIATE[ok]{cs}	/	/	/	/	7	/
RECEIVE_IMMEDIATE[ok]{cd}	/	/	/	/	8	/
RECEIVE_IMMEDIATE[ok]{sy}	/	/	/	/	9	/
RECEIVE_IMMEDIATE[ok]{ss}	/	/	/	/	10	/
RECEIVE_IMMEDIATE[ok]{sd}	/	/	/	/	11	/
RECEIVE_IMMEDIATE[ae]	/	/	/	/	12	/
RECEIVE_IMMEDIATE[bo]	/	/	/	/	*	/
RECEIVE_IMMEDIATE[da]	/	/	/	/	12	/
RECEIVE_IMMEDIATE[dn]	/	/	/	/	12	/
RECEIVE_IMMEDIATE[en]	/	/	/	/	-	/
RECEIVE_IMMEDIATE[ep]	/	/	/	/	-	/
RECEIVE_IMMEDIATE[et]	/	/	/	/	-	/
RECEIVE_IMMEDIATE[rf]	/	/	/	/	12	/
RECEIVE_IMMEDIATE[un]	/	/	/	/	-	/
REQUEST_TO_SEND	/	/	/	/	-	-
SEND_DATA[ok]	/	-	/	/	/	/
SEND_DATA[ae]	/	12	/	/	/	/
SEND_DATA[bo]	/	*	/	/	/	/
SEND_DATA[da]	/	12	/	/	/	/
SEND_DATA[ep]	/	5	/	/	/	/
SEND_DATA[rf]	/	12	/	/	/	/
SEND_ERROR[ok]	/	-	/	/	2	2
SEND_ERROR[ae]	/	12	/	/	/	/
SEND_ERROR[bo]	/	*	/	/	/	/
SEND_ERROR[da]	/	12	/	/	/	/
SEND_ERROR[dn]	/	/	/	/	12	/
SEND_ERROR[ep]	/	5	/	/	/	/
SEND_ERROR[rf]	/	12	/	/	12	12
SYNCPT[ok]	/	-	5	1	/	/
SYNCPT[bo]	/	*	*	*	/	/
SYNCPT[hm]	/	-	5	1	/	/
TEST(P)[ok]	/	/	/	/	-	/
TEST(P)[ae]	/	/	/	/	12	/
TEST(P)[bo]	/	/	/	/	*	/
TEST(P)[da]	/	/	/	/	12	/
TEST(P)[dn]	/	/	/	/	12	/
TEST(P)[en]	/	/	/	/	-	/
TEST(P)[ep]	/	/	/	/	-	/
TEST(P)[et]	/	/	/	/	-	/
TEST(P)[pn]	/	/	/	/	-	/
TEST(P)[rf]	/	/	/	/	12	/
TEST(P)[un]	/	/	/	/	-	/
TEST(Q)[ok]	/	-	-	-	-	/
TEST(Q)[un]	/	-	-	-	-	/

Figure E-2. Conversation State Transition Matrix (Part 2 of 3)

Conversation States (continued)						Verb
Confirm Send 7	Confirm Deallo-cate 8	Sync-point 9	Sync-point Send 10	Sync-point Deallo-cate 11	Deallo-cate 12	
/	/	/	/	/	/	RECEIVE_IMMEDIATE[ok]{da}
/	/	/	/	/	/	RECEIVE_IMMEDIATE[ok]{se}
/	/	/	/	/	/	RECEIVE_IMMEDIATE[ok]{co}
/	/	/	/	/	/	RECEIVE_IMMEDIATE[ok]{cs}
/	/	/	/	/	/	RECEIVE_IMMEDIATE[ok]{cd}
/	/	/	/	/	/	RECEIVE_IMMEDIATE[ok]{sy}
/	/	/	/	/	/	RECEIVE_IMMEDIATE[ok]{ss}
/	/	/	/	/	/	RECEIVE_IMMEDIATE[ok]{sd}
/	/	/	/	/	/	RECEIVE_IMMEDIATE[ae]
/	/	/	/	/	/	RECEIVE_IMMEDIATE[bo]
/	/	/	/	/	/	RECEIVE_IMMEDIATE[da]
/	/	/	/	/	/	RECEIVE_IMMEDIATE[dn]
/	/	/	/	/	/	RECEIVE_IMMEDIATE[en]
/	/	/	/	/	/	RECEIVE_IMMEDIATE[ep]
/	/	/	/	/	/	RECEIVE_IMMEDIATE[et]
/	/	/	/	/	/	RECEIVE_IMMEDIATE[rf]
/	/	/	/	/	/	RECEIVE_IMMEDIATE[un]
/	/	-	/	/	/	REQUEST_TO_SEND
/	/	/	/	/	/	SEND_DATA[ok]
/	/	/	/	/	/	SEND_DATA[ae]
/	/	/	/	/	/	SEND_DATA[bo]
/	/	/	/	/	/	SEND_DATA[da]
/	/	/	/	/	/	SEND_DATA[ep]
/	/	/	/	/	/	SEND_DATA[rf]
2	2	2	2	2	/	SEND_ERROR[ok]
/	/	/	/	/	/	SEND_ERROR[ae]
/	/	/	/	/	/	SEND_ERROR[bo]
/	/	/	/	/	/	SEND_ERROR[da]
/	/	/	/	/	/	SEND_ERROR[dn]
/	/	/	/	/	/	SEND_ERROR[ep]
12	12	12	12	12	/	SEND_ERROR[rf]
/	/	5	2	12	/	SYNCPT[ok]
/	/	*	*	*	/	SYNCPT[bo]
/	/	5	2	12	/	SYNCPT[hm]
/	/	/	/	/	/	TEST(P)[ok]
/	/	/	/	/	/	TEST(P)[ae]
/	/	/	/	/	/	TEST(P)[bo]
/	/	/	/	/	/	TEST(P)[da]
/	/	/	/	/	/	TEST(P)[dn]
/	/	/	/	/	/	TEST(P)[en]
/	/	/	/	/	/	TEST(P)[ep]
/	/	/	/	/	/	TEST(P)[et]
/	/	/	/	/	/	TEST(P)[pn]
/	/	/	/	/	/	TEST(P)[rf]
/	/	/	/	/	/	TEST(P)[un]
/	/	/	/	/	/	TEST(Q)[ok]
/	/	/	/	/	/	TEST(Q)[un]

Verb	Conversation States					
	Reset	Send	Defer Receive	Defer Deallo-cate	Receive	Confirm
	1	2	3	4	5	6
WAIT[ok]	/	/	/	/	-	/
WAIT[ae]	/	/	/	/	12	/
WAIT[bo]	/	/	/	/	*	/
WAIT[da]	/	/	/	/	12	/
WAIT[dn]	/	/	/	/	12	/
WAIT[en]	/	/	/	/	-	/
WAIT[ep]	/	/	/	/	-	/
WAIT[et]	/	/	/	/	-	/
WAIT[pn]	/	/	/	/	-	/
WAIT[rf]	/	/	/	/	12	/

Parameter Abbreviations (...)

- A TYPE(ABEND_PROG), TYPE(ABEND_SVC), or TYPE(ABEND_TIMER)
- C TYPE(CONFIRM), or TYPE(SYNC_LEVEL) with synchronization level CONFIRM
- F TYPE(FLUSH)
- L TYPE(LOCAL)
- P TEST(POSTED)
- Q TEST(REQUEST_TO_SEND_RECEIVED)
- S TYPE(SYNC_LEVEL) with synchronization level SYNCPT

Return-Code Abbreviations [...]

- ae ALLOCATION_ERROR
- bo BACKED_OUT
- da DEALLOCATE_ABEND_PROG, DEALLOCATE_ABEND_SVC, or DEALLOCATE_ABEND_TIMER
- dn DEALLOCATE_NORMAL
- en PROG_ERROR_NO_TRUNC or SVC_ERROR_NO_TRUNC
- ep PROG_ERROR_PURGING or SVC_ERROR_PURGING
- et PROG_ERROR_TRUNC or SVC_ERROR_TRUNC
- nm HEURISTIC_MIXED
- ok OK
- pe PARAMETER_ERROR
- pn POSTING_NOT_ACTIVE
- rf RESOURCE_FAILURE_NO_RETRY or RESOURCE_FAILURE_RETRY
- un UNSUCCESSFUL

Figure E-3. Conversation State Transition Matrix (Part 3 of 3)

Conversation States (continued)						Verb
Confirm Send 7	Confirm Deallo-cate 8	Sync-point 9	Sync-point Send 10	Sync-point Deallo-cate 11	Deallo-cate 12	
/	/	/	/	/	/	WAIT[ok]
/	/	/	/	/	/	WAIT[ae]
/	/	/	/	/	/	WAIT[bo]
/	/	/	/	/	/	WAIT[da]
/	/	/	/	/	/	WAIT[dn]
/	/	/	/	/	/	WAIT[en]
/	/	/	/	/	/	WAIT[ep]
/	/	/	/	/	/	WAIT[et]
/	/	/	/	/	/	WAIT[pn]
/	/	/	/	/	/	WAIT[rf]

What-Received Abbreviations {...}

co	CONFIRM
cd	CONFIRM_DEALLOCATE
cs	CONFIRM_SEND
da	DATA, DATA_COMPLETE, DATA_INCOMPLETE, or LL_TRUNCATED
sd	TAKE_SYNCPT_DEALLOCATE
se	SEND
ss	TAKE_SYNCPT_SEND
sy	TAKE_SYNCPT

Matrix Symbols

/	Verb cannot be issued in this state
-	Remain in current state
number	Number of next state
*	State at completion of most recent synchronization point

Verb	Conversation States					
	Reset 1	Send 2	Defer Receive 3	Defer Deallo-cate 4	Receive 5	Confirm 6
Initiating Conversation	/	/	/	/	/	/
ALLOCATE	/	/	/	/	/	/
BACKOUT	/	/	/	/	/	/
CONFIRM	/	X	/	/	>	>
CONFIRMED	/	>	>	>	>	/
DEALLOCATE(F)	/	X	>	>	>	>
DEALLOCATE(C)	/	X	/	/	>	>
DEALLOCATE(S)	/	X	>	>	>	>
DEALLOCATE(A)	/	/	/	/	/	/
DEALLOCATE(L)	/	>	>	>	>	>
FLUSH	/	/	/	/	>	>
GET_ATTRIBUTES	/	/	/	/	/	/
GET_TYPE	/	/	/	/	/	/
POST_ON_RECEIPT	/	>	>	>	/	>
PREPARE_TO_RECEIVE(F)	/	X	>	>	>	>
PREPARE_TO_RECEIVE(C)	/	X	/	/	>	>
PREPARE_TO_RECEIVE(S)	/	X	>	>	>	>
RECEIVE_AND_WAIT	/	X	>	>	/	>
RECEIVE_IMMEDIATE	/	>	>	>	/	>
REQUEST_TO_SEND	/	>	>	>	/	/
SEND_DATA	/	/	>	>	>	>
SEND_ERROR	/	/	>	>	/	/
SYNCPT	/	X	/	/	>	>
TEST(P)	/	>	>	>	/	>
TEST(Q)	/	/	/	/	/	>
WAIT	/	/	/	/	/	/

Parameter Abbreviations (...)
- A TYPE(ABEND_PROG), TYPE(ABEND_SVC), or TYPE(ABEND_TIMER)
- C TYPE(CONFIRM), or TYPE(SYNC_LEVEL) with synchronization level CONFIRM
- F TYPE(FLUSH)
- L TYPE(LOCAL)
- P TEST(POSTED)
- Q TEST(REQUEST_TO_SEND_RECEIVED)
- S TYPE(SYNC_LEVEL) with synchronization level SYNCPT

Figure E-4. Conversation State Check Matrix

Conversation States (continued)						Verb
Confirm Send 7	Confirm Deallo-cate 8	Sync-point 9	Sync-point Send 10	Sync-point Deallo-cate 11	Deallo-cate 12	
/	/	/	/	/	/	Initiating Conversation
/	/	/	/	/	/	ALLOCATE
/	/	/	/	/	>	BACKOUT
>	>	>	>	>	>	CONFIRM
/	/	>	>	>	>	CONFIRMED
>	>	>	>	>	>	DEALLOCATE(F)
>	>	/	/	/	>	DEALLOCATE(C)
>	>	>	>	>	>	DEALLOCATE(S)
/	/	/	/	/	>	DEALLOCATE(A)
>	>	>	>	>	/	DEALLOCATE(L)
>	>	>	>	>	>	FLUSH
/	/	/	/	/	/	GET_ATTRIBUTES
/	/	/	/	/	/	GET_TYPE
>	>	>	>	>	>	POST_ON_RECEIPT
>	>	>	>	>	>	PREPARE_TO_RECEIVE(F)
>	>	/	/	/	>	PREPARE_TO_RECEIVE(C)
>	>	>	>	>	>	PREPARE_TO_RECEIVE(S)
>	>	>	>	>	>	RECEIVE_AND_WAIT
>	>	>	>	>	>	RECEIVE_IMMEDIATE
>	>	/	>	>	>	REQUEST_TO_SEND
>	>	>	>	>	>	SEND_DATA
/	/	/	/	/	>	SEND_ERROR
>	>	/	/	/	>	SYNCPT
>	>	>	>	>	>	TEST(P)
>	>	>	>	>	>	TEST(Q)
/	/	/	/	/	/	WAIT

Matrix Symbols

- `>` State check ABEND condition occurs in this state
- `X` State check ABEND condition occurs in this state if the program is in the process of sending a logical record and the record is incomplete
- `/` State check ABEND condition cannot occur in this state

This page intentionally left blank

INDEX

C

D

E

F

G

H

I

P

R

S

U

V

W

Systems Network Architecture
Transaction Programmer's
Reference Manual For
LU Type 6.2

Publication No. GC30-3084-2

READER'S COMMENT FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address, and date:

_____

_____

_____

_____

What is your occupation? _____

Number of latest Newsletter associated with this publication: _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Reader's Comment Form

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Dept. E01
P.O. Box 12195
Research Triangle Park, N.C. 27709-2195

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

File No. 370/4300/8100-30

GC30-3084-2

Printed in U.S.A.