Systems Network Architecture

GC30-3084-05

Transaction Programmer's Reference Manual for LU Type 6.2

Systems Network Architecture

GC30-3084-05

Transaction Programmer's Reference Manual for LU Type 6.2

Note !

See "Notices" on page xiii.

Sixth Edition (June 1993)

This edition, GC30-3084-5, is a major revision of the previous edition, GC30-3084-4, and obsoletes that edition; it applies until otherwise indicated in a new edition. Consult Part 3 of the latest edition of *IBM System/370, 30xx, and 4300 Processors — Bibliography*, GC20-0001, for current information on this communication architecture. For a summary of the changes in this book, see "Summary of Changes" on page xxi.

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

A form for reader's comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation
APPC Marketing Enablement III, Department EA6A
P.O. Box 12195
Research Triangle Park, North Carolina 27709, U.S.A.

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you or restricting your use of it.

Contents

Part 1. Overview

Part 3. Control Operator Verbs

Part 4. Appendixes

Table of Contents

Figures

List of Figures

Tables

List of Tables

Notices

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM licensed program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to use these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, New York 10577.

The following paragraph does not apply to the United Kingdom or any country, province, or state where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS DOCUMENT ON AN "AS IS" BASIS WITHOUT WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

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. Collectively these functions are called *Advanced Program-to-Program Communication* (**APPC**).

This book is written for and may be used and referenced by IBM customers who design their 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 and communicating with each other. When used in conjunction with applicable IBM product publications, this book is an especially useful reference 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 in "Other Publications" on page xix, are also helpful in understanding the material in this book.

This book is divided into chapters within parts; appendixes provide detailed material related to some or all of the chapters.

---

[1] This book provides an architectural description of LU 6.2 functions. IBM LU 6.2 products implement a base set of functions and may implement one or more optional sets of functions. 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 completely and 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. See Appendix A, "Base and Option Sets for Product Support" on page A-1 for more details of the requirements upon IBM products that implement LU 6.2 functions.

The material in Part 1 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 (Parts 2, 3, and 4) is organized alphabetically within categories and subcategories for ease of reference. These chapters contain the detailed descriptions of the LU 6.2 services which are invoked by commands called *verbs*. This verb reference material (Parts 2, 3, and 4) will be easier to use if you first skim the "Contents" on page iii to visualize for yourself the organization of this material.

The overall organization of the entire book is described in the following sections.

Parts

This book is divided into four parts as follows:

Part 1, "Overview" provides general information about LU type 6.2.

Part 2, "Conversation Verbs" provides detailed descriptions of all of the **conversation verbs**.

Part 3, "Control Operator Verbs" provides detailed descriptions of all of the **control operator verbs**.

Part 4, "Appendixes" provides additional details, mostly in summary form, related to some or all of the verbs.

The first three parts are divided into chapters, which are numbered consecutively across the three parts.

Chapters

The chapters of this book are:

In Part 1, "Overview" —

Chapter 1, "Introduction" provides an introduction to LU type 6.2 and the general concepts used throughout this book.

Chapter 2, "Verb Categories" gives an overview of the LU 6.2 verbs in terms of their **categories**, conversation verbs and control operator verbs.

Chapter 3, "Verb Sets" gives an overview of the LU 6.2 verbs in terms of their **sets**, dividing them into base and option sets.

Chapter 4, "Verb Description Format" presents the format by which all LU 6.2 verbs are described in this book.

In Part 2, "Conversation Verbs" —

Chapter 5, "Mapped Conversation Verbs" contains detailed descriptions of mapped conversation verbs, listed alphabetically.

Chapter 6, "Type-Independent Conversation Verbs" contains detailed descriptions of verbs that are used on either mapped or basic conversations, listed alphabetically.

Chapter 7, "Basic Conversation Verbs" contains detailed descriptions of basic conversation verbs, listed alphabetically.

In Part 3, "Control Operator Verbs" —

Chapter 8, "Change Number of Session (CNOS) Verbs" contains detailed descriptions of verbs used to change, initialize, reset, and process the number of sessions between a given pair of LUs, listed alphabetically.

Chapter 9, "Session Control Verbs" contains detailed descriptions of verbs used to activate and deactivate sessions and conversation groups, listed alphabetically.

Chapter 10, "LU Definition Verbs" contains detailed descriptions of verbs used to define, display, or delete attributes of LUs, listed alphabetically.

Chapter 11, "Miscellaneous COPR Verbs" contains detailed descriptions of verbs used for miscellaneous COPR functions that don't fit into one of the above categories. The verbs are listed alphabetically, within the function group.

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. It is here that the base set, which must be supported by all LU 6.2 implementations, and the option sets of LU6.2 functions are identified.

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 book.

Appendix D, "List of SNA Service Transaction Programs" contains a list of SNA service transaction programs (that is, the SNA-defined transaction program names).

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.

Appendix F, "Conversation Return Codes" describes return codes from LU 6.2 to the transaction program for all conversation verbs.

Appendix G, "Control-Operator Return Codes" describes return codes from LU 6.2 to the transaction program for all control operator verbs.

Other Publications

Prerequisite Publication

Systems Network Architecture Concepts and Products, GC30-3072.

Related Publications

General SNA

Systems Network Architecture Technical Overview, GC30-3073.

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

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

Systems Network Architecture Formats, GA27-3136. (This book describes, among many formats, the formats used between LU 6.2s.)

SNA LU 6.2

SNA LU 6.2 Reference: Peer Protocols, SC31-6808. (This book describes the logic within LU 6.2 and the protocols between LU 6.2s. In contrast, the *SNA Transaction Programmer's Reference Manual for LU Type 6.2*, the book that you are now reading, describes the generic interface, or *protocol boundary*, between LU 6.2 and a transaction program.)

Systems Network Architecture Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2, SC30-3269-3. (This book describes the SSCP-dependent LU protocols, including SSCP-LU session services and SSCP-LU session control protocols. For protocols common to the dependent and independent LUs, see the preceding reference.)

CPI Communications

System Applications Architecture: Common Programming Interface Communications Reference, SC26-4399. (This book describes the uniform syntax to be used by transaction programs requesting LU 6.2 functions, which is supported by IBM products. This book is intended for CPI-C programmers.)

Common Programming Interface Communications Specification, SC31-6180. (This book describes the uniform syntax to be used by transaction programs requesting LU 6.2 functions. This book is intended for CPI-C developers.)

SNA APPN

SNA APPN Architecture Reference, SC30-3422.

SNA/DS

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

DIA

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.

SNA/MS

Systems Network Architecture Management Services Reference, SC30-3346.

SNA/FS

Systems Network Architecture File Services Reference, SC31-6807.

Summary of Changes

This edition includes new functions and editorial changes as identified in the following sections.

New Functions

LU 6.2 functions are available through verbs, verb parameters, and LU 6.2 option sets, which have been changed as follows:

- Existing conversation verbs have been modified to provide nonblocking support, full-duplex conversations and expedited data support, presume abort support for sync point, and enhanced SECURITY SAME features in the User ID verification tower. The LU definition verbs have been modified to provide for origin LU authorization and length-checked compression. The overview chapters and the appendices have been modified to reflect the inclusion of the new functions offered by the verbs.
- The new *VERBs* that provide a part of the new expedited data function are as follows.

  (MC_)RECEIVE_EXPEDITED_DATA

  (MC_)SEND_EXPEDITED_DATA
- The new *VERB* that is a part of the new nonblocking function is as follows.

  WAIT_FOR_COMPLETION
- New LU 6.2 option sets have been added. (See Appendix A.)

  The new option set numbers and names, are as follows:

  112 — Full-duplex conversations and expedited data
  113 — Nonblocking support
  216 — Origin LU authorization
  250 — Presume Abort performance improvements for Sync-point
  618 — Length-checked compression
- The USER_CONTROL_DATA parameter has been deleted from the following verbs:

  DEFINE_TP
  DISPLAY_TP

Editorial Changes

Figures and state transition-related matrices pertaining to half-duplex conversations have been renamed to indicate that they apply to half-duplex conversations and to distinguish them from the new figures and state transition-related matrices for full-duplex conversations.

Part 1. Overview

This part of the book gives an introduction to the general concepts of LU 6.2 and an overview of the LU 6.2 verbs.

The first chapter gives an introduction to the concepts of:

1. Logical unit type 6.2
2. Transaction program
3. Protocol boundary

The Introduction describes the general concepts used throughout the book.

The following two chapters give an overview of the LU 6.2 verbs by presenting them in two different ways:

1. Verb **CATEGORIES** and
2. Verb **SETS**.

Verb Categories divide verbs based upon their individual functions and the orientation of those functions:

- *Conversation* Verbs (application-oriented functions) and
- *Control-Operator* Verbs (system-oriented functions).

Verb Sets divide verbs based upon the LU 6.2 product implementation:

- A *Base Set* of Verbs (always available) and
- Multiple *Option Sets* of Verbs (may or may not be available).

Following the verb categories and verb sets chapters is a chapter that explains the format by which all of the LU 6.2 verbs are described in detail in parts 2 and 3 of this book.

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: Architecture 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 LU 6.2 Reference: Peer Protocols*.

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

---

[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.

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 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.

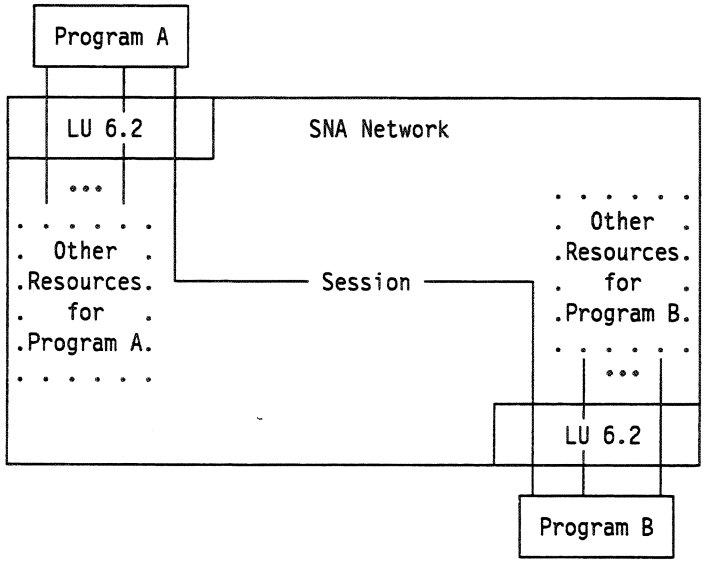


Figure 1-1. Transaction Programs and SNA Resources

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 LU 6.2 Reference: Peer Protocols* 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.

Flexible conversation styles — Conversations have the flexibility of selecting a conversation style of either two-way alternate (half-duplex) or two-way simultaneous (full-duplex). In addition, with either conversation style, conversations may send and receive selected user-defined data in an expedited manner. This choice of conversation styles gives the transaction program designer the flexibility to choose the style that is best suited for the type of transaction.

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.

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

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.

Data compression — To efficiently use transmission bandwidth, transaction programs can use sessions that employ data compression. SNA's length-checked compression is transparent to the transaction program. The possible benefits of using compression include improved response time and decreased transmission costs. Compression can be requested on a mode or LU basis.

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.

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 1-2 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 LU 6.2 Reference: Peer Protocols.*

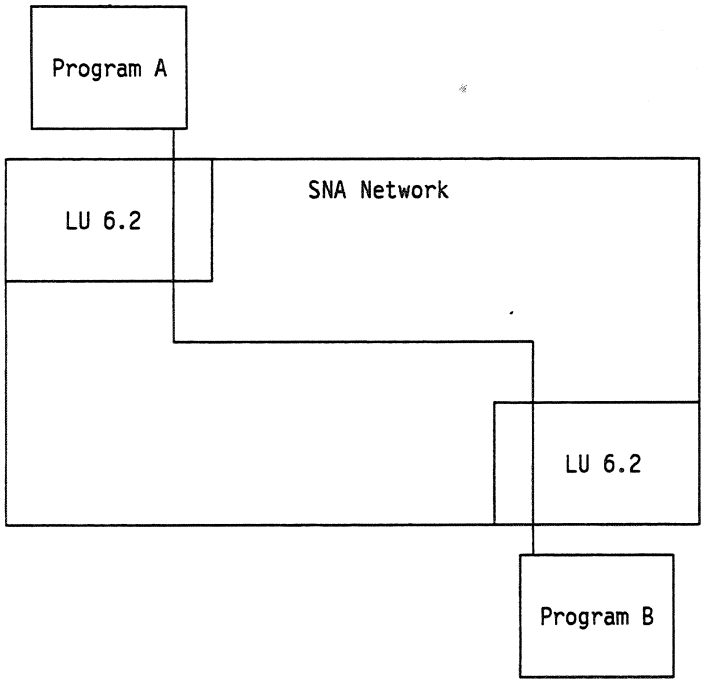


Figure 1-2. Program-to-Program Connection Through the SNA Network

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 1-3.

Figure 1-3. Effective Program-to-Program Connection

This view of program-to-program connection can be extended to a more general configuration of interconnected programs. Figure 1-4 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.

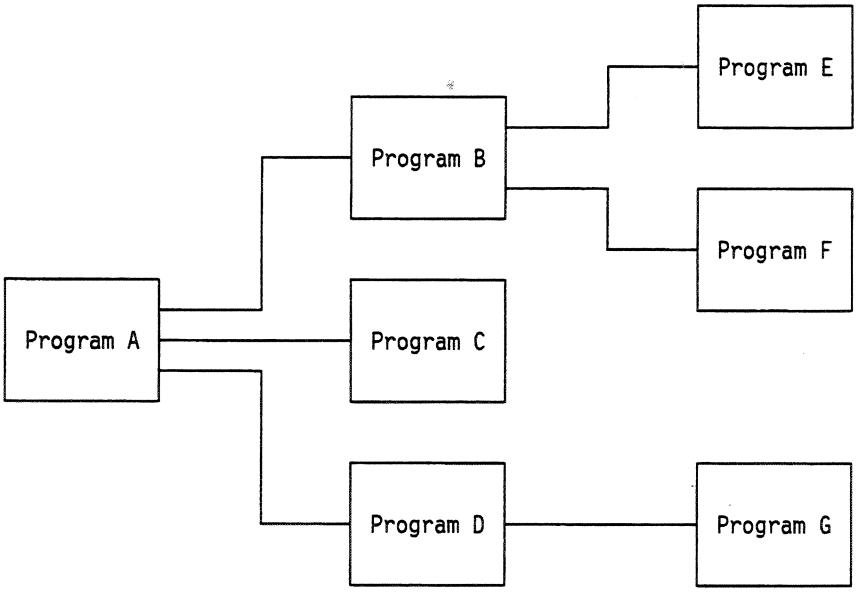


Figure 1-4. A Configuration of Interconnected Programs

The configuration of interconnected programs changes over time. In the example shown in Figure 1-4, 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 1-4, 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 2, "Verb Categories." and Chapter 3, "Verb Sets." The detailed descriptions of the verbs are given in Part 2, "Conversation Verbs" and Part 3, "Control Operator Verbs." Resource states associated with the conversation verbs are described in Appendix E, "Conversation State Matrices."

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,[4] 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.

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

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 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 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.

Transaction programs may select a conversation style of either two-way alternate (half-duplex) or two-way simultaneous (full-duplex). The conversation style desired is specified via the TYPE parameter on the (MC_)ALLOCATE verb.

Once a half-duplex conversation is allocated, the 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 send control to the other program.

Once a full-duplex conversation is allocated, both of the programs connected to the conversation are started in send-and-receive state. Both programs may issue verbs to send and receive data simultaneously with no transfer of send control required.

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.

Chapter 2. Verb Categories

This chapter presents an overview of the verbs in terms of their individual functions and the orientation of those functions. The verbs are divided into two major categories:

Conversation verbs
Control-operator verbs

as shown in Figure 2-1.

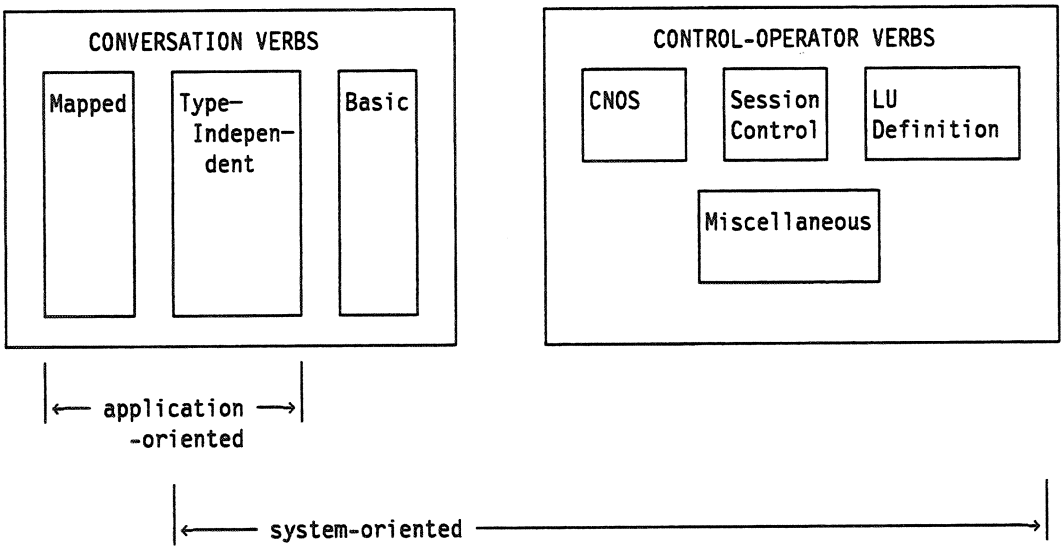


Figure 2-1. Two Major Categories of LU 6.2 Verbs

Each major category is divided into subcategories. Referring to Figure 2-1 from left to right, the first subcategory of verbs (mapped) is primarily for application-oriented TPs; the second subcategory of verbs (type-independent) is for either application- or system-oriented TPs; and the remaining subcategories are primarily for system-oriented TPs.

Each category defines a major subdivision of the LU 6.2 conversation 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.

The two major categories of LU 6.2 verbs are shown in Figure 2-2 on page 2-2 with two categories of TPs, a "regular" TP using the conversation verbs and a "control-operator" TP using the control-operator verbs. A regular TP communicates with a regular TP; a control-operator TP communicates with a control-operator TP. A regular TP uses either mapped and type-independent verbs or basic and type-independent verbs; a control-operator TP uses control-operator verbs. It is, of course, possible for a program to be written using mapped conversation verbs on

one conversation, basic conversation verbs on another conversation, and control-operator verbs on yet another conversation, but this is not the usual case.

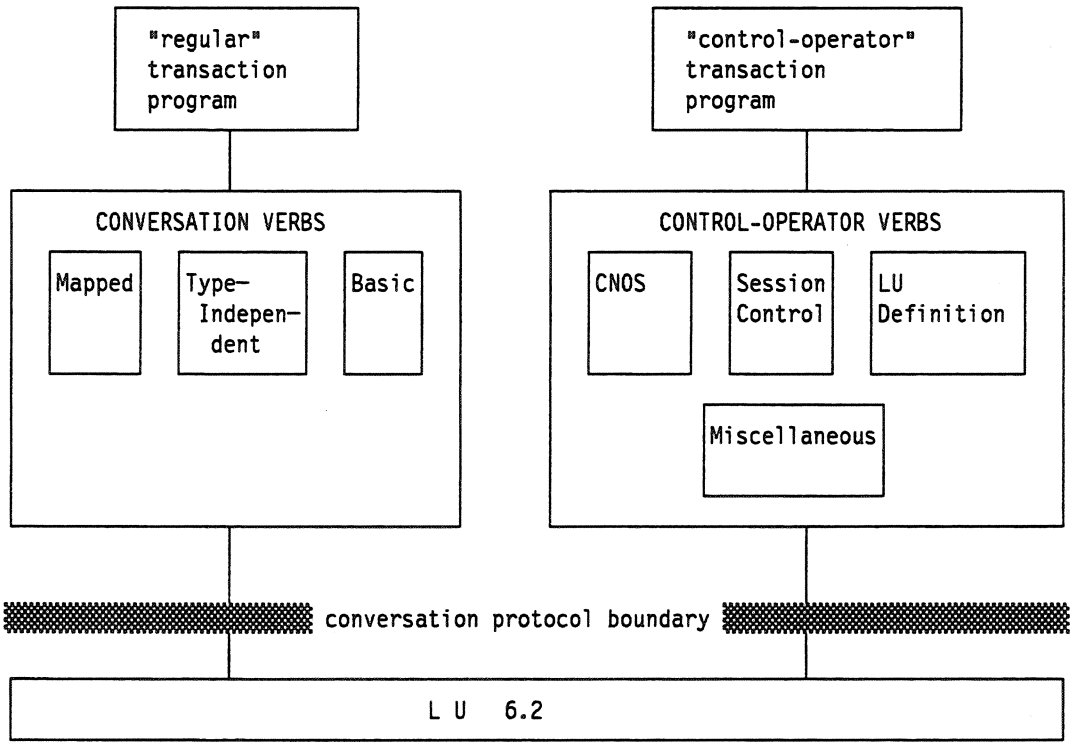


Figure 2-2. Two Categories of TPs and Two Categories of LU 6.2 Verbs

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*.[1] The overview description of the conversation verbs and control-operator verbs follows.

[1] 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.

Conversation Verbs

The conversation verbs, the first major category of LU 6.2 verbs, provide program-to-program communication by means of conversations between programs. The following combination of conversation types and conversation styles are specified by the TYPE parameter:

Mapped (half-duplex)
Basic (half-duplex)
Full-duplex mapped
Full-duplex basic

The verbs defining the conversation protocol boundary are divided into subcategories based on whether the verb is for application-oriented use (mapped), system-oriented use (basic), or either (type-independent). For both the mapped and basic subcategories, the program may select a conversation style of either two-way alternate (half-duplex) or two-way simultaneous (full-duplex). *(Full-duplex conversations are provided only by option set #112. See Table A-2.)* The subcategories of verbs are:

- Mapped conversation verbs for:
  - half-duplex conversations, or
  - full-duplex conversations (option set #112)
- Type-independent conversation verbs[2]
- Basic conversation verbs for:
  - half-duplex conversations, or
  - full-duplex conversations (option set #112)

as shown in Figure 2-3.

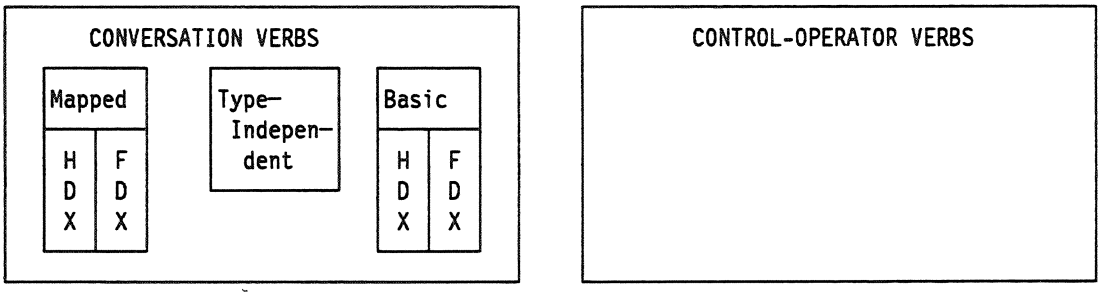


Figure 2-3. Three Subcategories of Conversation Verbs. The Mapped and Basic subcategories include support for half-duplex (HDX) and full-duplex (FDX) conversation styles.

[2] The type-independent verbs do not transfer data between transaction programs, and therefore duplexing is not a relevant attribute.

The verbs that require resource allocation (such as ALLOCATE and SEND_DATA) are further categorized as:

- Blocking verbs
- Nonblocking verbs

based on their processing.

Conversation Styles

LU 6.2 transaction programs may select either two-way alternate (half-duplex) or two-way simultaneous (full-duplex) conversations, if both the local and remote LUs support full-duplex conversations. *(Full-duplex conversations are provided only by option set #112, see Table A-2 on page A-8.)* The transaction program specifies this via the TYPE parameter on the (MC_)ALLOCATE verb.

When the local program initiates a half-duplex conversation (by issuing an (MC_)ALLOCATE verb), the local program is started in send state and the remote program is started in receive state. When the local program is finished sending and wishes to receive, it transfers send control to the remote program. This conversation style is useful for request/reply types of transactions. Because the program cannot send and receive simultaneously, it may assume that when it finishes sending a request and relinquishes send control to the partner, that the data received will be the partner program's reply to the request sent earlier. This implied correlation frees the program from implementation of request/response correlation values in the data.

When the local program initiates a full-duplex conversation, both the local and remote programs are started in send-and-receive state, where both sides may send and receive data simultaneously with no concept of send control. This conversation style is useful for communication subsystems that interleave multiple requests and replies onto a single full-duplex conversation. The data that is sent may be completely independent of the data that is received; the program may choose to relate the information if so desired; it is an application design choice.

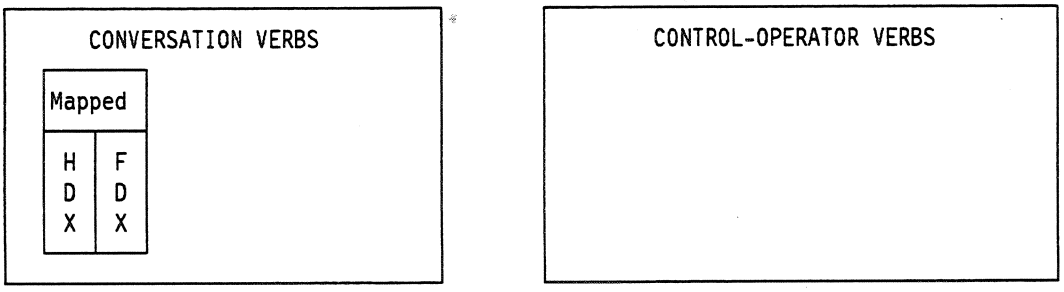
The use of nonblocking verbs with full-duplex conversations is recommended. In order to have a truly full-duplex conversation, where both send queue and receive queue verbs are outstanding simultaneously, the use of nonblocking verbs is required. With no restrictions on when the transaction programs may send and receive, transaction program deadlocks are more likely to occur. The use of nonblocking verbs will reduce the likelihood of transaction program deadlocks. See "Nonblocking Support for LU 6.2 Conversation Verbs" on page 3-9.

The conversation verbs provided for use with full-duplex conversations differ from those provided for half-duplex conversations. The verbs that the LU 6.2 protocol boundary provides for each are shown in the following table.

Half-Duplex Conversation Verbs	Full-Duplex Conversation Verbs
(MC_)ALLOCATE (MC_)CONFIRM (MC_)CONFIRMED (MC_)DEALLOCATE (MC_)FLUSH (MC_)GET_ATTRIBUTES (MC_)POST_ON_RECEIPT (MC_)PREPARE_FOR_SYNCPT (MC_)PREPARE_TO_RECEIVE (MC_)RECEIVE_AND_WAIT (MC_)RECEIVE_EXPEDITED_DATA (MC_)RECEIVE_IMMEDIATE (MC_)REQUEST_TO_SEND (MC_)SEND_DATA (MC_)SEND_ERROR (MC_)SEND_EXPEDITED_DATA (MC_)TEST	(MC_)ALLOCATE (MC_)DEALLOCATE (MC_)FLUSH (MC_)GET_ATTRIBUTES (MC_)RECEIVE_AND_WAIT (MC_)RECEIVE_EXPEDITED_DATA (MC_)RECEIVE_IMMEDIATE (MC_)SEND_DATA (MC_)SEND_ERROR (MC_)SEND_EXPEDITED_DATA

An overview of the subcategories of conversation verbs follows. Whenever a verb is optionally provided by an LU 6.2 implementation, the option set number that provides the verb is indicated following the verb description. See Chapter 3, "Verb Sets" on page 3-1 for an explanation of option sets and see your particular LU 6.2 product publications for a list of the option sets supported.

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.

Half-Duplex Conversations

A brief description of the mapped conversation verbs provided for half-duplex conversations follows.

MC_ALLOCATE allocates a mapped conversation connecting the local transaction program to a remote transaction program. A unique resource ID (in effect, a conversation 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 interactively communicate during a particular time period for distributed processing between the two programs.

MC_CONFIRMED sends a confirmation reply to the remote transaction program, in order for the two programs to interactively communicate during a particular time period for distributed processing between the two programs. 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. (Afterward, the transaction program can, of course, continue to execute without the just-deallocated conversation or any other conversation.)

MC_FLUSH transmits all information that the LU has buffered, such as data records from preceding MC_SEND_DATAs. *(This verb is provided only by option set #101. See Table A-2.)*

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 syn-

chronization level allocated for the mapped conversation. *(This verb is provided only by option set #102. See Table A-2.)*

Note: The GET_ATTRIBUTES verb is in the LU 6.2 base set of verbs; the MC_GET_ATTRIBUTES verb is not.

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. *(This verb is provided only by option sets #103 or 104. See Table A-2.)*

MC_PREPARE_FOR_SYNCPT causes the mapped conversation partner to prepare to advance to the next synchronization point. If an error is detected, the program has a chance to correct it and try the sync point operation again. *(This verb is provided only by option set #111. See Table A-2.)*

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. *(This verb is provided only by option set #105. See Table A-2.)*

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_EXPEDITED_DATA receives data sent by the remote transaction program in an expedited manner, via the MC_SEND_EXPEDITED_DATA verb. If expedited data is not available to receive, the program may or may not wait for expedited data to arrive on the mapped conversation, based on the value specified for the RETURN_CONTROL parameter. The mapped conversation does not change its state as a result of issuing this verb. *(This verb is provided only by option set #112. See Table A-2.)*

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. *(This verb is provided only by option set #106. See Table A-2.)*

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 (function management) 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_SEND_EXPEDITED_DATA sends data to the remote transaction program in an expedited manner. This means it may arrive at the remote transaction program before data sent earlier via the MC_SEND_DATA verb. The data record consists entirely of data and is not in logical record format; no length field (LL) is present. The mapped conversation does not change its state as the result of issuing this verb. *(This verb is provided only by option set #112. See Table A-2.)*

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. *(This verb is provided only by option set #103. See Table A-2.)*

Full-Duplex Conversations

A brief description of the mapped conversation verb provided for full-duplex conversations follows.

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

MC_DEALLOCATE closes the mapped conversation send queue and prohibits the program from issuing further send queue verbs. (See Table 3-2 on page 3-10 for the definition of send queue verbs.) The program issues this verb when it has finished sending data on the mapped conversation. The program will continue to receive data on the mapped conversation until it receives either a DEALLOCATE_NORMAL or DEALLOCATE_ABEND return code from the remote program.

MC_FLUSH transmits all information that the LU has buffered, such as data records from preceding MC_SEND_DATAs. *(This verb is provided only by option set #101. See Table A-2.)*

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. *(This verb is provided only by option set #102. See Table A-2.)*

Note: The GET_ATTRIBUTES verb is in the LU 6.2 base set of verbs; the MC_GET_ATTRIBUTES verb is not.

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 or mapped conversation status. Control is returned to the program with an indication of the type of information.

MC_RECEIVE_EXPEDITED_DATA receives data sent by the remote transaction program in an expedited manner, via the MC_SEND_EXPEDITED_DATA verb. If expedited data is not available to receive, the program may or may not wait for expedited data to arrive on the mapped conversation, based on the value specified for the RETURN_CONTROL parameter. The mapped conversation does not change its state as a result of issuing this verb. *(This verb is provided only by option set #112. See Table A-2.)*

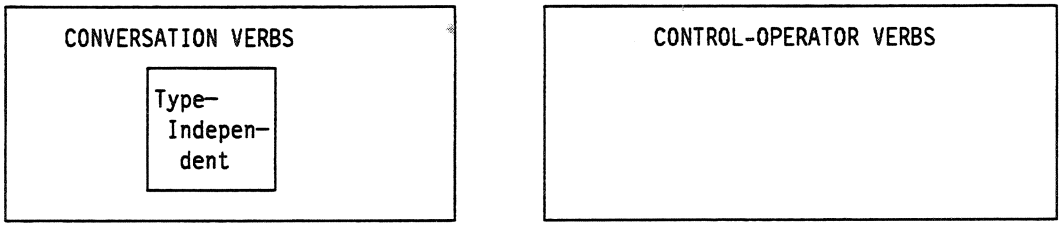
MC_RECEIVE_IMMEDIATE receives any information that is available from the specified mapped conversation, but does not wait for information to arrive. The information (if any) can be a data record or mapped conversation status. Control is returned to the program with an indication of whether any information was received and, if so, the type of information. *(This verb is provided only by option set #106. See Table A-2.)*

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 (function management) headers.

MC_SEND_ERROR informs the remote transaction program that the local program has detected an application error in a data record that it has sent. This verb applies to the send direction only and is not used to report errors in data records received.

MC_SEND_EXPEDITED_DATA sends data to the remote transaction program in an expedited manner. This means it may arrive at the remote transaction program before data sent earlier via a send queue verb, e.g., MC_SEND_DATA. (See Table 3-2 on page 3-10 for a complete list of send queue verbs.) The data record consists entirely of data and is not in logical record format; no length field (LL) is present. The mapped conversation does not change its state as the result of issuing this verb. *(This verb is provided only by option set #112. See Table A-2.)*

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. *(This verb is provided only by option set #108. See Table A-2.)*

GET_TP_PROPERTIES returns the properties of the transaction program (TP). The (MC_)GET_ATTRIBUTES verb gets attributes of a conversation, while the GET_TP_PROPERTIES verb gets properties of the transaction program that are common across all conversations this transaction program may have.

GET_TYPE returns the type of resource. The possible types returned are:

MAPPED_CONVERSATION
BASIC_CONVERSATION
FULL_DUPLEX_MAPPED_CONVERSATION
FULL_DUPLEX_BASIC_CONVERSATION.

(This verb is provided only by option set #110. See Table A-2.)

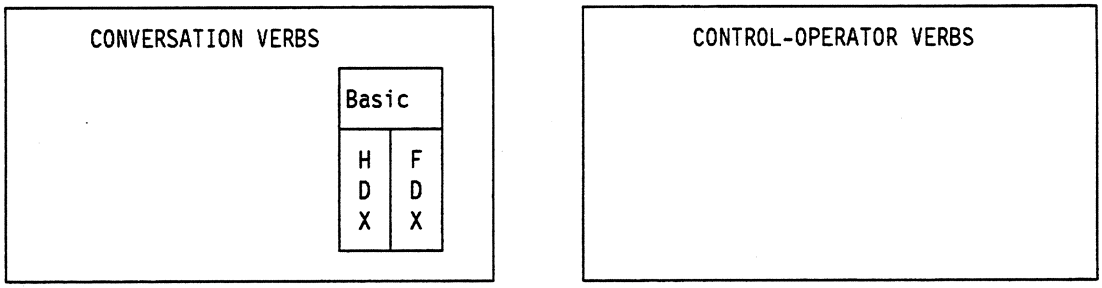
SET_SYNCPT_OPTIONS changes the options governing the execution of the sync point verbs: (MC_)PREPARE_FOR_SYNCPT, SYNCPT, and BACKOUT. These options can affect the performance of the sync point verbs. *(This verb is provided only by option set #108. See Table A-2.)*

SYNCPT advances all protected resources throughout a distributed transaction to the next synchronization point. *(This verb is provided only by option set #108. See Table A-2.)*

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. *(This verb is provided only by option set #104. See Table A-2.)*

WAIT_FOR_COMPLETION waits for posting to occur on one or more nonblocking operations represented in the specified list of wait objects. Posting of a nonblocking operation occurs when the LU has completed the associated nonblocking verb and filled all the return values. *(This verb is provided only by option set #113. See Table A-2.)*

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.

Half-Duplex Conversations

A brief description of the basic conversation verbs provided for half-duplex conversations 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 (in effect, a conversation 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 interactively communicate during a particular time period for distributed processing between the two programs.

CONFIRMED sends a confirmation reply to the remote program, in order for the two programs to interactively communicate during a particular time period for distributed processing between the two programs. 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. (Afterward, the transaction program can, of course, continue to execute without the just-deallocated conversation or any other conversation.)

FLUSH transmits all information that the LU has buffered, such as data from preceding SEND_DATAs. *(This verb is provided only by option set #101. See Table A-2.)*

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.

Note: The GET_ATTRIBUTES verb is in the LU 6.2 base set of verbs; the MC_GET_ATTRIBUTES verb is not.

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. *(This verb is provided only by option sets #103 or 104. See Table A-2.)*

PREPARE_FOR_SYNCPT causes the conversation partner to prepare to advance to the next synchronization point. If an error is detected, the program has a chance to correct it and try the sync point operation again. *(This verb is provided only by option set #111. See Table A-2.)*

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. *(This verb is provided only by option set #105. See Table A-2.)*

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_EXPEDITED_DATA receives data sent by the remote transaction program in an expedited manner, via the SEND_EXPEDITED_DATA verb. If expedited data is not available to receive, the program may or may not wait for expedited data to arrive on the mapped conversation, based on the value specified for the RETURN_CONTROL parameter. The mapped conversation does not change its state as a result of issuing this verb. *(This verb is provided only by option set #112. See Table A-2.)*

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. *(This verb is provided only by option set #106. See Table A-2.)*

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.

SEND_EXPEDITED_DATA sends data to the remote transaction program in an expedited manner. This means it may arrive at the remote transaction program before data sent earlier via the MC_SEND_DATA verb. The data record consists entirely of data and is not in logical record format; no length field (LL) is present. The mapped conversation does not change its state as the result of issuing this verb. *(This verb is provided only by option set #112. See Table A-2.)*

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. *(This verb is provided only by option set #103. See Table A-2.)*

Full-Duplex Conversations

A brief description of the basic conversation verbs provided for full-duplex conversations follows.

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

DEALLOCATE closes the conversation send queue and prohibits the program from issuing further send queue verbs. (See Table 3-2 on page 3-10 for the definition of send queue verbs.) The program issues this verb when it has finished sending data on the conversation. The program will continue to receive data on the conversation until it receives either a DEALLOCATE_NORMAL or DEALLOCATE_ABEND_* return code from the remote program.

FLUSH transmits all information that the LU has buffered; such as data from preceding SEND_DATAs, and notifies the remote program's receive queue that information has arrived. See Table 3-2 on page 3-10 for the definition of receive queue verbs. *(This verb is provided only by option set #101. See Table A-2.)*

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.

Note: The GET_ATTRIBUTES verb is in the LU 6.2 base set of verbs; the MC_GET_ATTRIBUTES verb is not.

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 or conversation status. Control is returned to the program with an indication of the type of information received.

RECEIVE_EXPEDITED_DATA receives data sent by the remote transaction program in an expedited manner, via the SEND_EXPEDITED_DATA verb. If expedited data is not available to receive, the program may or may not wait for expedited data to arrive on the mapped conversation, based on the value specified for the RETURN_CONTROL parameter. The mapped conversation does not change its state as a result of issuing this verb. *(This verb is provided only by option set #112. See Table A-2.)*

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 or conversation status. Control is returned to the program with an indication of whether any information was received and, if so, the type of information. *(This verb is provided only by option set #106. See Table A-2.)*

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 transaction program that the local program has detected an application error in a data record that it has sent, and truncates that logical record. This verb applies to the send direction only and is not used to report errors in data records received.

SEND_EXPEDITED_DATA sends data to the remote transaction program in an expedited manner. This means it may arrive at the remote transaction program before data sent earlier via a send queue verb, e.g., SEND_DATA. (See Table 3-2 on page 3-10 for a complete list of send queue verbs.) The data record consists entirely of data and is not in logical record format; no length field (LL) is present. The mapped conversation does not change its state as the result of issuing this verb. *(This verb is provided only by option set #112. See Table A-2.)*

Control-Operator Verbs

The control-operator verbs, the second major category of LU 6.2 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 (CNOS) verbs
Session control verbs
LU definition verbs

as shown in Figure 2-4.

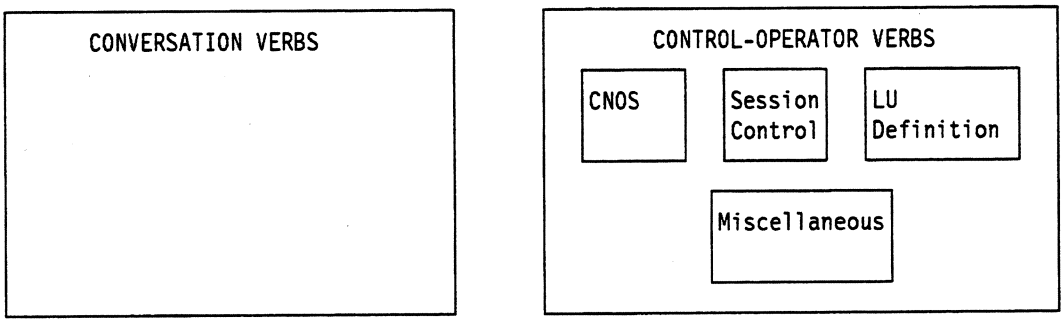
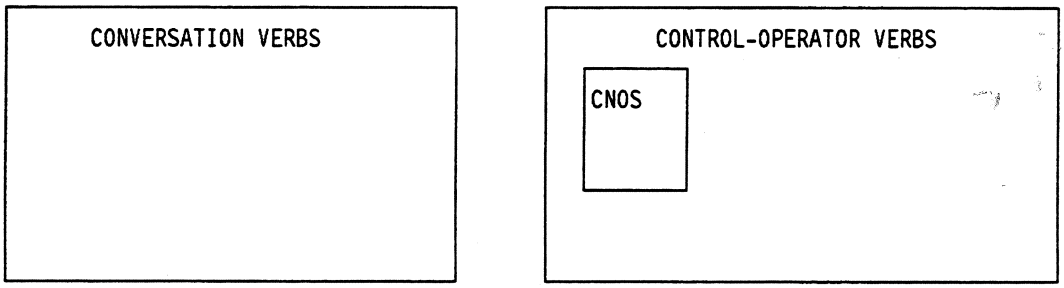


Figure 2-4. Three Subcategories of Control-Operator Verbs

An overview of the three subcategories of control-operator verbs follows.

Whenever a verb is optionally provided by an LU 6.2 implementation, the option set number that provides the verb is indicated following the verb description. See Chapter 3, "Verb Sets" on page 3-1 for an explanation of option sets and see your particular LU 6.2 product publications for a list of the option sets supported.

Change Number of Sessions (CNOS) Verbs



This subcategory of control-operator verbs consists of 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 (at the local LU) may issue are:

CHANGE_SESSION_LIMIT changes the (LU,mode) session limit from one nonzero value to another nonzero value. *(This verb is provided by option set #501 for local support. See Table A-2.)*

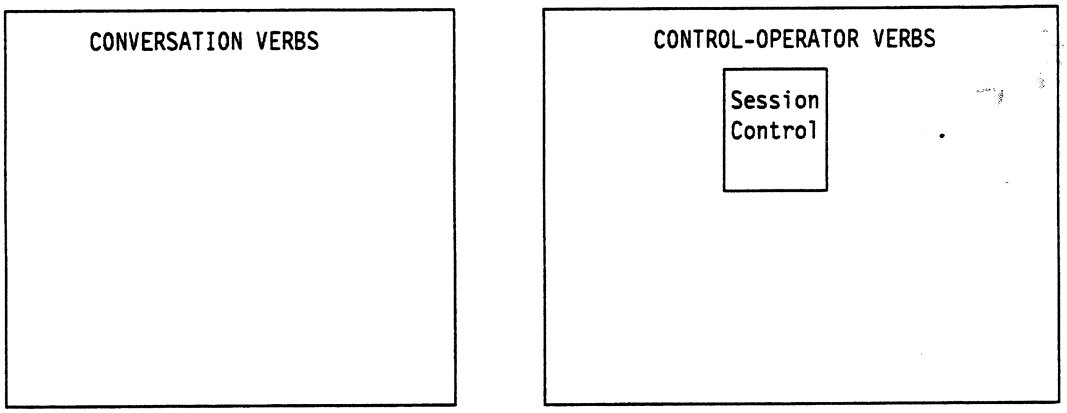
INITIALIZE_SESSION_LIMIT changes the (LU,mode) session limit from 0 to a value greater than 0.

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 (at the remote LU that receives the CNOS request) 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



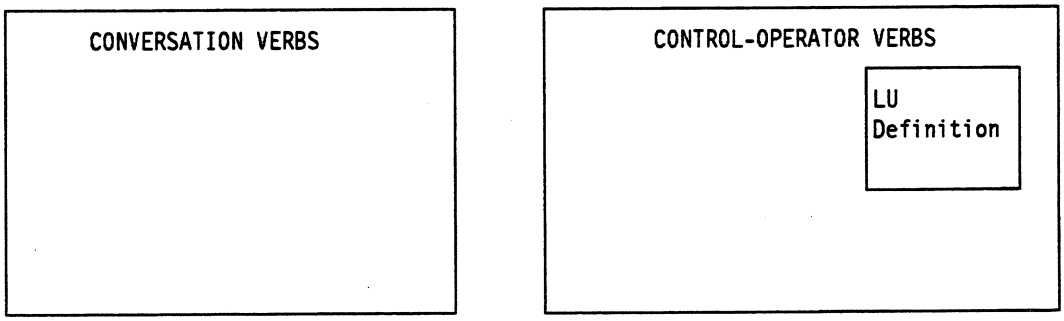
This subcategory of control-operator verbs consists of three verbs used for session control, one that activates an LU-LU session, one that deactivates a conversation group (a particular session or set of sessions), and one that deactivates an LU-LU session. These verbs are:

ACTIVATE_SESSION activates an LU-LU session between the local LU and a specified LU. *(This verb is provided only by option set #502. See Table A-2.)*

DEACTIVATE_CONVERSATION_GROUP deactivates a specified conversation group. The type of deactivation can be cleanup or normal. *(This verb is provided only by option set #401. See Table A-2.)*

DEACTIVATE_SESSION deactivates a specified LU-LU session. The type of deactivation can be cleanup or normal. *(This verb is provided only by option set #504. See Table A-2.)*

LU Definition Verbs



This subcategory of control-operator verbs consists of verbs 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. *(This verb is provided only by option set #505. See Table A-2.)*

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. *(This verb is provided only by option set #505. See Table A-2.)*

DEFINE_REMOTE_LU initializes or modifies parameter values that control the operation of the local LU in conjunction with a remote LU. *(This verb is provided only by option set #505. See Table A-2.)*

DEFINE_TP initializes or modifies parameter values that control the operation of the local LU in conjunction with a local transaction program. *(This verb is provided only by option set #505. See Table A-2.)*

DELETE deletes the local LU's operating-parameter values that have been defined by means of the DEFINE verbs. *(This verb is provided only by option set #505. See Table A-2.)*

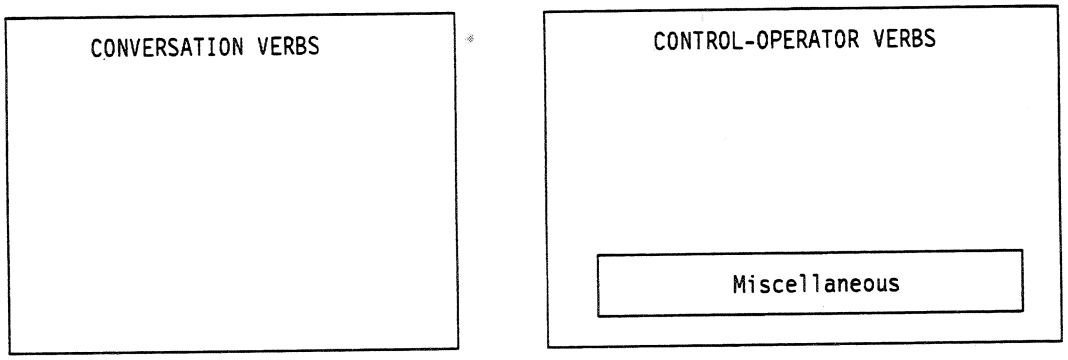
DISPLAY_LOCAL_LU returns parameter values that control the operation of the local LU. *(This verb is provided only by option set #505. See Table A-2.)*

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. *(This verb is provided only by option set #505. See Table A-2.)*

DISPLAY_REMOTE_LU returns parameter values that control the operation of the local LU in conjunction with the remote LU. *(This verb is provided only by option set #505. See Table A-2.)*

DISPLAY_TP returns parameter values that control the operation of the local LU in conjunction with a local transaction program. *(This verb is provided only by option set #505. See Table A-2.)*

Miscellaneous COPR Verbs



This subcategory of control-operator verbs consists of verbs providing functions that do not fit into one of the previously listed categories. The verbs are:

DISPLAY_SIGNED_ON_LIST returns the entries in each specified "signed-on" list to the control operator. *(This verb is provided only by option sets #219 #220. See Table A-2.)*

SIGNOFF requests removal of entries meeting specified criteria from one or both of the local LU's signed-on lists. If the local signed-on-from list is specified, the LU initiates appropriate sign-off actions so affected partner LUs can remove any corresponding entries from their signed-on-to lists. *(This verb is provided only by option sets #219 #220. See Table A-2.)*

PROCESS_SIGNOFF causes the LU receiving a sign-off request from a partner LU to process the request and remove from its local signed-on-to list the entries specified by the partner LU. *(This verb is provided only by option sets #219 #220. See Table A-2.)*

Chapter 3. Verb Sets

This chapter presents an overview of the verbs in terms of the two kinds of LU 6.2 implementation sets into which the verbs are divided:

BASE set

OPTION sets

as shown in Figure 3-1.

There is one set of verbs, a minimum set called the *BASE set*, that is always provided and there are other sets of verbs that are optionally provided by some of the various *OPTION sets* of LU 6.2, where an option set provides one or more verbs. Not every option set implements verbs; some implement just parameters and parameter values on one or more verbs. (Sets of verbs are discussed in detail in Appendix A, "Base and Option Sets for Product Support" on page A-1.) See the "RETURN_CODE parameter" entry in the index to this book for a straight alphabetical listing of all LU 6.2 verb names.

Some products implement no option sets, just the minimum set of verbs (that is, the base set).

There are two categories of LU 6.2 verbs (conversation and control-operator as explained in Chapter 2, "Verb Categories" on page 2-1), and each category is divided into a base set and option sets as shown in Figure 3-1.

CATEGORY	SET	
	BASE SET	OPTION SETS
CONVERSATION	Base Conversation Verbs	Optional Conversation Verbs
CONTROL-OPERATOR	Base Control-Operator Verbs	Optional Control-Operator Verbs

Figure 3-1. Verb Categories and Verb Sets

The verb sets are listed, combining the mapped and basic conversation subcategories, as follows:

BASE Conversation Verbs	OPTIONAL Conversation Verbs
(MC_)ALLOCATE (MC_)CONFIRM (MC_)CONFIRMED (MC_)DEALLOCATE GET_ATTRIBUTES[2] GET_TP_PROPERTIES (MC_)RECEIVE_AND_WAIT (MC_)REQUEST_TO_SEND (MC_)SEND_DATA (MC_)SEND_ERROR	BACKOUT (MC_)FLUSH[1] MC_GET_ATTRIBUTES[2] GET_TYPE (MC_)POST_ON_RECEIPT (MC_)PREPARE_FOR_SYNCPT (MC_)PREPARE_TO_RECEIVE[1] (MC_)RECEIVE_EXPEDITED_DATA (MC_)RECEIVE_IMMEDIATE (MC_)SEND_EXPEDITED_DATA SET_SYNCPT_OPTIONS SYNCPT (MC_)TEST WAIT WAIT_FOR_COMPLETION

BASE Control-Operator Verbs	OPTIONAL Control-Operator Verbs
CHANGE_SESSION_LIMIT INITIALIZE_SESSION_LIMIT PROCESS_SESSION_LIMIT RESET_SESSION_LIMIT	ACTIVATE_SESSION[1] CHANGE_SESSION_LIMIT[1] DEACTIVATE_CONVERSATION_GROUP DEACTIVATE_SESSION[1] DEFINE_LOCAL_LU DEFINE_MODE DEFINE_REMOTE_LU DEFINE_TP DELETE DISPLAY_LOCAL_LU DISPLAY_MODE DISPLAY_REMOTE_LU DISPLAY_SIGNED_ON_LIST DISPLAY_TP PROCESS_SIGNOFF SIGNOFF

[1] The (MC_)FLUSH and (MC_)PREPARE_TO_RECEIVE conversation verbs together with the CHANGE_SESSION_LIMIT and ACTIVATE_SESSION and DEACTIVATE_SESSION control-operator verbs are BASE verbs for remote support but OPTION verbs for local support. (See Appendix A, "Base and Option Sets for Product Support" on page A-1.)

[2] GET_ATTRIBUTES is a BASE verb; MC_GET_ATTRIBUTES is an OPTION verb.

All the option set verbs are discussed in detail in Appendix A, "Base and Option Sets for Product Support" on page A-1.

The following sections describe all LU 6.2 verbs by groups in terms of their general purpose.

Verbs in the LU 6.2 BASE Set (Alphabetically within Purpose Groups)

The minimum set of LU 6.2 verbs, called the **BASE**[3] **set of verbs**, is shown as follows, alphabetically within groups according to the general purpose of the verbs. It is this minimum set of verbs (including the mapped conversation verbs with an "MC" prefix, except for MC_GET_ATTRIBUTES) that every LU 6.2 product implementation always supports.[4]

The base set of verbs is, however, only part of the base set of LU 6.2 functions that implementations must support. Refer to the tables on pages A-25 through A-52.

BASE Verbs for starting/stopping conversations	**(MC_)ALLOCATE** start a conversation **(MC_)DEALLOCATE** stop a conversation **GET_ATTRIBUTES** determine conversation attributes[5] **GET_TP_PROPERTIES** determine TP properties
BASE Verbs for using conversations	**(MC_)RECEIVE_AND_WAIT** receive data, with waiting if necessary **(MC_)REQUEST_TO_SEND** request send control for sending data **(MC_)SEND_DATA** send data

[3] All the verb names listed are in the base set but not all the verb parameters and verb parameter values for these verbs are in the base set. See Appendix A, "Base and Option Sets for Product Support" on page A-1 for a detailed explanation.

[4] There are a few implementation exceptions (for example, some system printers) that have fewer verbs but these TPs communicate only with specific companion TPs and not with all transaction programs. Furthermore, these implementations are not programmable by transaction programmers and, hence, are said to have a closed application program interface (API).

[5] The MC_GET_ATTRIBUTES verb is not in the LU 6.2 BASE set of verbs; the GET_ATTRIBUTES verb is.

BASE Verbs for checking conversations	**(MC_)CONFIRM** request remote confirmation **(MC_)CONFIRMED** send remote confirmation **(MC_)SEND_ERROR** send error (message)
BASE Verbs for managing the logical network	**CHANGE_SESSION_LIMIT** change maximum number of sessions[6] **INITIALIZE_SESSION_LIMIT** initialize maximum number of sessions **PROCESS_SESSION_LIMIT** process maximum number of sessions **RESET_SESSION_LIMIT** reset maximum number of sessions

[6] This verb is both in the LU 6.2 BASE set for remote support and in an OPTION set for local support.

Verbs in LU 6.2 OPTION Sets (Alphabetically within Purpose Groups)

The optional sets of LU 6.2 verbs that are provided by LU 6.2 option sets in various IBM LU 6.2 implementations are shown as follows, alphabetically within groups according to the general purpose of the verbs. It is these sets of verbs that may or may not be present in a particular implementation of LU 6.2. An implementation can incorporate one or more (or none) of the option sets for optional verbs and, hence, can support some, but not necessarily all, of the optional verbs. (See Appendix A, "Base and Option Sets for Product Support" on page A-1 for a detailed description of all LU 6.2 option sets.)

OPTIONAL Verbs for starting/stopping conversations	**MC_GET_ATTRIBUTES** determine conversation attributes[7] **GET_TYPE** determine conversation type **SET_SYNCPT_OPTIONS** setup SYNCPT options
OPTIONAL Verbs for using conversations	**(MC_)FLUSH** force buffer transmission[6] **(MC_)POST_ON_RECEIPT** suspend communication **(MC_)PREPARE_TO_RECEIVE** change from send to receive state[6] **(MC_)RECEIVE_EXPEDITED_DATA** receive expedited data, with waiting if specified **(MC_)RECEIVE_IMMEDIATE** try to receive data or info, without waiting **(MC_)SEND_EXPEDITED_DATA** send expedited data **(MC_)TEST** test for received data (on a particular conversation) **WAIT** wait for received data (on any conversation) **WAIT_FOR_COMPLETION** wait for nonblocking operation to complete (on any conversation)

[7] The GET_ATTRIBUTES verb is in the LU 6.2 BASE set of verbs; the MC_GET_ATTRIBUTES verb is not.

OPTIONAL Verbs for checking conversations	**BACKOUT** reverse/unconfirm syncpt **(MC_)PREPARE_FOR_SYNCPT** setup for syncpt **SYNCPT** request/confirm sysncpt
OPTIONAL Verbs for full-duplex conversations	**(MC_)ALLOCATE** start a conversation **(MC_)DEALLOCATE** close the local program's send queue **(MC_)FLUSH** force buffer transmission **(MC_)GET_ATTRIBUTES** determine conversation attributes **(MC_)RECEIVE_AND_WAIT** receive data, with waiting if necessary **(MC_)RECEIVE_EXPEDITED_DATA** receive expedited data, with waiting if specified **(MC_)RECEIVE_IMMEDIATE** try to receive data or info, without waiting **(MC_)SEND_DATA** send data **(MC_)SEND_ERROR** send error (message) **(MC_)SEND_EXPEDITED_DATA** send expedited data

OPTIONAL Verbs for managing the logical network	**ACTIVATE_SESSION** activate a session[6] **CHANGE_SESSION_LIMIT** change maximum number of sessions[6] **DEACTIVATE_CONVERSATION_GROUP** deactivate a session identified by a conversation group id **DEACTIVATE_SESSION** deactivate a session[6] **DEFINE_LOCAL_LU** define a local LU **DEFINE_MODE** define a session mode **DEFINE_REMOTE_LU** define a remote LU **DEFINE_TP** define a local TP **DELETE** delete a local LU **DISPLAY_LOCAL_LU** display a local LU **DISPLAY_MODE** display a session mode **DISPLAY_REMOTE_LU** display a remote LU **DISPLAY_SIGNED_ON_LIST** display the local "signed-on" list **DISPLAY_TP** display a local TP **PROCESS_SIGNOFF** process sign-off requests from the partner LU **SIGNOFF** remove one or more entries from the local "signed-on" lists and notify affected partner LUs if necessary

Nonblocking Support for LU 6.2 Conversation Verbs

Nonblocking support enables the transaction program to issue a conversation verb and obtain control back prior to the completion of the verb. The verb whose execution is left incomplete becomes an *outstanding* verb. A conversation can have more than one verb outstanding at a time. An outstanding verb operation is identified by an operating system construct, called a *wait object* in this book.

The transaction program can issue a nonblocking verb in one of the following ways: with confirmation or without confirmation. In the former, the transaction program specifies a nonnull wait object when issuing a nonblocking verb. The LU associates the nonblocking verb operation with the supplied wait object when the execution of the verb is suspended because required resources are not available immediately. The LU *posts* the wait object when the associated nonblocking verb operation has completed. The transaction program can learn the status of the operation by checking on the wait object.

In the case of no confirmation, the transaction program specifies a NULL wait object when issuing a nonblocking verb, and gives up the ability to monitor the status of the nonblocking verb operation. The NULL wait object is used to conserve the usage of wait objects and also to provide the transaction program with the flexibility to handle a sequence of related operations, such as large I/O operations (see Figure B-18 on page B-40 for an example).

Based on the resources required during their execution (e.g., buffers and sessions), the LU manages the outstanding operations by means of *queues*. Conversation verbs are generally grouped under different queues according to the resources they share, while some conversation verbs (such as GET_TP_PROPERTIES and GET_TYPE) that do not require resource allocation are independent of the queues. Similarly, sync point services verbs are grouped under a separate queue managed by the sync point manager (SPM). See Table 3-1 on page 3-10 and Table 3-2 on page 3-10 for the corresponding queues to which the half-duplex and full-duplex verbs, respectively, belong.

Multiple queues can be active at the same time for a given conversation, and conversations of different types can have different sets of queues. Queues that can be simultaneously active in the half-duplex conversations are:

- allocate queue
- send-receive queue
- expedited-send queue
- expedited-receive queue
- sync point queue

and those queues in the full-duplex conversations are:

- allocate queue
- send queue
- receive queue
- expedited-send queue
- expedited-receive queue

Table 3-1. Half-duplex Conversation Verbs and Their Associated Queues

Verbs	Queues
(MC_)ALLOCATE	allocate queue
(MC_)CONFIRM	send-receive queue
(MC_)CONFIRMED	send-receive queue
(MC_)DEALLOCATE	send-receive queue
(MC_)FLUSH	send-receive queue
(MC_)GET_ATTRIBUTES	--
(MC_)POST_ON_RECEIPT	--
(MC_)PREPARE_FOR_SYNCPT	sync point queue
(MC_)PREPARE_TO_RECEIVE	send-receive queue
(MC_)RECEIVE_AND_WAIT	send-receive queue
(MC_)RECEIVE_IMMEDIATE	send-receive queue
(MC_)RECEIVE_EXPEDITED_DATA	expedited-receive queue
(MC_)REQUEST_TO_SEND	expedited-send queue
(MC_)SEND_DATA	send-receive queue
(MC_)SEND_ERROR	send-receive queue
(MC_)SEND_EXPEDITED_DATA	expedited-send queue
(MC_)TEST	--
BACKOUT	sync point queue
SET_SYNCPT_OPTIONS	--
GET_TP_PROPERTIES	--
GET_TYPE	--
SYNCPT	sync point queue
WAIT	--
WAIT_FOR_COMPLETION	--

Table 3-2. Full-duplex Conversation Verbs and Their Associated Queues

Verbs	Queues
(MC_)ALLOCATE SYNC_LEVEL(NONE)	allocate queue
(MC_)DEALLOCATE TYPE(FLUSH or ABEND)	send queue
(MC_)FLUSH	send queue
(MC_)GET_ATTRIBUTES	--
(MC_)RECEIVE_AND_WAIT	receive queue
(MC_)RECEIVE_IMMEDIATE	receive queue
(MC_)RECEIVE_EXPEDITED_DATA	expedited-receive queue
(MC_)SEND_DATA	send queue
(MC_)SEND_ERROR	send queue
(MC_)SEND_EXPEDITED_DATA	expedited-send queue
GET_TP_PROPERTIES	--
GET_TYPE	--
WAIT_FOR_COMPLETION	--

Each queue can support at least one outstanding operation at any time, while its maximum capacity is unspecified (i.e., an implementation choice). Associated with each queue is a queuing discipline that determines the processing order of the verbs served by the queue. Queues managed by the LU are divided into two groups based on queuing discipline:

- Conversation initiation queues
- Conversation queues

Conversation queues process the requests from the transaction program in a *first-in, first-out (FIFO)* manner. In these queues, the completion of a verb implies the completion of all the verbs that were issued to the same queue prior to that verb. The sync point queue also falls in this category. On the other hand, verbs issued to a conversation initiation queue are served in the order they are invoked, but they can complete in a different order, a queuing discipline called *first-in, random-out (FIRO)*. Table 3-3 summarizes the characteristics of the queues.

Table 3-3. Characteristics of Queues

QUEUE GROUPS	QUEUING DISCIPLINE	SCOPE	QUEUES
conversation initiation queues	FIRO	each TP	allocate queue
conversation queues	FIFO	each half-duplex conversation	send-receive queue, expedited-send queue, expedited-receive queue
		each full-duplex conversation	send queue, receive queue, expedited-send queue, expedited-receive queue
sync point queues	FIFO	each SPM	sync point queue

Conversation queues are further divided into two subgroups, normal-flow queues and expedited-flow queues, based on the data exchanged over the requests. The normal-flow queues covering:

- send-receive queue (half-duplex conversation)
- send queue (full-duplex conversation)
- receive queue (full-duplex conversation)

are used to convey normal-flow user data. On the other hand, the expedited-flow queues are used to convey expedited-flow data of which the processing takes precedence over the normal-flow data. The expedited-flow queues are:

- expedited-send queue
- expedited-receive queue

With both the expedited-flow and normal-flow conversation queues, implicit correlations exist within each subgroup of queues residing at the two ends of the conversation. The user data sent from the local program's send queue will be received by the remote program's receive queue. The same correlation applies to the expedited-flow conversation queues where the expedited-flow is handled. These correlations,

illustrated in Figure 3-2 on page 3-12 and Figure 3-3 on page 3-12, can serve as basic building blocks for the transaction programs. However, the transaction programs can freely exchange information in any agreeable manner through these queues.

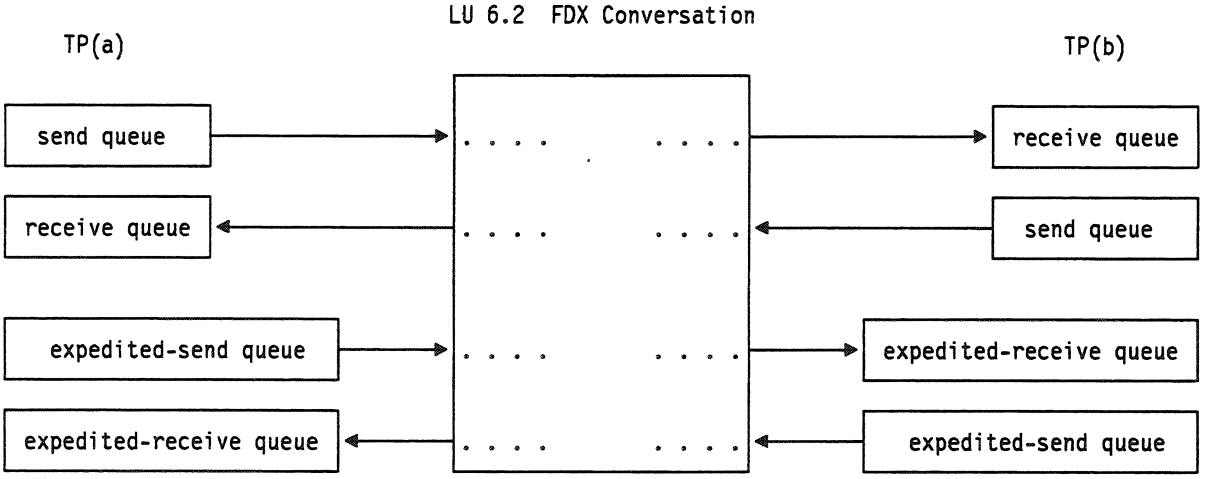


Figure 3-2. Correlation between Full-duplex Conversation Queues

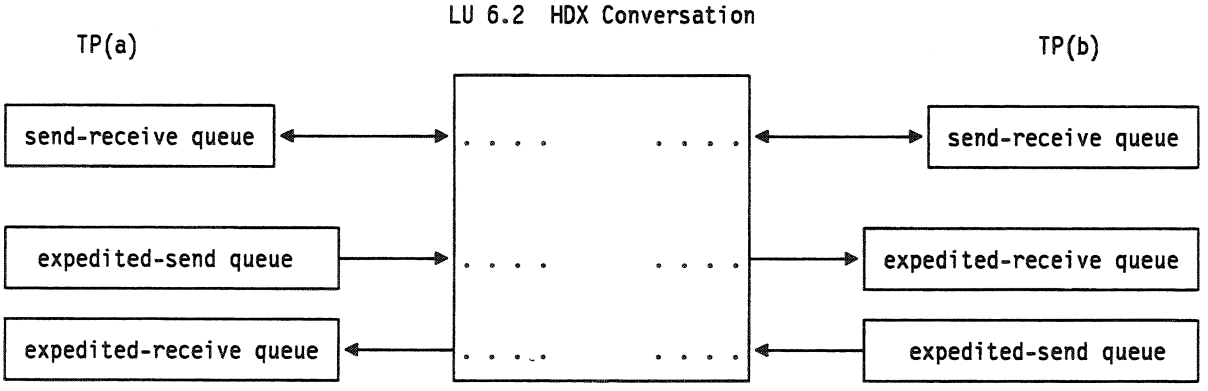


Figure 3-3. Correlation between Half-duplex Conversation Queues

Chapter 4. Verb Description Format

This chapter explains the format used to describe the verbs found in Part 2, "Conversation Verbs" and in Part 3, "Control Operator Verbs" of this book.

For every LU 6.2 verb the following information is presented:

Verb Function	What the verb does
Verb Syntax	What the verb parameters and parameter values are
Supplied Parameters	Which parameters are supplied to the LU
Supplied-and-Returned Parameters	Which parameters are supplied to and returned from the LU
Returned Parameters	Which parameters are returned from the LU
State Changes	What conversation state changes are possible
Notes (on Verb Usage)	What extra things you should know about this verb

The above components of the verb description format are described further in the following sections.

Verb Function

The description of each verb begins with a brief explanation of its function. If the verb is provided only by an option set, a highlighted sentence in parentheses follows the verb function description and identifies the option set number that provides the verb.

Verb Syntax

The verb's syntax is described next using a format box. The general representation of the format box is shown in Table 4-1 on page 4-2 and is explained in the sections following the figure.

Table 4-1. Format for Representing LU 6.2 Verb Syntax

VERB_NAME	*Option No.* [*Base/nnn*]	*Half-duplex* [√]	*Full-duplex* [√]
Supplied Parameters: **PARAMETER** (*variable*) **PARAMETER** *(default-value)* *(optional-value)* **PARAMETER** (*value, variable, ...*) **PARAMETER** *(SUBPARAMETER1)* *(SUBPARAMETER2 (variable))* Optional Series of Supplied Parameters **PARAMETER** *(default-value)* *(optional-value)* **PARAMETER** (...)		▌ nnn	
Supplied-and-Returned Parameters: **PARAMETER** *any of the above patterns*			
Returned Parameters: **PARAMETER** (*variable*) Optional Series of Returned Parameters **PARAMETER** (*variable*) **PARAMETER** (...)		▌ nnn	

As shown in the preceding general format box, the syntax description for each verb includes a verb name, half-duplex support, full-duplex support, option number (or an indication that it is a base function), verb parameters, and verb parameter values. The number of verb parameters varies for each verb. The number of verb parameters implemented by an LU 6.2 product depends on the number of option sets implemented.

Parameters and Parameter Values

Parameter names are shown as uppercase keywords. Parameter values (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.

Parameter Groups

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.

Required and Optional Parameters

Not all verb parameters are required; some of the parameters have default values and some of the parameters are available only when a particular LU 6.2 option set has been incorporated into the local LU 6.2 implementation. Default parameter values are shown in boldface and parameters available only with particular LU 6.2 option sets are followed by a phrase "▌ nnn" to the right of the parameter.

The supplied parameters that the LU 6.2 transaction programmer must code (that is, the required parameters) are listed first. The supplied parameters that may be omitted (that is, the optional parameters) are listed last, if there are any such parameters. The required and optional parameters are separated by a short, thick line from the left margin of the verb syntax table as shown in Table 4-1 on page 4-2. Stated differently, all parameters below the short, thick, horizontal line in a given parameter group are conditionally present; those above are always present.

The returned parameters are likewise separated into the required and optional parameters. Product publications that reference this book can use the required and optional separations to determine which parameters must be referenced in their publications. For example, a product that has implemented just the LU 6.2 base set of verbs and no option sets is required to reference just the required supplied and returned parameters. If, however, another product has implemented option set #108, for example, then the product must reference all of the verbs, verb parameters, and verb parameter values affected by this option set in addition to the required parameters.

The following rules apply:

Omission of any *supplied parameter that has a default value* is treated as if the default argument were specified on the parameter.

Omission of any *supplied parameter that is available only with an LU 6.2 option set* is treated as is described for the parameter in the parameter description following the verb.

Omission of any *returned parameter that is available only with an LU 6.2 option set* is treated as is described for the parameter in the parameter description following the verb. If the option set is not implemented, then no parameter value is returned.

Supplied Parameters

All of the parameters that can be given by the transaction program to the LU are listed immediately after the verb syntax.

Supplied-and-Returned Parameters

All of the parameters that can be given by the transaction program to the LU and that are also returned by the LU to the transaction program are listed, when there are such, following the supplied parameters.

Returned Parameters

All of the parameters that can be returned by the LU to the transaction program are listed immediately after the supplied parameters (that is, after those that can be given to the LU) or immediately after the supplied-and-returned parameters when there are such.

Included is a list of the return codes that can be returned to the transaction program when it resumes processing.

Return Codes and What-Received Indications

Each LU, both the local LU and the remote LU, responds to the verb from the transaction program with return codes and what-received indications as shown in Figure 4-1 on page 4-5. (See Appendix F, "Conversation Return Codes" on page F-1, Appendix G, "Control-Operator Return Codes" on page G-1, and "Conversation Return Codes and What-Received Indications" on page A-39 for detailed descriptions of return codes and what-received indications.)

Return codes are given by LU 6.2 to every LU 6.2 verb; what-received indications are given by LU 6.2 to the "receive" verbs, RECEIVE_AND_WAIT and RECEIVE_IMMEDIATE. Data, of course, is also given by LU 6.2 to the transaction program by the "receive" verbs when data is available.

Return codes and what-received indications are given by the *local* LU to the *local* TP and by the *remote* LU to the *remote* TP; these are not interchanged between local and remote locations.

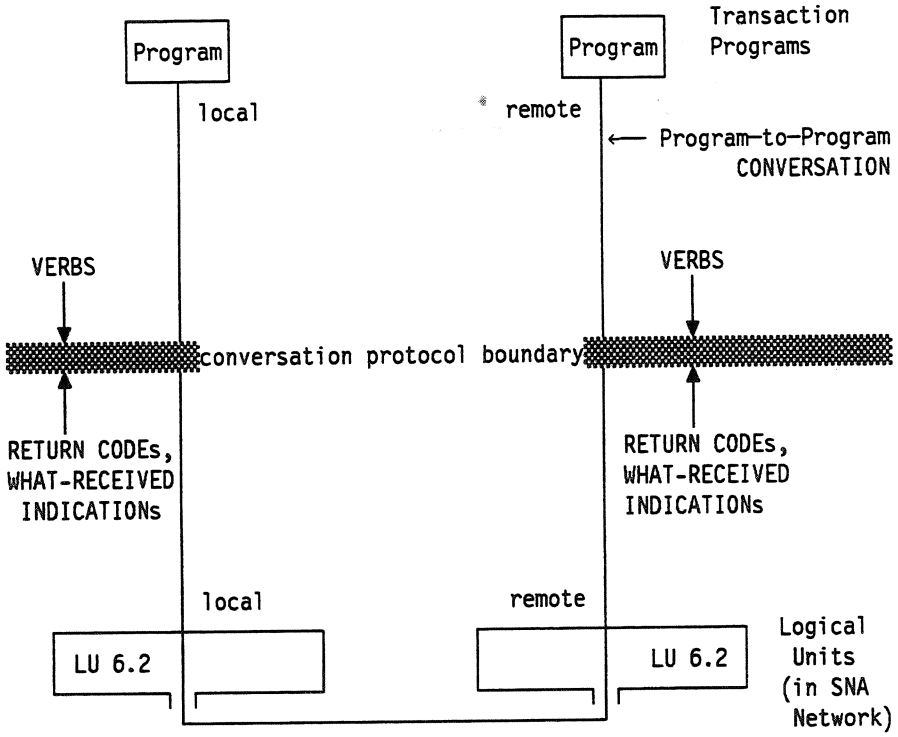


Figure 4-1. Verbs and Return Codes and What-Received Indications

Programming Error Conditions

Invalid specification or execution of a verb can cause error conditions to arise. In the LU 6.2 architecture, an error is reported to the program in the RETURN_CODE parameter as either PROGRAM_STATE_CHECK or PROGRAM_PARAMETER_CHECK.

Program 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. 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.

Program 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 Appendix E, "Conversation State Matrices" on page E-1 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.

See Appendix F, "Conversation Return Codes" on page F-1 and Appendix G, "Control-Operator Return Codes" on page G-1 for a description of these programming error return codes.

It is possible, however, for IBM products that implement LU 6.2 to choose to abnormally terminate or abend the program in lieu of providing these programming error return codes as explained above.

State Changes

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.

Also, Appendix E, "Conversation State Matrices" contains tables showing all the state changes.

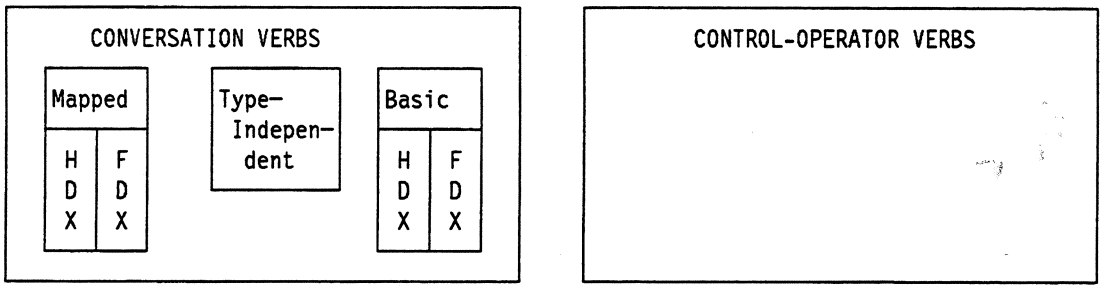
Notes

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

Notes on the Verb Description Format:

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 supplied and returned parameters listed in this book are unrelated to any product's API definition. Refer to the product's programming publications for details of its API.
3. For products that implement *LU 6.2 semantics (functions)* with the IBM Systems Application Architecture: Common Programming Interface Communications *(SAA CPI-C) syntax (format)*, you should also refer to *Systems Application Architecture: Common Programming Interface Communications Reference* for the CPI-C program CALL statements related to the LU 6.2 verbs.

Part 2. Conversation Verbs



This part describes the category of verbs called *conversation verbs*.[1] The conversation verbs define the LU 6.2 conversation protocol boundary. These verbs are used for program-to-program communication over a conversation connecting two programs. Each conversation is of a specific type:

Mapped (Half-Duplex)
Basic (Half-Duplex)
Full-Duplex Mapped
Full-Duplex Basic

The conversation verbs are divided into subcategories based on the conversation type. Both the mapped and basic subcategories include both half-duplex and full-duplex data transfer protocols. The subcategories of verbs are:

- Mapped conversation verbs

  half-duplex data transfer
  full-duplex data transfer

- Type-independent conversation verbs
- Basic conversation verbs

  half-duplex data transfer
  full-duplex data transfer

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 LU 6.2 Reference: Peer Protocols* for a description of the mapped conversation LU services component.

See Appendix E, "Conversation State Matrices" on page E-1 for a description of which verbs can be issued in each conversation state. You should study the conversation transition matrix, paying particular attention to the footnotes, before you begin using LU 6.2 verbs. This preparation will help you understand how verbs and conversations are controlled by the LU through conversation states, with one state

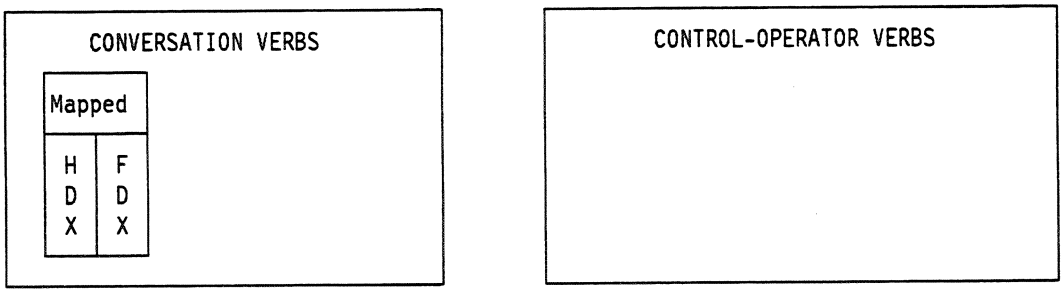
[1] All LU 6.2 verbs are described, generally, in Chapter 2, "Verb Categories" on page 2-1.

for each conversation that the TP has allocated (thus relieving the program from having to do this conversation control).

See Appendix F, "Conversation Return Codes" on page F-1 for a list of return codes that apply to the conversation verbs.

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

Chapter 5. Mapped Conversation Verbs



This chapter 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 mapped conversation between the local transaction program and a remote (partner) transaction program to a session between the local LU and the remote LU. This means that the local and remote programs have exclusive use of the session for the duration of the conversation. 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	*Option No.* [*Base*]	*Half-duplex* [√]	*Full-duplex* [√]
Supplied Parameters:			
LU_NAME (*variable*)			
MODE_NAME (*variable*)			
TPN (*variable*)			
TYPE *(HALF_DUPLEX_CONVERSATION)*			
(FULL_DUPLEX_CONVERSATION)		*112*	
RETURN_CONTROL			
(WHEN_SESSION_ALLOCATED)			
(IMMEDIATE)		*203*	
(WHEN_CONWINNER_ALLOCATED)		*201*	
(WHEN_CONVERSATION_GROUP_ALLOCATED)		*401*	
(WHEN_SESSION_FREE)		*205*	
CONVERSATION_GROUP_ID (*variable*)		*401*	
SYNC_LEVEL			
(NONE)			
(CONFIRM)			
(SYNCPT)		*108*	
SECURITY			
(NONE)			
(SAME)		*212,217,219*	
(PGM (USER_ID (variable))		*213*	
(PASSWORD (variable))		*213*	
(PROFILE (variable)))		*218*	
PIP (*NO*)			
(YES (variable1 variable2 ... variableN))		*241*	
WAIT_OBJECT			
(BLOCKING)			
(VALUE (variable))		*113*	
Returned Parameters:			
RETURN_CODE (*variable*)			
RESOURCE (*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. See options 606 and 607 for locally known and uninterpreted LU names.

- Variable specifies the name of the LU (local or remote) where the partner transaction program is located. (See Option#204 for the special *intra-LU case* where the local TP and the partner TP are located at the same LU.)

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 before it is sent. The SNA-defined mode names, SNASVCMG and CPSVCMG, are not allowed for mapped conversations. See the ALLOCATE verb. See "DEFINE_MODE" on page 10-9 for more information on the mode name.

TPN specifies the name of the remote transaction program to be connected at the other end of the mapped conversation. Mapped conversations cannot be established to SNA service 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.

- **HALF_DUPLEX_CONVERSATION** specifies to allocate a half-duplex mapped conversation.
- **FULL_DUPLEX_CONVERSATION** specifies to allocate a full-duplex mapped conversation.

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 on this verb. 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.
- **IMMEDIATE** specifies to allocate a session for the mapped conversation if a session is immediately available. 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. If a session cannot be activated, then control reverts to the program with a return code of UNSUCCESSFUL.
- **WHEN_CONWINNER_ALLOCATED** specifies to allocate a contention-winner session for the mapped conversation before returning control to the program.
- **WHEN_CONVERSATION_GROUP_ALLOCATED** specifies to allocate the session having the specified conversation group ID for the mapped conversation before returning control to the program.

- The LU_NAME parameter is omitted, it is implied.
- The MODE_NAME parameter is omitted, it is implied.
- The CONVERSATION_GROUP_ID parameter must be specified.

- **WHEN_SESSION_FREE** specifies to allocate a session for the mapped conversation if a session is available. If no session is available, the local program is willing to wait for session activation. If a session cannot be activated and there are no outstanding session activation requests to satisfy this MC_ALLOCATE, then control returns back to the program with a return code of ALLOCATION_FAILURE_RETRY.

Table 5-1. RETURN_CONTROL Parameters and Conditions for Return of Control to the (MC_)ALLOCATE Verb

	Con-Winner Session Acceptable?	**Con-Loser Session Acceptable?**	**Wait Expected?**
WHEN_SESSION_ ALLOCATED	Yes	Yes	Yes, indefinitely
IMMEDIATE	Yes	No	No
WHEN_CONWINNER_ ALLOCATED	Yes	No	Yes
WHEN_ CONVERSATION_ GROUP_ALLOCATED	A particular session is required		Yes, but only if conversation group is active
WHEN_SESSION_FREE	Yes	Yes	Yes, while there are outstanding session activation requests

CONVERSATION_GROUP_ID specifies the value of the conversation group ID that identifies a particular session when the RETURN_CONTROL parameter specifies WHEN_CONVERSATION_GROUP_ALLOCATED. When the RETURN_CONTROL parameter specifies a value other than WHEN_CONVERSATION_GROUP_ALLOCATED, this parameter is not used and is ignored.

The CONVERSATION_GROUP_ID is the identifier of a particular session between two particular LUs and provides a mechanism whereby one or more pairs of transaction programs (that is, one group of TPs at one LU and the other group of TPs at the other LU) can serially share the same identified session as if all the TPs were in a *conversation group* of intercommunicating TPs taking turns with their conversations over the same session. The conversation group ID is created by the LU and can be retrieved, when option 401 is installed, with the MC_GET_ATTRIBUTES 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 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 relating to confirmation, but they will not issue any verbs relating to sync point.

  SYNC_LEVEL(CONFIRM) is not supported for full-duplex conversations.
- **SYNCPT** specifies that the transaction programs can perform both confirmation and sync-point processing on this mapped conversation. The programs may issue verbs 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 send access security information which will cause the remote LU to create an equivalent security environment. If the remote LU trusts the local LU to verify user IDs and passwords, then the user ID and profile (if present) currently associated with the executing program is used. The access security information from the allocation request that initiates execution of a local program is used to initially set the associated user ID and profile values. The user ID is indicated as being already verified. If the remote LU does not trust the local LU to verify user IDs and passwords, or the local program does not have a user ID and profile associated with it, then the local LU, in conjunction with a local security manager, can attempt to determine user ID, password and profile value to send to the remote LU. This determination is implementation specific. If values cannot be determined, 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:

  Security PGM parameter notes:

  1. The USER_ID and PASSWORD parameters must always be specified as a pair if either is to be specified.
  2. Specifying a null value for any of the access security arguments is equivalent to omitting the argument.

  – **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.

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 and sequence of PIP subfields must agree with the number and sequence of PIP variables specified on the remote program's PROC statement (see "Transaction Program Structure and Execution" for a description of transaction program structure).

WAIT_OBJECT specifies whether the verb should be processed in a blocking or nonblocking manner. The default value is a blocking verb operation. If a nonblocking verb operation is to be requested, then WAIT_OBJECT(VALUE(variable)) must be specified.

- **BLOCKING** specifies this is a blocking verb operation.
- **VALUE(*variable*)** specifies a nonblocking verb operation of one of two types:
  - a ***null*** variable specifies to complete the verb operation but without notification.
  - a nonnull ***variable*** specifies to complete the verb operation but with notification. The variable represents a wait object. Its use here associates it with completion of this verb. It can be listed as an entry in the WAIT_OBJECT_LIST in a subsequent WAIT_FOR_COMPLETION verb for notification (posting) purposes. Origination of this variable (by the TP or the operating system) is implementation dependent.

Returned Parameters:

RETURN_CODE specifies the variable in which the result of the verb execution is indicated to the local program.

The RETURN_CONTROL parameter affects the return codes that can be returned to the program. All RETURN_CONTROL parameters share the following return codes:

- OK
- ALLOCATION_ERROR, with one of the following subcodes (Refer to Appendix F, "Conversation Return Codes" on page F-1 for a complete list of subcodes with their explanations):
  - SYNC_LEVEL_NOT_SUPPORTED_BY_LU
- OPERATION_INCOMPLETE
- OPERATION_NOT_ACCEPTED
- PROGRAM_PARAMETER_CHECK, for one of the following reasons:
  - The program does not have mapped conversation support defined.
  - MODE_NAME specifies the SNA-defined mode name, SNASVCMG or CPSVCMG.
  - SYNCPT specified and is not supported.
  - TPN specifies an SNA service transaction program.
  - TPN specifies a null (zero length) value.
  - The RETURN_CONTROL value specified is not supported.
  - USER_ID and PASSWORD are not specified together.

Additional return codes are possible depending on the RETURN_CONTROL value:

- RETURN_CONTROL(WHEN_SESSION_ALLOCATED) or RETURN_CONTROL(WHEN_CONWINNER_ALLOCATED)
  - ALLOCATION_ERROR, with one of the following subcodes (Refer to Appendix F, "Conversation Return Codes" on page F-1 for a complete list of subcodes with their explanations):
    - ALLOCATION_FAILURE_NO_RETRY
    - ALLOCATION_FAILURE_RETRY
  - PARAMETER_ERROR, for one of the following reasons:
    - Invalid LU name
    - Invalid mode name
- RETURN_CONTROL(IMMEDIATE)
  - PARAMETER_ERROR, for one of the following reasons:
    - Invalid LU name
    - Invalid mode name
  - UNSUCCESSFUL, for the following reason:
    - Session not immediately available
- RETURN_CONTROL(WHEN_CONVERSATION_GROUP_ALLOCATED)
  - ALLOCATION_ERROR, with one of the following subcodes (Refer to Appendix F, "Conversation Return Codes" on page F-1 for a complete list of subcodes with their explanations):
    - ALLOCATION_FAILURE_NO_RETRY
- RETURN_CONTROL(WHEN_SESSION_FREE)
  - ALLOCATION_ERROR, with one of the following subcodes (Refer to Appendix F, "Conversation Return Codes" on page F-1 for a complete list of subcodes with their explanations):
    - ALLOCATION_FAILURE_NO_RETRY
    - ALLOCATION_FAILURE_RETRY

Return codes for half-duplex conversations only:

The RETURN_CONTROL parameter affects the return codes that can be returned to the program. All RETURN_CONTROL parameters share the following return codes:

- PROGRAM_STATE_CHECK
  - New protected mapped conversations may not be allocated when the program is in the backout-required state on other conversations.

Return codes for full-duplex conversations only:

- ALLOCATION_ERROR, with the following subcode (Refer to Appendix F, "Conversation Return Codes" on page F-1 for a complete list of subcodes with their explanations):
  - FULL_DUPLEX_NOT_SUPPORTED_BY_LU
- PROGRAM_PARAMETER_CHECK, for one of the following reasons:
  - SYNC_LEVEL(CONFIRM) is not supported for full-duplex conversations.
  - A full-duplex conversation is specified but is not supported.

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.

State Changes:

Half-duplex conversations only:

Send state is entered.

Full-duplex conversations only:

Send-receive state is entered.

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(WHEN_CONVERSATION_GROUP_ALLOCATED), and no active session exists for the conversation group ID specified on the CONVERSATION_GROUP_ID parameter, a return code of ALLOCATION_FAILURE_NO_RETRY is returned to the program. This error can occur, for example, when the conversation group was active at the time the MC_GET_ATTRIBUTES verb was issued, but before the LU could allocate the mapped conversation on the session the session ended (say, because of a session outage). This error can also occur when the conversation group ID is invalid.
5. 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 may attempt to determine user ID, profile and password values to send to the remote LU. If the local LU is unable to determine access security information to send it will downgrade the specification to SECURITY(NONE). When the remote LU does accept security parameters, the TP at the remote LU can obtain the user ID and profile received in the allocation request by issuing the GET_TP_PROPERTIES verb.
6. 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.
7. For programs that use multiple conversations, a request for RETURN_CONTROL(IMMEDIATE) or RETURN_CONTROL(WHEN_SESSION_FREE) should be used; the remaining RETURN_CONTROL parameters may cause the conversation allocation request to be queued indefinitely.
8. References in this verb description to a program being in a particular state are only in terms of the allocated mapped conversation.

Half-duplex conversations only:

1. SYNC_LEVEL(SYNCPT) permits use of the SYNCPT and BACKOUT verbs to aid in maintaining consistency across all protected resources within a distributed logical unit of work. See *SNA LU 6.2 Reference: Peer Protocols* for more details of sync point.
2. Upon successful completion of the MC_ALLOCATE, the local program is in send state and the remote program is in receive state.

Full-duplex conversations only:

1. Both the local and remote programs are connected in send-receive state.

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 interactively communicate during a particular time period for distributed processing between the two programs. The LU flushes its send buffer as a function of this verb.

MC_CONFIRM	*Option No.* [*Base*]	*Half-duplex* [√]	*Full-duplex* [ ]
Supplied Parameters:			
RESOURCE (*variable*)			
WAIT_OBJECT			
(BLOCKING)			
(VALUE (variable))		*113*	
Returned Parameters:			
RETURN_CODE (*variable*)			
REQUEST_TO_SEND_RECEIVED (*variable*)			
EXPEDITED_DATA_RECEIVED (*variable*)		*112*	

Supplied Parameters:

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

WAIT_OBJECT specifies whether the verb should be processed in a blocking or nonblocking manner. The default value is a blocking verb operation. If a nonblocking verb operation is to be requested, then WAIT_OBJECT(VALUE(variable)) must be specified.

- **BLOCKING** specifies this is a blocking verb operation.
- VALUE(***variable***) specifies a nonblocking verb operation of one of two types:
  - a ***null*** variable specifies to complete the verb operation but without notification.
  - a nonnull ***variable*** specifies to complete the verb operation but with notification. The variable represents a wait object. Its use here associates it with completion of this verb. It can be listed as an entry in the WAIT_OBJECT_LIST in a subsequent WAIT_FOR_COMPLETION verb for notification (posting) purposes. Origination of this variable (by the TP or the operating system) is implementation dependent.

Returned Parameters:

RETURN_CODE specifies the variable in which the result of the verb execution is indicated to the local program.

- OK (remote program replied MC_CONFIRMED)
- ALLOCATION_ERROR
- BACKED_OUT
- DEALLOCATE_ABEND
- MAPPING_NOT_SUPPORTED
- MAP_NOT_FOUND
- MAP_EXECUTION_FAILURE
- OPERATION_INCOMPLETE
- OPERATION_NOT_ACCEPTED
- PROG_ERROR_PURGING
- PROGRAM_PARAMETER_CHECK, for one of the following reasons:
  - This mapped conversation was allocated with SYNC_LEVEL(NONE).
  - RESOURCE specifies an unassigned resource ID.
  - This verb is not supported for full-duplex conversations.
- PROGRAM_STATE_CHECK, for the following reason:
  - The mapped conversation is not in send or defer-receive state. See Note 2 below.
- RESOURCE_FAILURE_NO_RETRY
- RESOURCE_FAILURE_RETRY
- USER_CONTROL_DATA_NOT_SUPPORTED

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received.

- 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.

EXPEDITED_DATA_RECEIVED specifies the variable in which is returned an indication of whether SEND_EXPEDITED_DATA has been received.

- YES indicates a SEND_EXPEDITED_DATA notification has been received from the remote transaction program.
- NO indicates a SEND_EXPEDITED_DATA notification has not been received from the remote transaction program.

State Changes:

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

No state change occurs when the verb is issued in send 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. This verb cannot be issued in defer-deallocate state, a state entered when MC_DEALLOCATE TYPE(SYNC_LEVEL) is issued when the synchronization level for the mapped conversation is SYNCPT.
3. 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).
4. No state change has occurred because EXPEDITED_DATA_RECEIVED indicates YES.

   When expedited data is received by the LU, it is indicated on the pertinent verbs if the return parameter EXPEDITED_DATA_RECEIVED exists on the verbs. It will continue to be indicated until such time that the data is received by the TP issuing the MC_RECEIVE_EXPEDITED_DATA verb.
5. If a session becomes unavailable just prior to the issuance of the MC_CONFIRMED verb (in response to the MC_CONFIRM verb from the partner program), then the program issuing the MC_CONFIRMED verb will not receive an indication of a failure until another verb is issued following the MC_CONFIRMED verb.
6. 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 interactively communicate during a particular time period for distributed processing between the two programs. 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	*Option No.* [*Base*]	*Half-duplex* [√]	*Full-duplex* [ ]
Supplied Parameters:			
RESOURCE (*variable*)			
WAIT_OBJECT			
(BLOCKING)			
(VALUE (variable))		*113*	
Returned Parameters:			
RETURN_CODE (*variable*)			

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID of the mapped conversation being used.

WAIT_OBJECT specifies whether the verb should be processed in a blocking or nonblocking manner. The default value is a blocking verb operation. If a nonblocking verb operation is to be requested, then WAIT_OBJECT(VALUE(variable)) must be specified.

- **BLOCKING** specifies this is a blocking verb operation.
- **VALUE(*variable*)** specifies a nonblocking verb operation of one of two types:
  - a ***null*** variable specifies to complete the verb operation but without notification.
  - a nonnull ***variable*** specifies to complete the verb operation but with notification. The variable represents a wait object. Its use here associates it with completion of this verb. It can be listed as an entry in the WAIT_OBJECT_LIST in a subsequent WAIT_FOR_COMPLETION verb for notification (posting) purposes. Origination of this variable (by the TP or the operating system) is implementation dependent.

Returned Parameters:

RETURN_CODE specifies the variable in which the result of the verb execution is indicated to the local program.

- OK
- OPERATION_INCOMPLETE
- OPERATION_NOT_ACCEPTED
- PROGRAM_PARAMETER_CHECK, for one of the following reasons:
  - RESOURCE specifies an unassigned resource ID.
  - This verb is not supported for full-duplex conversations.
- PROGRAM_STATE_CHECK, for the following reason:
  - The mapped conversation is not in confirm, confirm-send, or confirm-deallocate state.

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.

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, or by issuing MC_SEND_ERROR as an indication that it encountered an error.
3. If a session becomes unavailable just prior to the issuance of the MC_CONFIRMED verb (in response to the MC_CONFIRM verb from the partner program), then the program issuing the MC_CONFIRMED verb will not receive an indication of a failure until another verb is issued following the MC_CONFIRMED verb.
4. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

MC_DEALLOCATE

For half-duplex conversations:

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 a 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.

For full-duplex conversations:

MC_DEALLOCATE with TYPE(FLUSH) closes the local program's send queue. Both the local and remote program must close their send queues independently; therefore, two MC_DEALLOCATE TYPE(FLUSH) verbs are required to end a full-duplex conversation. Notification that the partner has closed its send queue is given to the receive queue in the form of a DEALLOCATE_NORMAL return code.

MC_DEALLOCATE with TYPE(ABEND) is an abrupt termination that will close both sides of the conversation simultaneously. This notification is returned to the remote program's send queue as an ERROR_INDICATION return code, and to the remote program's receive queue as a DEALLOCATE_ABEND return code.

MC_DEALLOCATE	*Option No.* [*Base*]	*Half-duplex* [√]	*Full-duplex* [√]
Supplied Parameters:			
RESOURCE (*variable*)			
TYPE (*SYNC_LEVEL*)			
(FLUSH)			
(CONFIRM)			
(ABEND)			
(LOCAL)			
WAIT_OBJECT			
(BLOCKING)			
(VALUE (variable))		*113*	
Returned Parameters:			
RETURN_CODE (*variable*)			
EXPEDITED_DATA_RECEIVED (*variable*)		*112*	

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 SYNCPT. If the SYNCPT is successful (as indicated by a return code of OK on that verb), the mapped conversation is then deallocated normally; otherwise, the state of the mapped conversation is determined by the return code. See the SYNCPT verb for further discussion of state changes. If the program issues BACKOUT, the conversation is not deallocated.
- **FLUSH** specifies to execute the function of the MC_FLUSH verb and deallocate the mapped conversation normally.

  TYPE(FLUSH) is not permitted on mapped conversations allocated with SYNC_LEVEL(SYNCPT).
- **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.

  TYPE(CONFIRM) is not permitted on mapped conversations allocated with SYNC_LEVEL(SYNCPT), and is not supported for full-duplex conversations.
- **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 Appendix F, "Conversation Return Codes" on page F-1 and Note 5 on page 5-21).

  TYPE(LOCAL) is not supported for full-duplex conversations.

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.

WAIT_OBJECT specifies whether the verb should be processed in a blocking or nonblocking manner. The default value is a blocking verb operation. If a nonblocking verb operation is to be requested, then WAIT_OBJECT(VALUE(variable)) must be specified.

- **BLOCKING** specifies this is a blocking verb operation.
- **VALUE(*variable*)** specifies a nonblocking verb operation of one of two types:
  - a ***null*** variable specifies to complete the verb operation but without notification.
  - a nonnull ***variable*** specifies to complete the verb operation but with notification. The variable represents a wait object. Its use here associates it with completion of this verb. It can be listed as an entry in the WAIT_OBJECT_LIST in a subsequent WAIT_FOR_COMPLETION verb for notification (posting) purposes. Origination of this variable (by the TP or the operating system) is implementation dependent.

Returned Parameters:

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

- A flush-level deallocate can be issued in two ways: with TYPE(SYNC_LEVEL) specified when the synchronization level allocated to this mapped conversation is NONE, or with TYPE(FLUSH) specified when the synchronization level allocated to this mapped conversation is not SYNCPT. If one of these is issued, one of the following return codes is returned:
  - OK (deallocation is complete)
  - OPERATION_INCOMPLETE
  - OPERATION_NOT_ACCEPTED
  - PROGRAM_PARAMETER_CHECK, for one of the following reasons:
    - RESOURCE specifies an unassigned resource ID.
- If TYPE(CONFIRM) is specified and the synchronization level allocated to this mapped conversation is NONE, (for full-duplex conversations a synchronization level of CONFIRM is not allowed) one of the following return codes is returned:
  - PROGRAM_PARAMETER_CHECK, for one of the following reasons:
    - RESOURCE specifies an unassigned resource ID.
    - TYPE(CONFIRM) is specified and the mapped conversation is allocated with SYNC_LEVEL(NONE).
- If TYPE(ABEND) is specified, one of the following return codes is returned:
  - OK (deallocation is complete)
  - OPERATION_INCOMPLETE
  - OPERATION_NOT_ACCEPTED
  - PROGRAM_PARAMETER_CHECK, for one of the following reasons:
    - RESOURCE specifies an unassigned resource ID.

Return codes for half-duplex conversations only:

- A confirm-level deallocate can be issued in two ways: with TYPE(SYNC_LEVEL) or with TYPE(CONFIRM) specified; in both cases the synchronization level allocated to this mapped conversation must be CONFIRM. If one of these is issued, one of the following return codes is returned:
  - OK (deallocation is complete)
  - ALLOCATION_ERROR
  - DEALLOCATE_ABEND
  - MAPPING_NOT_SUPPORTED
  - MAP_NOT_FOUND
  - MAP_EXECUTION_FAILURE
  - OPERATION_INCOMPLETE
  - OPERATION_NOT_ACCEPTED
  - PROG_ERROR_PURGING
  - PROGRAM_PARAMETER_CHECK, for one of the following reasons:
    - RESOURCE specifies an unassigned resource ID.
  - PROGRAM_STATE_CHECK, for the following reason:
    - The mapped conversation is not in send state.
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY
  - USER_CONTROL_DATA_NOT_SUPPORTED
- If TYPE(SYNC_LEVEL) is specified and the synchronization level allocated to this mapped conversation is SYNCPT, one of the following return codes is returned:
  - OK (deallocation is deferred)
  - OPERATION_INCOMPLETE
  - OPERATION_NOT_ACCEPTED
  - PROGRAM_PARAMETER_CHECK, for the following reason:
    - RESOURCE specifies an unassigned resource ID.
  - PROGRAM_STATE_CHECK, for the following reason:
    - The mapped conversation is not in send state.
- If TYPE(CONFIRM) or TYPE(FLUSH) is specified and the synchronization level allocated to this mapped conversation is SYNCPT, the following return code is returned:
  - PROGRAM_PARAMETER_CHECK, for one of the following reasons:
    - RESOURCE specifies an unassigned resource ID.
    - TYPE(CONFIRM) or TYPE(FLUSH) is specified and the mapped conversation is allocated with SYNC_LEVEL(SYNCPT).
- If TYPE(ABEND) is specified, one of the following return codes is returned:
  - PROGRAM_STATE_CHECK, for one of the following reasons:
    - The mapped conversation is in deallocate state.
- If TYPE(LOCAL) is specified, one of the following return codes is returned:
  - OK (deallocation is complete)
  - PROGRAM_PARAMETER_CHECK, for one of the following reasons:
    - RESOURCE specifies an unassigned resource ID.
  - PROGRAM_STATE_CHECK, for one of the following reasons:
    - The mapped conversation is not in deallocate state.

Return codes for full-duplex conversations only:

- A flush-level deallocate can be issued in two ways: with either TYPE(SYNC_LEVEL) or TYPE(FLUSH) specified. If one of these is issued, one of the following return codes is returned:
  - ERROR_INDICATION, with one of the following subcodes (Refer to Appendix F, "Conversation Return Codes" on page F-1 for a complete list of subcodes with their explanations):
    - ALLOCATION_ERROR_PENDING
    - DEALLOCATE_ABEND_PENDING
    - RESOURCE_FAILURE_NO_RETRY_PENDING
    - RESOURCE_FAILURE_RETRY_PENDING
    - UNKNOWN_ERROR_TYPE_PENDING
  - PROGRAM_STATE_CHECK, for the following reason:
    - The conversation is not in send-receive or send-only state.
- If TYPE(CONFIRM) is specified, the following return code is returned:
  - PROGRAM_PARAMETER_CHECK, for the following reason:
    - TYPE(CONFIRM) is not supported for full-duplex conversations.
- If TYPE(SYNC_LEVEL) is specified and the synchronization level allocated to this mapped conversation is SYNCPT, the following return code is returned:
  - PROGRAM_PARAMETER_CHECK, for the following reason:
    - SYNC_LEVEL(SYNCPT) is not supported for full-duplex conversations.
- If TYPE(LOCAL) is specified, the following return code is returned:
  - PROGRAM_PARAMETER_CHECK, for the following reason:
    - TYPE(LOCAL) is not supported for full-duplex conversations.

EXPEDITED_DATA_RECEIVED specifies the variable in which is returned an indication of whether SEND_EXPEDITED_DATA has been received.

- YES indicates a SEND_EXPEDITED_DATA notification has been received from the remote transaction program.
- NO indicates a SEND_EXPEDITED_DATA notification has not been received from the remote transaction program.

The EXPEDITED_DATA_RECEIVED variable is not set when the MC_DEALLOCATE causes the conversation state to change to reset. The variable is set only if all of the following conditions hold:

- The conversation type (specified via the TYPE parameter on the MC_ALLOCATE verb) is FULL_DUPLEX.
- The MC_DEALLOCATE TYPE specified is either FLUSH or SYNC_LEVEL.
- The MC_DEALLOCATE verb is issued from send-receive state.

State Changes:

Half-duplex conversations only:

Deallocate-Pending state is entered when TYPE(CONFIRM) is specified and the reply to the CONFIRM from the remote program has not been received. This is a transient state. When the verb completes, the conversation enters **Reset** or **Receive** state.

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

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

Full-duplex conversations:

Receive-only state is entered when a legitimate flush-level MC_DEALLOCATE is specified and the current conversation state is send-receive.

Reset state is entered when either a legitimate flush-level MC_DEALLOCATE is specified and the current conversation state is send-only, or when a TYPE(ABEND) is specified.

Notes:

1. 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.
2. The EXPEDITED_DATA_RECEIVED indicator is not set when the conversation is in reset state.

Half-duplex conversations only:

1. When defer-deallocate state is entered, the LU buffers the deallocation information to be sent to the remote LU until the local program issues a SYNCPT verb that causes the LU to flush its send buffer. If the program issues the BACKOUT verb instead, the data is flushed but the deallocate information is not sent.
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 receives an OK return code, indicating that all programs throughout the transaction responded to the sync-point request by issuing SYNCPT. The mapped conversation remains allocated when the program receives a BACKED_OUT return code to the SYNCPT verb indicating that a remote program responded to the sync-point request by issuing SEND_ERROR, or one or more programs responded 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(FLUSH) is not allowed with a mapped conversation with SYNC_LEVEL(SYNCPT).

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. TYPE(CONFIRM) is functionally equivalent to TYPE(SYNC_LEVEL) with a synchronization level of CONFIRM.

   When a DEALLOCATE TYPE(CONFIRM) verb is issued in a nonblocking manner, its execution may be suspended because of waiting for a reply to the CONFIRM request from the remote program. If so suspended, the verb receives an OPERATION_INCOMPLETE return code, and the conversation enters deallocate-pending state. In this state, the program cannot exchange any expedited data.

   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.

   TYPE(CONFIRM) is not allowed with a mapped conversation with SYNC_LEVEL(SYNCPT).

   When MC_DEALLOCATE TYPE(ABEND) is issued on a mapped conversation with synchronization level SYNCPT, all protected conversations used by that program enter the backout-required state.

5. 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_* return code.

   A TYPE(LOCAL) deallocation of the conversation is necessary in order for the local LU to recognize that the local program no longer needs a conversation after the remote program has deallocated the conversation. It is analagous to a called party hanging up a telephone after the calling party has already done so.

6. 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) or TYPE(SYNC_LEVEL) and the synchronization level is NONE.
- CONFIRM_DEALLOCATE what-received indication: The local program specified either TYPE(CONFIRM) or TYPE(SYNC_LEVEL) and the synchronization level is 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.

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

Full-duplex conversations only:

1. MC_DEALLOCATE TYPE(FLUSH) closes the local program's send queue. If a send queue verb is issued after the send queue has been closed, a PROGRAM_STATE_CHECK is returned. Both the local and remote program must close their send queues independently; therefore, two MC_DEALLOCATE TYPE(FLUSH) verbs are required to end the conversation. The send queues are closed independently so that data in transit in both directions is delivered before ending the conversation. Notification that the partner has closed its send queue is given to the receive queue in the form of a DEALLOCATE_NORMAL return code.

   When a MC_DEALLOCATE TYPE(FLUSH) is issued from send-receive state, the conversation state changes to receive-only, and the program may continue to receive data until a DEALLOCATE_NORMAL return code is received.
2. When a transaction program receives a DEALLOCATE_NORMAL return code while in send-receive state, the program is placed in send-only state. The program may continue to send data until it is ready to end the conversation by issuing an MC_DEALLOCATE TYPE(FLUSH).
3. An ERROR_INDICATION return code is returned when a terminating error has been received from the remote program; the MC_DEALLOCATE TYPE(FLUSH) request is not performed. The terminating error may be a DEALLOCATE_ABEND, an ALLOCATION_ERROR, or a conversation failure. The nature of the terminating error is reflected in a subcode to the ERROR_INDICATION return code. If the nature of the terminating error is

not known, a subcode of UNKNOWN_ERROR_TYPE_PENDING is returned.

When an ERROR_INDICATION return code is returned while the conversation state is send-only, the conversation state will change to reset. If an ERROR_INDICATION is returned while the conversation state is send-receive, no state change occurs; however, the send queue is effectively closed and subsequent send queue verbs will be rejected with the ERROR_INDICATION return code.

When an ERROR_INDICATION is returned while the conversation state is send-receive, the local program may end the conversation by either issuing receive queue verbs until the return code for the terminating error is received, or by issuing a MC_DEALLOCATE with TYPE(ABEND).

4. When the subcode associated with the ERROR_INDICATION return code is ALLOCATION_ERROR, no additional information is returned regarding the nature of the ALLOCATION_ERROR.

5. MC_DEALLOCATE TYPE(ABEND) is an abrupt termination that will close both sides of the conversation. Between the time when the local program's send queue begins processing the MC_DEALLOCATE TYPE(ABEND) and that processing is completed, verbs outstanding on both the local and remote conversation queues may receive notification that a MC_DEALLOCATE TYPE(ABEND) verb is in progress. The return codes used to give MC_DEALLOCATE TYPE(ABEND) notification to the conversation queues are shown in Table 5-2.

   When the MC_DEALLOCATE TYPE(ABEND) verb completes, the conversation state changes to reset and any remaining outstanding verbs are posted with a PROGRAM_PARAMETER_CHECK return code.

Table 5-2. Notification That a MC_DEALLOCATE TYPE(ABEND) Is in Progress.

	Local Program	**Remote Program**
send queue	MC_DEALLOCATE TYPE(ABEND) is processed on this queue.	ERROR_INDICATION (DEALLOCATE_ABEND_ PENDING)
receive queue	Data available to receive is returned, followed by DEALLOCATE_ABEND	Data available to receive is returned, followed by DEALLOCATE_ABEND.
expedited-send queue	CONVERSATION_ENDED	CONVERSATION_ENDED
expedited-receive queue	Expedited data available to receive is returned, followed by CONVERSATION_ENDED.	Expedited data available to receive is returned, followed by CONVERSATION_ENDED.

6. An indication of EXPEDITED_DATA_RECEIVED may be returned on an MC_DEALLOCATE TYPE(FLUSH) that was issued from send-receive state.

7. MC_DEALLOCATE with TYPE(CONFIRM) or TYPE(LOCAL) is not supported.

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_SEND_DATA, MC_PREPARE_TO_RECEIVE (see Note 2 below), 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	*Option No.* [*101*]	*Half-duplex* [√]	*Full-duplex* [√]
Supplied Parameters:			
RESOURCE (*variable*)			
WAIT_OBJECT *(BLOCKING)* *(VALUE (variable))*		*113*	
Returned Parameters:			
RETURN_CODE (*variable*)			

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID of the mapped conversation over which data is to be flushed.

WAIT_OBJECT specifies whether the verb should be processed in a blocking or nonblocking manner. The default value is a blocking verb operation. If a nonblocking verb operation is to be requested, then WAIT_OBJECT(VALUE(variable)) must be specified.

- **BLOCKING** specifies this is a blocking verb operation.
- **VALUE(*variable*)** specifies a nonblocking verb operation of one of two types:
  - a ***null*** variable specifies to complete the verb operation but without notification.
  - a nonnull ***variable*** specifies to complete the verb operation but with notification. The variable represents a wait object. Its use here associates it with completion of this verb. It can be listed as an entry in the WAIT_OBJECT_LIST in a subsequent WAIT_FOR_COMPLETION verb for notification (posting) purposes. Origination of this variable (by the TP or the operating system) is implementation dependent.

Returned Parameters:

RETURN_CODE specifies the variable in which the result of the verb execution is indicated to the local program.

- OK
- OPERATION_INCOMPLETE
- OPERATION_NOT_ACCEPTED
- PROGRAM_PARAMETER_CHECK, for the following reason:
  - RESOURCE specifies an unassigned resource ID.

Return codes for half-duplex conversations only:

- PROGRAM_STATE_CHECK, for the following reason:
  - The mapped conversation is not in send or defer-receive state.

Return codes for full-duplex conversations only:

- ERROR_INDICATION, with one of the following subcodes (Refer to Appendix F, "Conversation Return Codes" on page F-1 for a complete list of subcodes with their explanations):
  - ALLOCATION_ERROR_PENDING
  - DEALLOCATE_ABEND_PENDING
  - RESOURCE_FAILURE_NO_RETRY_PENDING
  - RESOURCE_FAILURE_RETRY_PENDING
  - UNKNOWN_ERROR_TYPE_PENDING
- PROGRAM_STATE_CHECK, for the following reason:
  - The full-duplex conversation is not in send-receive or send-only state.

State Changes:

Half-duplex conversations only:

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

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

Full-duplex conversations:

Reset state is entered when an ERROR_INDICATION return code is returned while in send-only 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. 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.
3. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

Half-duplex conversations only:

1. This verb cannot legally be issued in defer-deallocate state, a state entered when MC_DEALLOCATE TYPE(SYNC_LEVEL) is issued when the synchronization level for the mapped conversation is SYNCPT.
2. 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.

Full-duplex conversations only:

1. A full-duplex transaction program might choose to issue an MC_FLUSH after issuing an MC_SEND_ERROR in order to cause the FMH-7 to be sent to the partner program immediately. This is especially true if the program intends to begin receiving data after issuing the MC_SEND_ERROR.
2. An ERROR_INDICATION return code is returned when a terminating error has been received from the remote program; the MC_FLUSH request is not performed. The terminating error may be a DEALLOCATE_ABEND, an ALLOCATION_ERROR, or a conversation failure. The nature of the terminating error is reflected in a subcode to the ERROR_INDICATION return code. If the nature of the terminating error is not known, a subcode of UNKNOWN_ERROR_TYPE_PENDING is returned.

   When an ERROR_INDICATION return code is returned while the conversation state is send-only, the conversation state will change to reset. If an ERROR_INDICATION is returned while the conversation state is send-receive, no state change occurs; however, the send queue is effectively closed and subsequent send queue verbs will be rejected with the ERROR_INDICATION return code.

   When an ERROR_INDICATION is returned while the conversation state is send-receive, the local program may end the conversation by either issuing receive queue verbs until the return code for the terminating error is received, or by issuing a MC_DEALLOCATE with TYPE(ABEND).
3. When the subcode associated with the ERROR_INDICATION return code is ALLOCATION_ERROR, no additional information is returned regarding the nature of the ALLOCATION_ERROR.

MC_GET_ATTRIBUTES

Returns information pertaining to the specified mapped conversation.

Note: The GET_ATTRIBUTES verb is in the LU 6.2 Base; this verb is not.

MC_GET_ATTRIBUTES	*Option No.* [*102*]	*Half-duplex* [√]	*Full-duplex* [√]
Supplied Parameters:			
RESOURCE (*variable*)			
Returned Parameters:			
RETURN_CODE (*variable*)			
PARTNER_LU_NAME (*variable*)			
PARTNER_FULLY_QUALIFIED_LU_NAME (*variable*)			
MODE_NAME (*variable*)			
SYNC_LEVEL (*variable*)			
CONVERSATION_STATE (*variable*)		*108*	
CONVERSATION_CORRELATOR (*variable*)		*243 or 251*	
SESSION_ID (*variable*)		*251*	
CONVERSATION_GROUP_ID (*variable*)		*401*	

Supplied Parameters:

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

Returned Parameters:

RETURN_CODE specifies the variable in which the result of the verb execution is indicated to the local program.

- OK
- PROGRAM_PARAMETER_CHECK, for one of the following reasons:
  - RESOURCE specifies an unassigned resource ID.
  - CONVERSATION_STATE is specified and not supported.
  - CONVERSATION_CORRELATOR is specified and not supported.
  - CONVERSATION_GROUP_ID is specified and not supported.
  - SESSION_ID is specified and not supported.

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. See "FULLY_QUALIFIED_LU_NAME" in "DEFINE_MODE" on page 10-9.

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

CONVERSATION_STATE specifies the variable for returning the current conversation state of the specified mapped conversation. Possible values of the conversation state are shown in Appendix E, "Conversation State Matrices" on page E-1. All states shown in this appendix can be returned except the reset state, since a mapped conversation with a recognized resource ID is never in this state.

CONVERSATION_CORRELATOR specifies the variable for returning the conversation correlator. The conversation correlator is created and maintained by the LU. The conversation correlator is useful to identify a conversation instance and, unlike the RESOURCE parameter, can be known globally. The LU uses it during syncpoint resynchronization. If no conversation correlator is used on the mapped conversation, a null value is returned.

SESSION_ID specifies the variable for returning the identifier of the session being used by the conversation. This parameter value can be used in a DEACTIVATE_SESSION verb.

CONVERSATION_GROUP_ID specifies the variable for returning the conversation group identifier. The CONVERSATION_GROUP_ID is the identifier of a particular session between two particular LUs and provides a mechanism whereby one or more pairs of transaction programs (i.e., one TP at one LU and the other TP at the other LU) can serially share the same identified session as if all the TPs were in a *conversation group* of intercommunicating TPs taking turns with their conversations over the same session.

State Changes:

None

Notes:

1. The program may issue this verb in order to obtain the attributes of the mapped conversation, including the mapped conversation by which the program was started.
2. 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	*Option No.* [*103,104*]	*Half-duplex* [√]	*Full-duplex* [ ]
Supplied Parameters: **RESOURCE** (*variable*) **LENGTH** (*variable*)			
Returned Parameters: **RETURN_CODE** (*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. Posting occurs when the length value is reached.

Returned Parameters:

RETURN_CODE specifies the variable in which the result of the verb execution is indicated to the local program.

- OK
- PROGRAM_PARAMETER_CHECK, for one of the following reasons:
  - This verb is not supported.
  - RESOURCE specifies an unassigned resource ID.
  - This verb is not supported for full-duplex conversations.
- PROGRAM_STATE_CHECK, for the following reason:
  - The mapped conversation is not in receive state.

State Changes:

None

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_AND_WAIT
   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_FOR_SYNCPT

Causes the protected resources associated with the mapped conversation to be prepared to advance to the next synchronization point. Protected resources are those currently allocated to the transaction with a synchronization level of SYNCPT. As part of the MC_PREPARE_FOR_SYNCPT verb processing, the LU flushes its send buffer for the mapped conversation.

MC_PREPARE_FOR_SYNCPT	*Option No.* [*111*]	*Half-duplex* [√]	*Full-duplex* [ ]
Supplied Parameters:			
RESOURCE (*variable*)			
WAIT_OBJECT *(BLOCKING)* *(VALUE (variable))*		*113*	
Returned Parameters:			
RETURN_CODE (*variable*)			

Supplied Parameters:

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

WAIT_OBJECT specifies whether the verb should be processed in a blocking or nonblocking manner. The default value is a blocking verb operation. If a nonblocking verb operation is to be requested, then WAIT_OBJECT(VALUE(variable)) must be specified.

- **BLOCKING** specifies this is a blocking verb operation.
- VALUE(***variable***) specifies a nonblocking verb operation of one of two types:
  - a ***null*** variable specifies to complete the verb operation but without notification.
  - a nonnull ***variable*** specifies to complete the verb operation but with notification. The variable represents a wait object. Its use here associates it with completion of this verb. It can be listed as an entry in the WAIT_OBJECT_LIST in a subsequent WAIT_FOR_COMPLETION verb for notification (posting) purposes. Origination of this variable (by the TP or the operating system) is implementation dependent.

Returned Parameters:

RETURN_CODE specifies the variable in which the return code is returned to the transaction program. The return code indicates whether the prepare operation failed, or caused the mapped conversation partner to prepare to commit, or caused the mapped conversation partner to backout.

- OK: Protected resources at the mapped conversation partner are prepared to commit. All protected resources subordinate to the mapped conversation partner are also prepared to commit.
- BACKED_OUT: Protected resources at the mapped conversation partner have been backed out. The BACKED_OUT return code has one of the following subcodes, which indicates whether the backout occurred consistently in the rest of the distributed transaction subordinate to the mapped conversation partner.
  - ALL_AGREED: All protected resources in the distributed transaction have backed out.
  - LUW_OUTCOME_PENDING: As a result of failures and at least one program in the distributed transaction specifying SET_SYNCPT_OPTIONS WAIT_FOR_OUTCOME(NO), it is not known whether all protected resources have successfully backed out.
  - LUW_OUTCOME_MIXED: At least one protected resource has been advanced to the next synchronization point as a result of operator intervention following a failure. The distributed transaction is damaged.
  - OPERATION_INCOMPLETE
  - OPERATION_NOT_ACCEPTED
- PROGRAM_PARAMETER_CHECK, for one of the following reasons:
  - RESOURCE specifies an unassigned resource ID.
  - The synchronization level of the conversation is not SYNCPT.
  - This verb is not supported for full-duplex conversations.
- PROGRAM_STATE_CHECK, for the following reason:
  - The mapped conversation is not in send, defer-receive, or defer-deallocate state.
- ALLOCATION_ERROR
- DEALLOCATE_ABEND
- USER_CONTROL_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

State Changes:

From send state the conversation enters the sync-point-send state.

From defer-receive state the conversation enters the sync-point state.

From defer-deallocate state the conversation enters the sync-point-deallocate state.

***State Changes** (when RETURN_CODE indicates BACKED_OUT):*

Backout-required state is entered.

Notes:

1. The program may issue MC_PREPARE_FOR_SYNCPT on a protected mapped conversation that is in send, defer-receive, or defer-deallocate state. The MC_PREPARE_FOR_SYNCPT verb causes the LU to send a sync-point request on the mapped conversation. The remote program receives the sync-point request by means a TAKE_SYNCPT value of the WHAT_RECEIVED parameter of the MC_RECEIVE_AND_WAIT or MC_RECEIVE_IMMEDIATE verb. When the remote program receives this indicator, it may issue one of the following, with the specified effect:
   - SYNCPT: causes the sync-point request to be propagated to its other protected resources. When this completes, a sync-point message is sent to the local LU, causing the local program to get the OK return code.
   - BACKOUT: causes the backout operation to be propagated to its subordinate protected resources, and a back out message to be returned to the LU of the local program. The local program that issued the MC_PREPARE_FOR_SYNCPT verb gets a BACKED_OUT return code. All other protected conversations at the local program enter the backout-required state.
   - DEALLOCATE TYPE(ABEND): causes all protected conversations at the local program to enter the backout-required state.
   - SEND_ERROR: The local program gets a PROG_ERROR_NO_TRUNC return code. The resulting conversation state for the local program is receive state.
2. The remote program may have issued an MC_SEND_ERROR verb before receiving the TAKE_SYNCPT_* indication. If this occurs, control is returned to the local program with PROG_ERROR_PURGING return code and the conversation is in receive state.
3. If the local program receives an error return code that puts it in receive state, it can then attempt to fix the error and retry the sync point operation later. This is a major difference with the SYNCPT verb: if an error indication to data sent earlier is received during execution of a SYNCPT verb, the transaction is backed out without giving the program a chance to fix it.
4. RETURN_CODE indicates whether the mapped conversation partner has successfully prepared to commit. The subcode to RETURN_CODE indicates the action taken by the programs in the distributed transaction that are subordinate to the mapped conversation partner, i.e., the transaction programs to which it propagates the sync-point operation.
5. References in this verb description to a program being in a particular state are only in terms of each resource.

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	*Option No.* [*105*]	*Half-duplex* [√]	*Full-duplex* [ ]
Supplied Parameters:			
RESOURCE (*variable*)			
TYPE (*SYNC_LEVEL*)			
(FLUSH)			
(CONFIRM)			
LOCKS			
(SHORT)			
(LONG)		*244*	
WAIT_OBJECT			
(BLOCKING)			
(VALUE (variable))		*113*	
Returned Parameters:			
RETURN_CODE (*variable*)			

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID of the mapped conversation being used.

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

- **SYNC_LEVEL** 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. See Appendix E, "Conversation State Matrices" to see which state change results from which return code.

- If SYNC_LEVEL(SYNCPT), enter defer-receive 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. See Appendix E, "Conversation State Matrices" to see which state change results from which 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. See Appendix E, "Conversation State Matrices" to see which state change results from which 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-receive 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 or SYNCPT 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.

WAIT_OBJECT specifies whether the verb should be processed in a blocking or nonblocking manner. The default value is a blocking verb operation. If a nonblocking verb operation is to be requested, then WAIT_OBJECT(VALUE(variable)) must be specified.

- **BLOCKING** specifies this is a blocking verb operation.
- VALUE(***variable***) specifies a nonblocking verb operation of one of two types:
  - a ***null*** variable specifies to complete the verb operation but without notification.
  - a nonnull ***variable*** specifies to complete the verb operation but with notification. The variable represents a wait object. Its use here associates it with completion of this verb. It can be listed as an entry in the WAIT_OBJECT_LIST in a subsequent WAIT_FOR_COMPLETION verb for notification (posting) purposes. Origination of this variable (by the TP or the operating system) is implementation dependent.

Returned Parameters:

RETURN_CODE specifies the variable in which the result of the verb execution is indicated to the local program. The TYPE parameter and synchronization level of the mapped conversation affect the return codes that can be returned to the program.

The following return codes can be returned for all values of the TYPE parameter with any synchronization level:

- OK
- OPERATION_INCOMPLETE
- OPERATION_NOT_ACCEPTED
- PROGRAM_PARAMETER_CHECK, for one of the following reasons:
  - RESOURCE specifies an unassigned resource ID.
  - The TYPE parameter is inconsistent with the synchronization level of the mapped conversation, e.g., TYPE(CONFIRM) is specified and the conversation is allocated with SYNC_LEVEL(NONE).
  - This verb is not supported for full-duplex conversations.
- PROGRAM_STATE_CHECK, for the following reason:
  - The mapped conversation is not in send state.

Additional return codes are possible if TYPE(CONFIRM) is specified or if TYPE(SYNC_LEVEL) is specified when the synchronization level allocated to the mapped conversation is CONFIRM:

- ALLOCATION_ERROR
- BACKED_OUT (can only occur when the synchronization level is SYNCPT)
- DEALLOCATE_ABEND
- USER_CONTROL_DATA_NOT_SUPPORTED
- MAPPING_NOT_SUPPORTED
- MAP_NOT_FOUND
- MAP_EXECUTION_FAILURE

- PROG_ERROR_PURGING
- RESOURCE_FAILURE_NO_RETRY
- RESOURCE_FAILURE_RETRY

State Changes:

Defer receive 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.

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, MC_FLUSH, or MC_CONFIRM, 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. The use of LOCKS(LONG) can result in a reduction of line flows at the cost of longer completion of confirm or sync-point processing.
8. 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.

For half-duplex conversations:

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.

For full-duplex conversations:

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

MC_RECEIVE_AND_WAIT	*Option No.* [*Base*]	*Half-duplex* [√]	*Full-duplex* [√]
Supplied Parameters:			
RESOURCE (*variable*)			
WAIT_OBJECT *(BLOCKING)* *(VALUE (variable))*		*113*	
Supplied-and-Returned Parameters:			
LENGTH (*variable*)			
Returned Parameters:			
RETURN_CODE (*variable*) **REQUEST_TO_SEND_RECEIVED** (*variable*) **EXPEDITED_DATA_RECEIVED** (*variable*) **DATA** (*variable*) **WHAT_RECEIVED** (*variable*)		*112*	
MAP_NAME (*variable*)		*246*	

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID of the mapped conversation.

WAIT_OBJECT specifies whether the verb should be processed in a blocking or nonblocking manner. The default value is a blocking verb operation. If a nonblocking verb operation is to be requested, then WAIT_OBJECT(VALUE(variable)) must be specified.

- **BLOCKING** specifies this is a blocking verb operation.
- **VALUE(*variable*)** specifies a nonblocking verb operation of one of two types:
  - a ***null*** variable specifies to complete the verb operation but without notification.
  - a nonnull ***variable*** specifies to complete the verb operation but with notification. The variable represents a wait object. Its use here associates it with completion of this verb. It can be listed as an entry in the WAIT_OBJECT_LIST in a subsequent WAIT_FOR_COMPLETION

verb for notification (posting) purposes. Origination of this variable (by the TP or the operating system) is implementation dependent.

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 the result of the verb execution is indicated to the local program. The return codes that can be returned depend on the state of the mapped conversation at the time this verb is issued. The verb may be issued in send state or receive state. For full-duplex conversations, the verb may be issued from either send-receive state or receive-only state. State changes for full-duplex conversations occur only if a DEALLOCATE_* or ALLOCATION_ERROR return code is received.

- OK
- ALLOCATION_ERROR
- DEALLOCATE_ABEND
- DEALLOCATE_NORMAL
- MAPPING_NOT_SUPPORTED
- MAP_NOT_FOUND
- MAP_EXECUTION_FAILURE
- OPERATION_INCOMPLETE
- OPERATION_NOT_ACCEPTED
- PROG_ERROR_NO_TRUNC
- PROGRAM_PARAMETER_CHECK
  - RESOURCE specifies an unassigned resource ID.
- RESOURCE_FAILURE_NO_RETRY
- RESOURCE_FAILURE_RETRY
- USER_CONTROL_DATA_NOT_SUPPORTED

Return codes for half-duplex conversations only:

- BACKED_OUT
- PROG_ERROR_PURGING
- PROGRAM_STATE_CHECK
  - The mapped conversation is not in send or receive state.

Return codes for full-duplex conversations only:

- PROGRAM_STATE_CHECK
  The full-duplex conversation is not in send-receive or receive-only state.

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received.

- 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.

This parameter is always set to NO for full-duplex conversations.

EXPEDITED_DATA_RECEIVED specifies the variable in which is returned an indication of whether SEND_EXPEDITED_DATA has been received.

- YES indicates a SEND_EXPEDITED_DATA notification has been received from the remote transaction program.
- NO indicates a SEND_EXPEDITED_DATA notification has not been received from the remote transaction program.

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).
- USER_CONTROL_DATA_COMPLETE indicates the program received a complete data record or the last remaining portion of the record, and the data record contains user control data.
- USER_CONTROL_DATA_TRUNCATED indicates the program received less than a complete data record containing user control data, and the LU discarded the remainder of the data record.
- USER_CONTROL_DATA_INCOMPLETE indicates the program received less than a complete data record containing user control data, 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).

WHAT_RECEIVED for half-duplex conversations only:

- 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 either SYNCPT or MC_PREPARE_FOR_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 either SYNCPT or MC_PREPARE_FOR_SYNCPT. Issuing the SYNCPT verb also causes an affirmative reply to be returned to the remote program if the sync-point function is successful. 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 either SYNCPT or MC_PREPARE_FOR_SYNCPT. Issuing the SYNCPT verb also causes an affirmative reply to be returned to the remote program if the sync-point function is successful. 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:

Half-duplex conversations only:

Receive state is entered when the verb is issued in send state and WHAT_RECEIVED indicates DATA_COMPLETE, DATA_INCOMPLETE, USER_CONTROL_DATA_COMPLETE, or USER_CONTROL_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, USER_CONTROL_DATA_COMPLETE, or USER_CONTROL_DATA_INCOMPLETE.

Full-duplex Conversations only:

Reset state is entered when an ALLOCATION_ERROR return code is received, when a DEALLOCATE_NORMAL return code is received while in receive-only state, or a DEALLOCATE_ABEND return code is received.

Send-only state is entered when a DEALLOCATE_NORMAL return code is received while in send-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_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 USER_CONTROL_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, USER_CONTROL_DATA_TRUNCATED, or USER_CONTROL_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.

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_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.
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.
6. It is the responsibility of both sending and receiving installations to maintain the map-name definitions referenced by their application transaction programs.
7. The USER_CONTROL_DATA parameter of the MC_SEND_DATA verb allows the remote transaction program to indicate that it is passing user control data, such as FM headers. The presence of user control data is significant only to the transaction programs; the sending and receiving LUs do not perform any processing other than indicating that user control data is present.
8. No state change has occurred because EXPEDITED_DATA_RECEIVED indicates YES.

   When expedited data is received by the LU, it is indicated on the pertinent verbs if the return parameter EXPEDITED_DATA_RECEIVED exists on the verbs. It will continue to be indicated until such time that the data is received by the TP issuing the MC_RECEIVE_EXPEDITED_DATA verb.
9. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

Half-duplex conversations only:

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. 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.
3. 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" on page A-53).

4. 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.

Full-duplex conversations only:

1. The REQUEST_TO_SEND notification is always set to NO.
2. If the conversation allocation request has been buffered and has not yet been sent to the partner, the MC_RECEIVE_AND_WAIT processing will flush the send buffer and the allocation request will be sent to the partner.

MC_RECEIVE_EXPEDITED_DATA

Receives data sent by the remote transaction program in an expedited manner, via the MC_SEND_EXPEDITED_DATA verb. The program can issue this verb in any conversation state except for reset state.

MC_RECEIVE_EXPEDITED_DATA	*Option No.* [*112*]	*Half-duplex* [√]	*Full-duplex* [√]
Supplied Parameters:			
RESOURCE (*variable*)			
RETURN_CONTROL			
(WHEN_EXPEDITED_DATA_RECEIVED)			
(IMMEDIATE)			
WAIT_OBJECT			
(BLOCKING)			
(VALUE (variable))		*113*	
Supplied-and-Returned Parameters:			
LENGTH (*variable*)			
Returned Parameters:			
RETURN_CODE (*variable*)			
REQUEST_TO_SEND_RECEIVED (*variable*)			
EXPEDITED_DATA_RECEIVED (*variable*)		*112*	
DATA (*variable*)			

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID.

RETURN_CONTROL specifies when the local LU is to return control to the local program, in relation to the receiving of expedited data for the mapped conversation.

- **WHEN_EXPEDITED_DATA_RECEIVED** specifies to receive the expedited data for the mapped conversation before returning control to the program.
- **IMMEDIATE** specifies to receive the expedited data for the mapped conversation if expedited data is immediately available and to return control to the program with a return code indicating whether expedited data has been received.

WAIT_OBJECT specifies whether the verb should be processed in a blocking or nonblocking manner. The default value is a blocking verb operation. If a nonblocking verb operation is to be requested, then WAIT_OBJECT(VALUE(variable)) must be specified.

- **BLOCKING** specifies this is a blocking verb operation.
- **VALUE(*variable*)** specifies a nonblocking verb operation of one of two types:
  - a ***null*** variable specifies to complete the verb operation but without notification.
  - a nonnull ***variable*** specifies to complete the verb operation but with notification. The variable represents a wait object. Its use here associates it with completion of this verb. It can be listed as an entry in the WAIT_OBJECT_LIST in a subsequent WAIT_FOR_COMPLETION verb for notification (posting) purposes. Origination of this variable (by the TP or the operating system) is implementation dependent.

Supplied-and-Returned Parameters:

LENGTH specifies the variable containing a length value that is the buffer size the program has provided to receive the expedited data. If insufficient buffer size (including LENGTH = 0) is specified on the LENGTH parameter while the program is receiving the expedited data, a return code of INSUFFICIENT_BUFFER_SIZE_PROVIDED is returned to the program. In this case, the LENGTH value returned indicates the actual amount of data that is available and nothing is placed in the DATA parameter. If the program specifies a buffer size that is greater than or equal to the actual amount of data that is received, LENGTH is also set to the amount actually received. The valid expedited data size may vary from 0-86 bytes.

The program may issue the MC_RECEIVE_EXPEDITED_DATA verb to test if there is expedited data to be received rather than receive the data to its buffer. This can be done by specifying LENGTH = 0 and RETURN_CONTROL = IMMEDIATE in the MC_RECEIVE_EXPEDITED_DATA verb. If expedited data is available, the return code INSUFFICIENT_BUFFER_SIZE_PROVIDED is returned to the program. The program can then decide what to do next. If expedited data is not available, a return code UNSUCCESSFUL is returned to the program and the LENGTH field is not significant.

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 verb may be issued in any state except in the reset state.

- OK
- CONVERSATION_ENDED
- EXPEDITED_DATA_NOT_SUPPORTED_BY_LU
- INSUFFICIENT_BUFFER_SIZE_PROVIDED, for the following reason:
  - The expedited data the program received exceeds the receive buffer that the program was provided. The expedited data record cannot be partially received.

- OPERATION_INCOMPLETE
- OPERATION_NOT_ACCEPTED
- PROGRAM_PARAMETER_CHECK, for one of the following reasons:
  - RESOURCE specifies an unassigned resource ID
  - Expedited data is not supported
- PROGRAM_STATE_CHECK, for the following reason:
  - The mapped conversation is in deallocate-pending state.

An additional return code is possible depending on the RETURN_CONTROL value:

- RETURN_CONTROL(IMMEDIATE)
  - UNSUCCESSFUL, for the following reason:
    - Expedited data not immediately available

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received.

- YES indicates a REQUEST_TO_SEND notification has been received from the remote transaction program.
- NO indicates a REQUEST_TO_SEND notification has not been received from the remote transaction program.

This parameter is always set to NO for full-duplex conversations.

EXPEDITED_DATA_RECEIVED specifies the variable in which is returned an indication of whether SEND_EXPEDITED_DATA has been received.

- YES indicates a SEND_EXPEDITED_DATA notification has been received from the remote transaction program.
- NO indicates a SEND_EXPEDITED_DATA notification has not been received from the remote transaction program.

DATA specifies the variable in which the program is to receive the data. When the program provides insufficient buffer size, as indicated by the LENGTH parameter, nothing is placed in this variable.

State Changes:

None

Notes:

1. For RETURN_CONTROL(WHEN_EXPEDITED_DATA_RECEIVED), if a normal conversation deallocation occurs when a MC_RECEIVE_EXPEDITED_DATA verb is outstanding, and no expedited data is available to receive, a return code of CONVERSATION_ENDED will be returned. CONVERSATION_ENDED is also returned when a DEALLOCATE(ABEND), allocation error, or conversation failure occurs. This return code indicates that a MC_RECEIVE_EXPEDITED_DATA verb was outstanding when the conversation ended, the verb was not completed successfully, and the expedited-receive queue is effectively closed.

No state change occurs because CONVERSATION_ENDED is returned. Subsequent MC_RECEIVE_EXPEDITED_DATA verbs will not be performed, but will be rejected with the CONVERSATION_ENDED return code.

2. If a MC_SEND_EXPEDITED_DATA verb is issued from the local program, a MC_RECEIVE_EXPEDITED_DATA verb needs to be issued by the remote program to receive the expedited data.
3. No state change has occurred because EXPEDITED_DATA_RECEIVED indicates YES.

   When expedited data is received by the LU, it is indicated on the pertinent verbs if the return parameter EXPEDITED_DATA_RECEIVED exists on the verbs. It will continue to be indicated until such time that the data is received by the TP issuing the RECEIVE_EXPEDITED_DATA verb.

Full-duplex conversations only:

1. The REQUEST_TO_SEND notification is always set to NO.

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.

When the send-receive queue for half-duplex conversations or the receive queue for full-duplex conversations is not empty, the verb can be blocked just like the MC_RECEIVE_AND_WAIT verb.

MC_RECEIVE_IMMEDIATE	*Option No.* [*106*]	*Half-duplex* [√]	*Full-duplex* [√]
Supplied Parameters:			
RESOURCE (*variable*)			
WAIT_OBJECT			
(BLOCKING)			
(VALUE (variable))		*113*	
Supplied-and-Returned Parameters:			
LENGTH (*variable*)			
Returned Parameters:			
RETURN_CODE (*variable*)			
REQUEST_TO_SEND_RECEIVED (*variable*)			
EXPEDITED_DATA_RECEIVED (*variable*)		*112*	
DATA (*variable*)			
WHAT_RECEIVED (*variable*)			
MAP_NAME (*variable*)		*246*	

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID of the conversation.

WAIT_OBJECT specifies whether the verb should be processed in a blocking or nonblocking manner. The default value is a blocking verb operation. If a non-blocking verb operation is to be requested, then WAIT_OBJECT(VALUE(variable)) must be specified.

- **BLOCKING** specifies this is a blocking verb operation.
- VALUE(*variable*) specifies a nonblocking verb operation of one of two types:
  - a ***null*** variable specifies to complete the verb operation but without notification.
  - a nonnull ***variable*** specifies to complete the verb operation but with notification. The variable represents a wait object. Its use here associates it with completion of this verb. It can be listed as an entry in the WAIT_OBJECT_LIST in a subsequent WAIT_FOR_COMPLETION verb for notification (posting) purposes. Origination of this variable (by the TP or the operating system) is implementation dependent.

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 the result of the verb execution is indicated to the local program.

- OK
- ALLOCATION_ERROR
- DEALLOCATE_ABEND
- DEALLOCATE_NORMAL
- MAPPING_NOT_SUPPORTED
- MAP_NOT_FOUND
- MAP_EXECUTION_FAILURE
- OPERATION_INCOMPLETE
- OPERATION_NOT_ACCEPTED
- PROG_ERROR_NO_TRUNC
- PROGRAM_PARAMETER_CHECK, for the following reason:
  - RESOURCE specifies an unassigned resource ID.
- RESOURCE_FAILURE_NO_RETRY
- RESOURCE_FAILURE_RETRY
- UNSUCCESSFUL - There is nothing to receive.
- USER_CONTROL_DATA_NOT_SUPPORTED

Return codes for half-duplex conversations only:

- BACKED_OUT
- PROG_ERROR_PURGING
- PROGRAM_STATE_CHECK, for the following reason:
  - The mapped conversation is not in receive state.

Return codes for full-duplex conversations only:

- PROGRAM_STATE_CHECK, for the following reason:

- The full-duplex conversation is not in send-receive or receive-only state.

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received.

- 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.

Note: If RETURN_CODE is set to PROGRAM_PARAMETER_CHECK or PROGRAM_STATE_CHECK, the value contained in REQUEST_TO_SEND_RECEIVED is meaningless.

This parameter is always set to NO for full-duplex conversations.

EXPEDITED_DATA_RECEIVED specifies the variable in which is returned an indication of whether SEND_EXPEDITED_DATA has been received.

- YES indicates a SEND_EXPEDITED_DATA notification has been received from the remote transaction program.
- NO indicates a SEND_EXPEDITED_DATA notification has not been received from the remote transaction program.

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.

WHAT_RECEIVED for both half-duplex and full-duplex conversations:

- 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).
- USER_CONTROL_DATA_COMPLETE indicates the program received a complete data record or the last remaining portion of the record, and the data record contains user control data.
- USER_CONTROL_DATA_TRUNCATED indicates the program received less than a complete data record containing user control data, and the LU discarded the remainder of the data record.
- USER_CONTROL_DATA_INCOMPLETE indicates the program received less than a complete data record containing user control data, 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).

WHAT_RECEIVED for half-duplex conversations only:

- 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 either SYNCPT or MC_PREPARE_FOR_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 issued MC_PREPARE_TO_RECEIVE with TYPE(SYNC_LEVEL), the synchronization level is SYNCPT, and the remote program subsequently issued either SYNCPT or MC_PREPARE_FOR_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 either SYNCPT or MC_PREPARE_FOR_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:

Half-duplex conversations only:

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, USER_CONTROL_DATA_COMPLETE, or USER_CONTROL_DATA_INCOMPLETE.

Full-duplex conversations only:

Reset state is entered when a DEALLOCATE_NORMAL indication is received while in receive-only state, when a DEALLOCATE_ABEND return code is received, or when an ALLOCATION_ERROR return code is received.

Send-only state is entered when a DEALLOCATE_NORMAL indication is received while in send-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 USER_CONTROL_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, USER_CONTROL_DATA_TRUNCATED, or USER_CONTROL_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. It is the responsibility of both sending and receiving installations to maintain the map-name definitions referenced by their application transaction programs.
7. The USER_CONTROL_DATA parameter of the MC_SEND_DATA verb allows the remote transaction program to indicate that it is passing user control data, such as FM headers. The presence of user control data is significant only to the transaction programs; the sending and receiving LUs do not perform any processing other than indicating that user control data is present.
8. No state change has occurred because EXPEDITED_DATA_RECEIVED indicates YES.

   When expedited data is received by the LU, it is indicated on the pertinent verbs if the return parameter EXPEDITED_DATA_RECEIVED exists on the verbs. It will continue to be indicated until such time that the data is received by the TP issuing the MC_RECEIVE_EXPEDITED_DATA verb.
9. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

Half-duplex conversations only:

1. 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.
2. 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" on page A-53).

3. 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.

Full-duplex conversations only:

1. The REQUEST_TO_SEND notification is always set to NO.

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	*Option No.* [*Base*]	*Half-duplex* [√]	*Full-duplex* [ ]
Supplied Parameters: **RESOURCE** (*variable*) **WAIT_OBJECT** *(BLOCKING)* *(VALUE (variable))*		*113*	
Returned Parameters: **RETURN_CODE** (*variable*)			

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID for the conversation.

WAIT_OBJECT specifies whether the verb should be processed in a blocking or nonblocking manner. The default value is a blocking verb operation. If a nonblocking verb operation is to be requested, then WAIT_OBJECT(VALUE(variable)) must be specified.

- **BLOCKING** specifies this is a blocking verb operation.
- VALUE(***variable***) specifies a nonblocking verb operation of one of two types:
  - a ***null*** variable specifies to complete the verb operation but without notification.
  - a nonnull ***variable*** specifies to complete the verb operation but with notification. The variable represents a wait object. Its use here associates it with completion of this verb. It can be listed as an entry in the WAIT_OBJECT_LIST in a subsequent WAIT_FOR_COMPLETION verb for notification (posting) purposes. Origination of this variable (by the TP or the operating system) is implementation dependent.

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

- CONVERSATION_ENDED
- OPERATION_INCOMPLETE
- OPERATION_NOT_ACCEPTED
- PROGRAM_PARAMETER_CHECK, for one of the following reasons:
  - RESOURCE specifies an unassigned resource ID.
  - This verb is not supported for full-duplex conversations.
- PROGRAM_STATE_CHECK, for one of the following reasons:
  - The mapped conversation is not in receive, confirm, send, or sync-point state.
  - The mapped conversation is in deallocate-pending state.

State Changes:

None

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. The MC_REQUEST_TO_SEND verb, as described in this book, may be issued only when the conversation is in send, 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.

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

6. If a normal conversation deallocation occurs when a MC_REQUEST_TO_SEND verb is outstanding, a return code of CONVERSATION_ENDED is returned. CONVERSATION_ENDED is also returned when a DEALLOCATE(ABEND), allocation error, or conversation failure occurs. This return code indicates that a MC_REQUEST_TO_SEND verb was outstanding when the conversation ended, the verb was not completed successfully, and the expedited-send queue is effectively closed.

   No state change occurs because CONVERSATION_ENDED is returned. Subsequent MC_REQUEST_TO_SEND verbs will not be performed, but will be rejected with the CONVERSATION_ENDED return code.

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 user control data.

MC_SEND_DATA	*Option No.* [*Base*]	*Half-duplex* [√]	*Full-duplex* [√]
Supplied Parameters:			
RESOURCE (*variable*)			
DATA (*variable*)			
LENGTH (*variable*)			
MAP_NAME			
(NO)			
(YES (variable))		*246*	
USER_CONTROL_DATA			
(NO)			
(YES)		*247*	
ENCRYPT			
(NO)			
(YES)		*611, 617*	
WAIT_OBJECT			
(BLOCKING)			
(VALUE (variable))		*113*	
Returned Parameters:			
RETURN_CODE (*variable*)			
REQUEST_TO_SEND_RECEIVED (*variable*)			
EXPEDITED_DATA_RECEIVED (*variable*)		*112*	

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.

[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.

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.

USER_CONTROL_DATA specifies whether the data record contains user control data.

- **NO** specifies that user control data is not present in the data record.
- **YES** specifies that the data record contains user control data.

ENCRYPT specifies encryption is to be used when the mode allows selective encryption.

- **NO** specifies encryption is not required.
- **YES** specifies encryption is required.

If the encryption of the mode name is none, ENCRYPTION_NOT_SUPPORTED will be returned. If the encryption for the mode name is mandatory, the ENCRYPT parameter is ignored, as all data is encrypted. If the mode allows selective encryption, the action by the LU depends on the result of the negotiation with the remote LU at session activation:

- If selective encryption was agreed upon, the default value of the parameter is **NO**.
- If mandatory encryption was agreed upon, the parameter is ignored, as all data is encrypted.

WAIT_OBJECT specifies whether the verb should be processed in a blocking or nonblocking manner. The default value is a blocking verb operation. If a nonblocking verb operation is to be requested, then WAIT_OBJECT(VALUE(variable)) must be specified.

- **BLOCKING** specifies this is a blocking verb operation.
- VALUE(***variable***) specifies a nonblocking verb operation of one of two types:
  - a ***null*** variable specifies to complete the verb operation but without notification.
  - a nonnull ***variable*** specifies to complete the verb operation but with notification. The variable represents a wait object. Its use here associates it with completion of this verb. It can be listed as an entry in the WAIT_OBJECT_LIST in a subsequent WAIT_FOR_COMPLETION verb for notification (posting) purposes. Origination of this variable (by the TP or the operating system) is implementation dependent.

Returned Parameters:

RETURN_CODE specifies the variable in which the result of the verb execution is indicated to the local program.

- OK
- ENCRYPTION_NOT_SUPPORTED
- MAPPING_NOT_SUPPORTED
- MAP_NOT_FOUND
- MAP_EXECUTION_FAILURE
- OPERATION_INCOMPLETE
- OPERATION_NOT_ACCEPTED
- PROGRAM_PARAMETER_CHECK, for the following reason:
  - RESOURCE specifies an unassigned resource ID.
- USER_CONTROL_DATA_NOT_SUPPORTED

Return codes for half-duplex conversations only:

- ALLOCATION_ERROR
- BACKED_OUT
- DEALLOCATE_ABEND
- PROG_ERROR_PURGING
- PROGRAM_STATE_CHECK, for the following reason:
  - The mapped conversation is not is send state.
- RESOURCE_FAILURE_NO_RETRY
- RESOURCE_FAILURE_RETRY

Return codes for full-duplex conversations only:

- ERROR_INDICATION, with one of the following subcodes (Refer to Appendix F, "Conversation Return Codes" on page F-1 for a complete list of subcodes with their explanations):
  - ALLOCATION_ERROR_PENDING
  - DEALLOCATE_ABEND_PENDING
  - RESOURCE_FAILURE_NO_RETRY_PENDING
  - RESOURCE_FAILURE_RETRY_PENDING
  - UNKNOWN_ERROR_TYPE_PENDING
- PROGRAM_STATE_CHECK, for the following reason:
  - The conversation is not in send-receive or send-only state.

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received.

- 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.

This parameter is always set to NO for full-duplex conversations.

EXPEDITED_DATA_RECEIVED specifies the variable in which is returned an indication of whether SEND_EXPEDITED_DATA has been received.

- YES indicates a SEND_EXPEDITED_DATA notification has been received from the remote transaction program.
- NO indicates a SEND_EXPEDITED_DATA notification has not been received from the remote transaction program.

State Changes:

Half-duplex conversations only:

None

Full-duplex conversations only:

Reset state is entered when an ERROR_INDICATION return code is returned while in send-only 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 map name 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. No state change has occurred because EXPEDITED_DATA_RECEIVED indicates YES.

   When expedited data is received by the LU, it is indicated on the pertinent verbs if the return parameter EXPEDITED_DATA_RECEIVED exists on the verbs. It will continue to be indicated until such time that the data is received by the TP issuing the MC_RECEIVE_EXPEDITED_DATA verb.
9. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

Half-duplex conversations only:

1. 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).

Full-duplex conversations only:

1. The REQUEST_TO_SEND notification is always set to NO.
2. An ERROR_INDICATION return code is returned when a terminating error has been received from the remote program; the MC_SEND_DATA request is not performed. The terminating error may be a DEALLOCATE_ABEND, an ALLOCATION_ERROR, or a conversation failure. The nature of the terminating error is reflected in a subcode to the ERROR_INDICATION return code. If the nature of the terminating error is not known, a subcode of UNKNOWN_ERROR_TYPE_PENDING is returned.

   When an ERROR_INDICATION return code is returned while the conversation state is send-only, the conversation state will change to reset. If an ERROR_INDICATION is returned while the conversation state is send-receive, no state change occurs; however, the send queue is effectively closed and subsequent MC_SEND_DATA verbs will be rejected with the ERROR_INDICATION return code.

   When an ERROR_INDICATION is returned while the conversation state is send-receive, the local program may end the conversation by either issuing receive queue verbs until the return code for the terminating error is received, or by issuing a MC_DEALLOCATE with TYPE(ABEND).
3. When the subcode associated with the ERROR_INDICATION return code is ALLOCATION_ERROR, no additional information is returned regarding the nature of the ALLOCATION_ERROR.

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.

For half-duplex conversations:

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.

For full-duplex conversations:

No state changes occur as a result of issuing this verb. MC_SEND_ERROR is a send queue verb that is used to report errors that occur in the send direction only. The error notification will be reported to the remote program's receive queue.

MC_SEND_ERROR	*Option No.* [*Base*]	*Half-duplex* [√]	*Full-duplex* [√]
Supplied Parameters:			
RESOURCE (*variable*)			
WAIT_OBJECT			
(BLOCKING)			
(VALUE (variable))		*113*	
Returned Parameters:			
RETURN_CODE (*variable*)			
REQUEST_TO_SEND_RECEIVED (*variable*)			
EXPEDITED_DATA_RECEIVED (*variable*)		*112*	

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID of the conversation.

WAIT_OBJECT specifies whether the verb should be processed in a blocking or nonblocking manner. The default value is a blocking verb operation. If a nonblocking verb operation is to be requested, then WAIT_OBJECT(VALUE(variable)) must be specified.

- **BLOCKING** specifies this is a blocking verb operation.
- **VALUE(*variable*)** specifies a nonblocking verb operation of one of two types:

- a *null* variable specifies to complete the verb operation but without notification.
- a nonnull *variable* specifies to complete the verb operation but with notification. The variable represents a wait object. Its use here associates it with completion of this verb. It can be listed as an entry in the WAIT_OBJECT_LIST in a subsequent WAIT_FOR_COMPLETION verb for notification (posting) purposes. Origination of this variable (by the TP or the operating system) is implementation dependent.

Returned Parameters:

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

- OK
- OPERATION_INCOMPLETE
- OPERATION_NOT_ACCEPTED
- PROGRAM_PARAMETER_CHECK
  - RESOURCE specifies an unassigned resource ID.

Return codes for half-duplex conversations only:

- If this verb is issued in send state, the following return codes can be returned:
  - ALLOCATION_ERROR
  - BACKED_OUT
  - DEALLOCATE_ABEND
  - 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:
  - 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:
  - RESOURCE_FAILURE_NO_RETRY
  - RESOURCE_FAILURE_RETRY
- If this verb is issued in any state other than send, receive, confirm or sync-point state, the following return codes can be returned:
  - PROGRAM_STATE_CHECK
    - The mapped conversation is not in send, receive, confirm, or sync-point state.

Return codes for full-duplex conversations only:

- ERROR_INDICATION, with one of the following subcodes (Refer to Appendix F, "Conversation Return Codes" on page F-1 for a complete list of subcodes with their explanations):

  - ALLOCATION_ERROR_PENDING
  - DEALLOCATE_ABEND_PENDING
  - RESOURCE_FAILURE_NO_RETRY_PENDING
  - RESOURCE_FAILURE_RETRY_PENDING
  - UNKNOWN_ERROR_TYPE_PENDING
- PROGRAM_STATE_CHECK, for the following reason:
  - The conversation is not in send-receive or send-only state.

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received.

- 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.

This parameter is always set to NO for full-duplex conversations.

EXPEDITED_DATA_RECEIVED specifies the variable in which is returned an indication of whether SEND_EXPEDITED_DATA has been received.

- YES indicates a SEND_EXPEDITED_DATA notification has been received from the remote transaction program.
- NO indicates a SEND_EXPEDITED_DATA notification has not been received from the remote transaction program.

State Changes:

Half-duplex conversations only:

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.

Full-duplex conversations only:

Reset state is entered when an ERROR_INDICATION return code is returned while in send-only 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" on page A-53), 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. 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.
4. No state change has occurred because EXPEDITED_DATA_RECEIVED indicates YES.

   When expedited data is received by the LU, it is indicated on the pertinent verbs if the return parameter EXPEDITED_DATA_RECEIVED exists on the verbs. It will continue to be indicated until such time that the data is received by the TP issuing the MC_RECEIVE_EXPEDITED_DATA verb.
5. References in this verb description to a program being in a particular state are only in terms of the specified mapped conversation.

Half-duplex conversations only:

1. 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.
2. 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
   - MAPPING_NOT_SUPPORTED
   - MAP_NOT_FOUND
   - MAP_EXECUTION_FAILURE
   - PROG_ERROR_NO_TRUNC
   - PROG_ERROR_PURGING
   - USER_CONTROL_DATA_NOT_SUPPORTED

The return code DEALLOCATE_NORMAL is reported instead of ALLOCATION_ERROR or DEALLOCATE_ABEND. If a DEALLOCATE_NORMAL return code to MC_SEND_ERROR is received on a mapped conversation allocated with SYNC_LEVEL(SYNCPT), the LU assumes that a DEALLOCATE_ABEND_* has occurred, even though the return code indication was purged, since MC_DEALLOCATE TYPE(FLUSH) or TYPE(CONFIRM) are not allowed on protected conversations. As a result, other protected conversations of that program will be put in the backout-required state.

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.

3. 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).
4. 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.

Full-duplex conversations only:

1. The REQUEST_TO_SEND notification is always set to NO.
2. MC_SEND_ERROR is a send queue verb that is used to report an error that occurred in the local program's send direction. The error is reported to the remote transaction program's receive queue as PROG_ERROR_NO_TRUNC. No data truncation occurs at the mapped conversation protocol boundary.

3. MC_SEND_ERROR is not supported in the receive direction; therefore, there is no equivalent to MC_SEND_ERROR issued from receive state for half-duplex conversations. Purging of incoming information does not take place.

4. An ERROR_INDICATION return code is returned when a terminating error has been received from the remote program, and the MC_SEND_ERROR verb is not performed. The terminating error may be an MC_DEALLOCATE_ABEND, an ALLOCATION_ERROR, or a conversation failure. The nature of the terminating error is reflected in a subcode to the ERROR_INDICATION return code. If the nature of the terminating error is not known, a subcode of UNKNOWN_ERROR_TYPE_PENDING is returned.

   When an ERROR_INDICATION return code is returned while the conversation state is send-only, the conversation state will change to reset. If an ERROR_INDICATION is returned while the conversation state is send-receive, no state change occurs; however, the send queue is effectively closed and subsequent MC_SEND_ERROR verbs will be rejected with the ERROR_INDICATION return code.

   When an ERROR_INDICATION is returned while the conversation state is send-receive, the local program may end the conversation by either issuing receive queue verbs until the return code for the terminating error is received, or by issuing a MC_DEALLOCATE with TYPE(ABEND).

5. When the subcode associated with the ERROR_INDICATION return code is ALLOCATION_ERROR, no additional information is returned regarding the nature of the ALLOCATION_ERROR.

MC_SEND_EXPEDITED_DATA

Sends data to the remote transaction program in an expedited manner. This means it may arrive at the remote transaction program before data sent earlier via a send queue verb, e.g., MC_SEND_DATA. (See "Nonblocking Support for LU 6.2 Conversation Verbs" on page 3-9 for a complete list of send queue verbs.)

At the remote transaction program, a MC_RECEIVE_EXPEDITED_DATA verb is used to receive this data.

MC_SEND_EXPEDITED_DATA	*Option No.* [*112*]	*Half-duplex* [√]	*Full-duplex* [√]
Supplied Parameters:			
DATA (*variable*) **LENGTH** (*variable*) **RESOURCE** (*variable*)			
WAIT_OBJECT *(BLOCKING)* *(VALUE (variable))*		*113*	
Returned Parameters:			
RETURN_CODE (*variable*) **REQUEST_TO_SEND_RECEIVED** (*variable*) **EXPEDITED_DATA_RECEIVED** (*variable*)			

Supplied Parameters:

DATA specifies the variable containing the data record to be sent. The data record consists entirely of data and is not in logical record format: no length field (LL) is present.

LENGTH specifies the variable containing the length (in bytes) of the data record to be sent. The valid length is 1-86. If the value specified for the LENGTH field is not in the range 1-86, an error return code PROGRAM_PARAMETER_CHECK is returned to the MC_SEND_EXPEDITED_DATA verb.

The amount of DATA to be sent is determined by the length value specified. If the amount of DATA to be sent is greater than the length specified, the extra data is ignored.

RESOURCE specifies the variable containing the resource ID.

WAIT_OBJECT specifies whether the verb should be processed in a blocking or nonblocking manner. The default value is a blocking verb operation. If a nonblocking verb operation is to be requested, then WAIT_OBJECT(VALUE(variable)) must be specified.

- **BLOCKING** specifies this is a blocking verb operation.
- **VALUE(*variable*)** specifies a nonblocking verb operation of one of two types:
  - a ***null*** variable specifies to complete the verb operation but without notification.
  - a nonnull ***variable*** specifies to complete the verb operation but with notification. The variable represents a wait object. Its use here associates it with completion of this verb. It can be listed as an entry in the WAIT_OBJECT_LIST in a subsequent WAIT_FOR_COMPLETION verb for notification (posting) purposes. Origination of this variable (by the TP or the operating system) is implementation dependent.

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 verb may be issued in any state except in reset state.

Return codes for both half-duplex and full-duplex conversations:

- OK
- CONVERSATION_ENDED
- EXPEDITED_DATA_NOT_SUPPORTED_BY_LU
- OPERATION_INCOMPLETE
- OPERATION_NOT_ACCEPTED
- PROGRAM_PARAMETER_CHECK, for one of the following reasons:
  - RESOURCE specifies an unassigned resource ID
  - LENGTH specifies an invalid length value has been used that is not in the range 1-86.
  - Expedited data is not supported
- PROGRAM_STATE_CHECK, for the following reason:
  - The mapped conversation is in deallocate-pending state.

REQUEST_TO_SEND_RECEIVED specifies the variable in which is returned an indication of whether REQUEST_TO_SEND has been received.

- YES indicates a REQUEST_TO_SEND notification has been received from the remote transaction program.
- NO indicates a REQUEST_TO_SEND notification has not been received from the remote transaction program.

This parameter is always set to NO for full-duplex conversations.

EXPEDITED_DATA_RECEIVED specifies the variable in which is returned an indication of whether SEND_EXPEDITED_DATA has been received.

- YES indicates a SEND_EXPEDITED_DATA notification has been received from the remote transaction program.

- NO indicates a SEND_EXPEDITED_DATA notification has not been received from the remote transaction program.

State Changes:

None

Notes:

1. When the remote LU receives the expedited data, it retains the data until the remote program issues a RECEIVE_EXPEDITED_DATA verb.
2. It is the implementation's choice to allow the remote LU to retain one or more than one expedited data at a time (per conversation). However, the not-yet-received expedited data must not be overwritten by the new one. The implementation should not send + RSP(EXPD) to the remote LU unless an expedited-flow buffer is available to receive the next expedited data (if any).
3. If a normal conversation deallocation occurs when a MC_SEND_EXPEDITED_DATA verb is outstanding, a return code of CONVERSATION_ENDED is returned. CONVERSATION_ENDED is also returned when a DEALLOCATE(ABEND), allocation error, or conversation failure occurs. This return code indicates that a MC_SEND_EXPEDITED_DATA verb was outstanding when the conversation ended, the verb was not completed successfully, and the expedited-send queue is effectively closed.

   No state change occurs because CONVERSATION_ENDED is returned. Subsequent MC_SEND_EXPEDITED_DATA verbs will not be performed, but will be rejected with the CONVERSATION_ENDED return code.
4. If a MC_SEND_EXPEDITED_DATA verb is issued from the local program, a MC_RECEIVE_EXPEDITED_DATA verb needs to be issued by the remote program to receive the expedited data.

Full-duplex conversations only:

1. The REQUEST_TO_SEND notification is always set to NO.

MC_TEST

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

MC_TEST	*Option No.* [*103,245*]	*Half-duplex* [√]	*Full-duplex* [ ]
Supplied Parameters:			
RESOURCE (*variable*)			
TEST (***POSTED***) (*REQUEST_TO_SEND_RECEIVED*)		*245*	
Returned Parameters:			
RETURN_CODE (*variable*)			

Supplied Parameters:

RESOURCE specifies the variable containing the resource ID of the conversation.

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 the result of the verb execution is indicated to the local 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
- USER_CONTROL_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
- PROGRAM_PARAMETER_CHECK, for one of the following reasons:
  - RESOURCE specifies an unassigned resource ID.
  - This verb is not supported for full-duplex conversations.
- PROGRAM_STATE_CHECK, for the following reason:
  - TEST(POSTED) is specified and the mapped conversation is not in receive state.

• If TEST(REQUEST_TO_SEND_RECEIVED) is specified, one of the following return codes is returned:

- OK
- UNSUCCESSFUL
- PROGRAM_PARAMETER_CHECK, for one of the following reasons:
  - RESOURCE specifies an unassigned resource ID.
  - This verb is not supported for full-duplex conversations.
- PROGRAM_STATE_CHECK, for the following reason:
  - TEST(REQUEST_TO_SEND_RECEIVED) is specified and the mapped conversation is not in send, defer-receive, defer-deallocate, or receive state.

State Changes:

None

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.

  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).

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

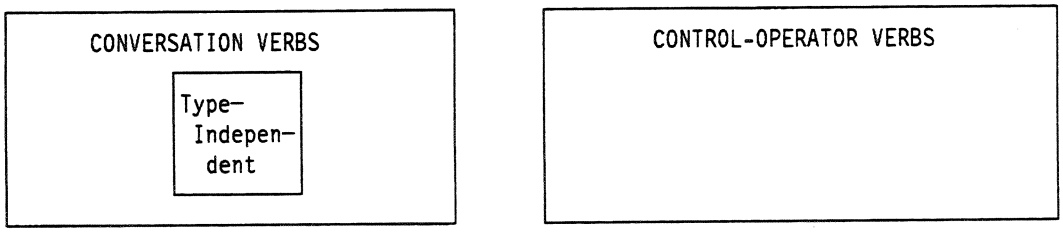
The remaining return codes indicate posting was active for the mapped conversation and it has been posted for the reason indicated by the specific return code. Posting is now reset.

3. 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.

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

Chapter 6. Type-Independent Conversation Verbs



This chapter 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, defer receive, or defer deallocate state.

BACKOUT	*Option No.* [*108*]
Supplied Parameters:	
WAIT_OBJECT *(BLOCKING)* *(VALUE (variable))*	*113*
Returned Parameters:	
RETURN_CODE (*variable*)	

Supplied Parameters:

WAIT_OBJECT specifies whether the verb should be processed in a blocking or nonblocking manner. The default value is a blocking verb operation. If a nonblocking verb operation is to be requested, then WAIT_OBJECT(VALUE(variable)) must be specified.

- **BLOCKING** specifies this is a blocking verb operation.
- VALUE(***variable***) specifies a nonblocking verb operation of one of two types:
  - a ***null*** variable specifies to complete the verb operation but without notification.
  - a nonnull ***variable*** specifies to complete the verb operation but with notification. The variable represents a wait object. Its use here associates it with completion of this verb. It can be listed as an entry in the WAIT_OBJECT_LIST in a subsequent WAIT_FOR_COMPLETION verb for notification (posting) purposes. Origination of this variable (by the TP or the operating system) is implementation dependent.

Returned Parameters:

RETURN_CODE specifies the variable in which the result of the verb execution is indicated to the local program.

- OK: All local resources have been successfully backed out. The OK return code has one of the following subcodes, which indicates the success of the backout operation in the rest of the distributed transaction.
  - ALL_AGREED: All protected resources in the distributed transaction have successfully backed out.
  - LUW_OUTCOME_PENDING: As a result of failures and at least one program in the distributed transaction specifying SET_SYNCPT_OPTIONS WAIT_FOR_OUTCOME(NO), it is not known whether all protected resources have backed out.
  - LUW_OUTCOME_MIXED: At least one protected resource has been advanced to the next synchronization point as a result of operator intervention. The distributed transaction is damaged.

    See Notes 6 and 7 for a discussion of the circumstances that could cause the last two subcodes to occur.
- OPERATION_INCOMPLETE
- OPERATION_NOT_ACCEPTED

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.

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.

5. The RETURN_CODE indicates whether the local backout succeeded. If it did, the OK subcode indicates the success of backing out the rest of the distributed transaction.
6. The LUW_OUTCOME_PENDING or LUW_OUTCOME_MIXED subcodes can only occur if the BACKOUT verb follows the (MC_)PREPARE_FOR_SYNCPT verb and an LU or conversation failure occurs somewhere in the distributed transaction. In that case, part of the synchronization operation was completed before the BACKOUT verb was issued, and the two partners affected by the failure are required to resynchronize with each other. The LUW_OUTCOME_PENDING subcode occurs if a first attempt at resynchronization fails and the program at the LU attempting resynchronization had issued the SET_SYNCPT_OPTIONS verb with the WAIT_FOR_OUTCOME(NO) option. The LUW_OUTCOME_MIXED subcode occurs if an operator somewhere in the distributed transaction forced local resources to commit.
7. The program can get the subcode LUW_OUTCOME_PENDING even when the local value of the WAIT_FOR_OUTCOME option on the SET_SYNCPT_OPTIONS verb is YES. This can occur if another program in the distributed transaction has set the WAIT_FOR_OUTCOME option to NO.

GET_TP_PROPERTIES

Returns information pertaining to the transaction program issuing the verb.

GET_TP_PROPERTIES	*Option No.* [*Base*]
Supplied Parameters:	
(*no parameters*)	
Returned Parameters:	
RETURN_CODE (*variable*)	
OWN_FULLY_QUALIFIED_LU_NAME (*variable*)	
OWN_TP_NAME (*variable*)	*109*
OWN_TP_INSTANCE (*variable*)	*109*
SECURITY_USER_ID (*variable*)	*212*
SECURITY_PROFILE (*variable*)	*215*
LUW_IDENTIFIER (*variable*)	*251*
PROTECTED_LUW_IDENTIFIER (*variable*)	*108, 251*

Supplied Parameters:

No Parameters are supplied for this verb.

Returned Parameters:

RETURN_CODE specifies the variable in which the result of the verb execution is indicated to the local program.

- OK
- PROGRAM_PARAMETER_CHECK, for one of the following reasons:
  - 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.
  - PROTECTED_LUW_IDENTIFIER is specified and not supported.

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

OWN_TP_NAME specifies the variable for returning the local program's own transaction program name.

OWN_TP_INSTANCE specifies the variable for returning the system-generated identifier for this instance of the transaction program. This instance identifier can be passed to other programs, which can use it to communicate with this transaction program instance.

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 used by the transaction program when it accesses unprotected resources. The LUW identifier is created and maintained by the LU. The LU uses it for accounting and network management purposes. If the LU has no LUW identifier for unprotected resources, a null value is returned.

PROTECTED_LUW_IDENTIFIER specifies the variable for returning the logical unit of work (LUW) identifier used by the transaction program when it accesses protected resources, such as conversations with SYNC_LEVEL(SYNCPT). The LUW identifier is created and maintained by the LU. The LU uses it for accounting and network management purposes, as well as to identify the current logical unit of work that will be committed or backed out by the next syncpoint operation. If the LU has no LUW identifier for protected resources, a null value is returned.

State Changes:

None

Notes:

1. The value returned in the PROTECTED_LUW_IDENTIFIER parameter can change with each BACKOUT or SYNCPT or DEALLOCATE TYPE(ABEND) verb.

GET_TYPE

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

GET_TYPE	*Option No.* [*110*]
Supplied Parameters: **RESOURCE** (*variable*)	
Returned Parameters: **RETURN_CODE** (*variable*) **TYPE** (*variable*)	

Supplied Parameters:

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

Returned Parameters:

RETURN_CODE specifies the variable in which the result of the verb execution is indicated to the local program.

- OK
- PROGRAM_PARAMETER_CHECK, for the following reason:
  - RESOURCE specifies an unassigned resource ID.

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

- BASIC_CONVERSATION
- MAPPED_CONVERSATION
- FULL_DUPLEX_BASIC_CONVERSATION
- FULL_DUPLEX_MAPPED_CONVERSATION

State Changes:

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.
2. A program that supports more than one conversation type should issue a GET_TYPE as the first verb when the program is started. The value returned for the TYPE parameter will indicate what type of conversation was started.

SET_SYNCPT_OPTIONS

Changes the options governing the execution of the SYNCPT, BACKOUT, PREPARE_FOR_SYNCPT, and MC_PREPARE_FOR_SYNCPT verbs. The default options are in effect when the transaction program starts. The options set by this verb remain in use until the verb is issued again or the transaction program ends. The program is not required to specify any parameter whose value it does not need to change.

SET_SYNCPT_OPTIONS	*Option No.* [*108*]
Supplied Parameters:	
(*No parameters*)	
OK_TO_LEAVE_OUT	*250*
(NO)	*250*
(YES)	*250*
SELECT_LAST_AGENT	*250*
(YES)	*250*
(NO)	*250*
VOTE_READ_ONLY_PERMITTED	
(NO)	
(YES)	*249*
WAIT_FOR_OUTCOME	
(YES)	
(NO)	
ACTION_IF_PROBLEMS	
(BACKOUT)	
(COMMIT)	
Returned Parameters:	
RETURN_CODE (*variable*)	

Supplied Parameters:

OK_TO_LEAVE_OUT indicates whether this program may be left out of subsequent units of work. This parameter should be NO if the program performs work independently, rather than responding to requests. If this parameter is set to YES, a SYNCPT or BACKOUT verb may not complete until a request arrives for a subsequent unit of work for the program.

SET_SYNCPT_OPTIONS

Changes the options governing the execution of the SYNCPT, BACKOUT, PREPARE_FOR_SYNCPT, and MC_PREPARE_FOR_SYNCPT verbs. The default options are in effect when the transaction program starts. The options set by this verb remain in use until the verb is issued again or the transaction program ends. The program is not required to specify any parameter whose value it does not need to change.

SET_SYNCPT_OPTIONS	*Option No.* [*108*]
Supplied Parameters:	
(*No parameters*)	
OK_TO_LEAVE_OUT	*250*
(NO)	*250*
(YES)	*250*
SELECT_LAST_AGENT	*250*
(YES)	*250*
(NO)	*250*
VOTE_READ_ONLY_PERMITTED	
(NO)	
(YES)	*249*
WAIT_FOR_OUTCOME	
(YES)	
(NO)	
ACTION_IF_PROBLEMS	
(BACKOUT)	
(COMMIT)	
Returned Parameters:	
RETURN_CODE (*variable*)	

Supplied Parameters:

OK_TO_LEAVE_OUT indicates whether this program may be left out of subsequent units of work. This parameter should be NO if the program performs work independently, rather than responding to requests. If this parameter is set to YES, a SYNCPT or BACKOUT verb may not complete until a request arrives for a subsequent unit of work for the program.

SELECT_LAST_AGENT indicates whether the last-agent optimization is to be used during the sync point operation. The last-agent optimization decreases the time required to perform the sync point operation by transferring the commit decision to a partner (called last agent). It is more advantageous when remote partners participate in the transaction.

VOTE_READ_ONLY_PERMITTED specifies whether the local LU may vote read only in a sync point operation if it has made no changes to protected resources and none of the resources subordinate to it in the distributed transaction have made any changes. Specifying VOTE_READ_ONLY_PERMITTED(YES) can decrease the time required to complete the sync point operation, but it means that the local TP will not learn whether the syncpoint operation resulted in the transaction being backed out or committed in the rest of the distributed transaction. If the local program specifies VOTE_READ_ONLY_PERMITTED(YES), it will receive VOTED_READ_ONLY in the secondary return code to the SYNCPT verb if no local or subordinate protected resources have been updated since the last synchronization point.

WAIT_FOR_OUTCOME specifies whether the outcome of the sync point operation at all subordinate resources in the distributed transaction must be known before control is returned to the program. If a failure occurs and the program has specified WAIT_FOR_OUTCOME(NO), control may be returned from the SYNCPT, BACKOUT, PREPARE_FOR_SYNCPT, or MC_PREPARE_FOR_SYNCPT verb with a secondary return code value of LUW_OUTCOME_PENDING, indicating that the outcome at one or more distributed partners is unknown. If the program has specified WAIT_FOR_OUTCOME(YES), control will not be returned to the program from these verbs until the outcome of the syncpoint operation at all subordinate resources is known.

Note that control may be returned before the outcome is fully known even if WAIT_FOR_OUTCOME(YES) is specified by the local program if other programs in the distributed transaction specified WAIT_FOR_OUTCOME(NO) or if the program changed the option between the (MC_)PREPARE_FOR_SYNCPT verb and the SYNCPT verb.

ACTION_IF_PROBLEMS specifies the action to be taken with protected resources if the LU learns of a problem at a point in the sync point operation when it does not know whether to commit or backout. Possible problems include sync point protocol violations or heuristic damage, such that some subordinate protected resources have been advanced to the next synchronization point and others have been restored to the previous synchronization point as a result of operator intervention.

Returned Parameters:

RETURN_CODE specifies the variable in which the result of the verb execution is indicated to the local program. The return codes that can be returned are:

- OK (the sync point options have been modified). This return code will be returned even if the program has no protected conversations.
- PROGRAM_PARAMETER_CHECK
  - An unsupported option is specified.

State Changes:

No state change occurs when the verb is issued.

Notes:

1. The program may issue SET_SYNCPT_OPTIONS at any time.
2. One of these options allows the program to specify information which the LU cannot know and which can reduce the time required to finish subsequent sync point operations. Thus the purpose of these options is to improve performance.
   - OK_TO_LEAVE_OUT provides the program with the option of specifying that it can be left out of any subsequent transaction. The program should only be set to YES if it is OK for the program to be suspended until the next transaction that requires its services. An example of a case where this option might be used is the case where the local program has a slave relationship to a conversation partner, such that if it does not receive anything from the partner it does nothing. In this case it is safe to leave it out.
3. Three of these options allow the program to specify the tradeoff it wishes to make between improved performance and some other facet of the syncpoint operation:
   - SELECT_LAST_AGENT allows the program to choose improved performance by reducing the number of flows needed to perform the sync point operation at the expense of parallelism. If the parameter is set to YES the system will first prepare for commitment all the not-last-agents and then transfer the commit decision to the chosen last agent instead of preparing all the agents in parallel.

     This optimization is advantageous if remote partners that performed updates participated in the transaction.
   - VOTE_READ_ONLY_PERMITTED allows the program to choose improved performance at the cost of not knowing the outcome of any syncpoint operation that occurs when it has not made changes to protected resources.
   - WAIT_FOR_OUTCOME set to NO allows the program to choose to have the sync point verb return control to the program before the outcome of the entire syncpoint operation is known. If this occurs, a message may be issued to the operator of the program's LU, but the program itself will not learn if damage to the consistency of the distributed transaction is subsequently detected.
   - References in this verb description to a program being in a particular state are only in terms of the specified conversation.

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, defer receive, or defer deallocate state.

SYNCPT	*Option No.* [*108*]
Supplied Parameters:	
(no parameters) **WAIT_OBJECT** *(BLOCKING)* *(VALUE (variable))*	▌*113*
Returned Parameters:	
RETURN_CODE (*variable*)	

Supplied Parameters:

WAIT_OBJECT specifies whether the verb should be processed in a blocking or nonblocking manner. The default value is a blocking verb operation. If a nonblocking verb operation is to be requested, then WAIT_OBJECT(VALUE(variable)) must be specified.

- **BLOCKING** specifies this is a blocking verb operation.
- **VALUE(*variable*)** specifies a nonblocking verb operation of one of two types:
  - a ***null*** variable specifies to complete the verb operation but without notification.
  - a nonnull ***variable*** specifies to complete the verb operation but with notification. The variable represents a wait object. Its use here associates it with completion of this verb. It can be listed as an entry in the WAIT_OBJECT_LIST in a subsequent WAIT_FOR_COMPLETION verb for notification (posting) purposes. Origination of this variable (by the TP or the operating system) is implementation dependent.

Returned Parameters:

RETURN_CODE specifies the variable in which the return code is returned to the transaction program. The return code indicates whether the sync point operation succeeded, and if so, the action taken by the local LU. The subcode under the OK

or BACKED_OUT return code indicates what occurred in the rest of the distributed transaction.

- OK (sync point resulted in committing local resources) When the return code is OK, the subcode indicates the state of the sync-point tree that is subordinate to the transaction program, i.e., the resources to which it propagated the sync-point operation by issuing the SYNCPT verb.
  - ALL_AGREED indicates that all protected resources in the distributed transaction to which this program has propagated the sync-point operation have been advanced to the next synchronization point.
  - VOTED_READ_ONLY indicates that all protected resources in the distributed transaction to which this program has propagated the syncpoint operation have voted read only, including the local resources. In order for a program to receive this return code, it must have previously issued the SET_SYNCPT_OPTIONS verb with VOTE_READ_ONLY_PERMITTED(YES).
  - LUW_OUTCOME_PENDING: As a result of failures and at least one program in the distributed transaction specifying SET_SYNCPT_OPTIONS WAIT_FOR_OUTCOME(NO), it is not known whether all subordinate protected resources have performed the same action performed by the local resources.
  - LUW_OUTCOME_MIXED indicates that some protected resources in the distributed transaction have been advanced to the next synchronization point and others have been restored to the previous synchronization point. This mixed status of protected resources occurs when an LU operator at one or more of the protected resources intervenes in order to free locks, forcing the resource to commit or backout. See *SNA LU 6.2 Reference: Peer Protocols* for more details.
- BACKED_OUT: The sync-point operation resulted in backing out protected resources. Since the resources have already been backed out when the SYNCPT verb completes with this return code, the conversations are not in backout required state.

  When the return code is BACKED_OUT, the subcode indicates the state of the sync-point tree that is subordinate to the transaction program, i.e., the resources to which it propagated the sync point operation by issuing the SYNCPT verb.
  - ALL_AGREED indicates that all protected resources in the distributed transaction to which this program has propagated the sync-point operation have backed out the transaction.
  - LUW_OUTCOME_PENDING: As a result of failures and at least one program in the distributed transaction specifying SET_SYNCPT_OPTIONS WAIT_FOR_OUTCOME(NO), it is not known whether all subordinate protected resources have backed out.
  - LUW_OUTCOME_MIXED indicates that some protected resources in the distributed transaction have been advanced to the next synchronization point and others have been restored to the previous synchronization point. This mixed status of protected resources occurs when an LU operator at one or more of the protected resources intervenes in order to free locks, forcing the resource to commit or backout. See *SNA LU 6.2 Reference: Peer Protocols* for more details.
- OPERATION_INCOMPLETE
- OPERATION_NOT_ACCEPTED

- PROGRAM_STATE_CHECK, for one of the following reasons:
  - A protected resource is not in send, defer-receive, defer-deallocate, 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.
  - A non-conversation protected resource is not in a state that permits a sync-point operation.

State Changes:

Reset state is entered when the verb is issued in the defer-deallocate state entered by the preceding (MC_)DEALLOCATE verb

Receive state is entered when the verb is issued in the defer-receive 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-send 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-deallocate 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.

State Changes (when RETURN_CODE indicates BACKED_OUT):

The conversation returns to the state at the end of the last synchronization point, i.e., the state at the end of the last SYNCPT verb if one has been executed since the start of the program, otherwise the state at the start of the program.

Notes:

1. The program may issue SYNCPT when all protected conversations are in send, defer-receive, defer-deallocate, sync-point, sync-point-send, sync-point-deallocate state, or a combination of these states; however, only one conversation can be in any 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 receive 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 deallocate 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 may 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-receive, defer-deallocate, sync-point, sync-point-send , sync-point-deallocate 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. Unless it has a value indicating that a parameter or state check occurred, RETURN_CODE indicates whether the local resources committed or backed out as a result of the sync-point operation. The RETURN_CODE subcode indicates the state of the sync-point tree that is subordinate to the transaction program, i.e., the resources to which it propagates the sync-point operation by issuing the SYNCPT verb.

5. Use of sync point ensures consistency of the protected resources involved in a distributed transaction unless an operator intervenes, forcing a protected resource to commit or backout without waiting for the sync-point operation to complete. Consistency means that if the return codes OK and ALL_AGREED are returned to the transaction program that issued the first SYNCPT verb (called the initiator), they will also have been returned to the dependent SYNCPT verbs issued by every other transaction program participating in the distributed logical unit of work.

   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.

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

WAIT

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	*Option No.* [*104*]
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 the result of the verb execution is indicated to the local program. 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
  - MAP_EXECUTION_FAILURE
  - MAP_NOT_FOUND
  - MAPPING_NOT_SUPPORTED
  - PROG_ERROR_NO_TRUNC

- PROG_ERROR_PURGING
- PROGRAM_PARAMETER_CHECK, for the following reason:
  - RESOURCE_LIST specifies an unassigned resource ID.
  - This verb is not supported for full-duplex conversations.
- PROGRAM_STATE_CHECK, for the following reason:
  - A conversation is not in receive state.
- RESOURCE_FAILURE_NO_RETRY
- RESOURCE_FAILURE_RETRY
- USER_CONTROL_DATA_NOT_SUPPORTED

• 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
- PROGRAM_PARAMETER_CHECK, for the following reason:
  - RESOURCE_LIST specifies an unassigned resource ID.
- PROGRAM_STATE_CHECK, for the following reason:
  - A conversation is not in receive state.
- 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:

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 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).
5. References in this verb description to a program being in a particular state are in terms of each conversation.